US 20100191961A1

(19) United States
(12) Patent Application Publication
Master
(10) Pub. No.: US 2010/0191961 A1
(43) Pub. Date: Jul. 29, 2010

(54) METHOD AND SYSTEM ACHIEVING INDIVIDUALIZED PROTECTED SPACE IN AN OPERATING SYSTEM

(75) Inventor: **Paul L. Master**, Sunnyvale, CA (US)

Correspondence Address:
NIXON PEABODY LLP
401 9TH STREET, N.W.
WASHINGTON, DC 20004 (US)

(73) Assignee: **QST Holdings, Inc.**, Palo Alto, CA (US)

(21) Appl. No.: **12/648,792**

(22) Filed: **Dec. 29, 2009**

Related U.S. Application Data

(63) Continuation of application No. 10/437,855, filed on May 13, 2003, now Pat. No. 7,660,984.

(60) Provisional application No. 60/380,877, filed on May 13, 2002.

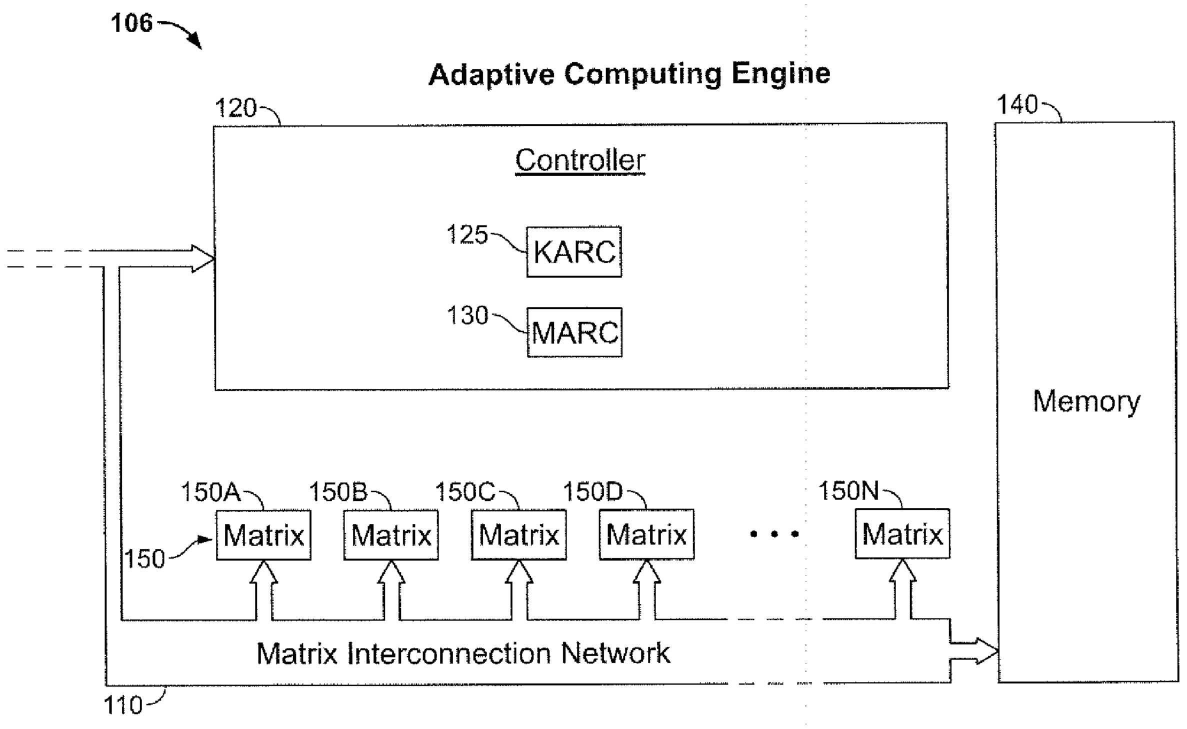
Publication Classification

(51) **Int. Cl.**
H04L 9/00 (2006.01)

(52) **U.S. Cl.** **713/164**

(57) **ABSTRACT**

Aspects for achieving individualized protected space in an operating system are provided. The aspects include performing on demand hardware instantiation via an ACE (an adaptive computing engine), and utilizing the hardware for monitoring predetermined software programming to protect an operating system.



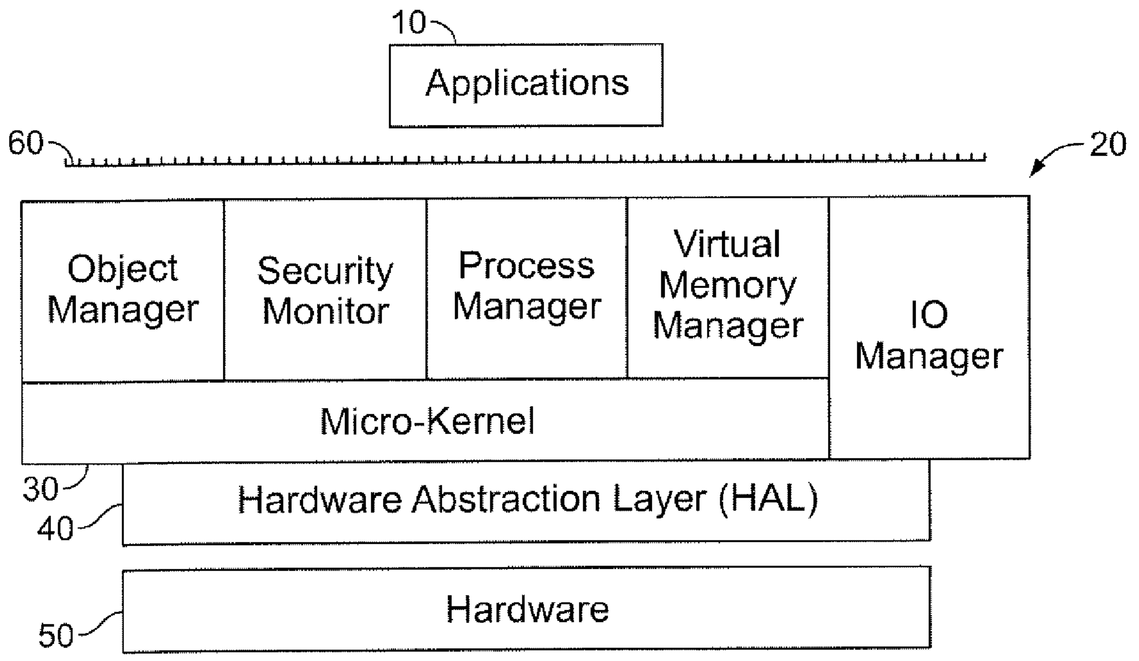


FIG. 1
(Prior Art)

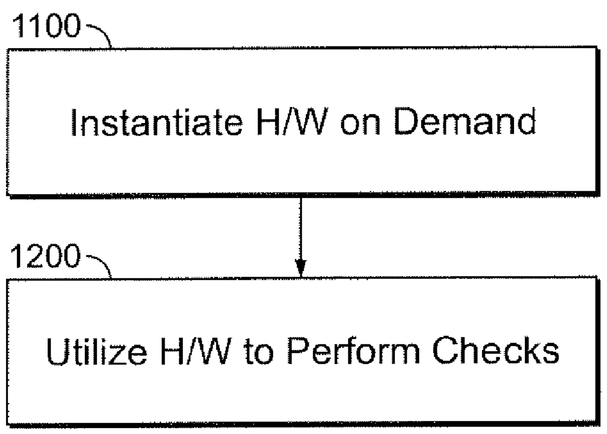


FIG. 4

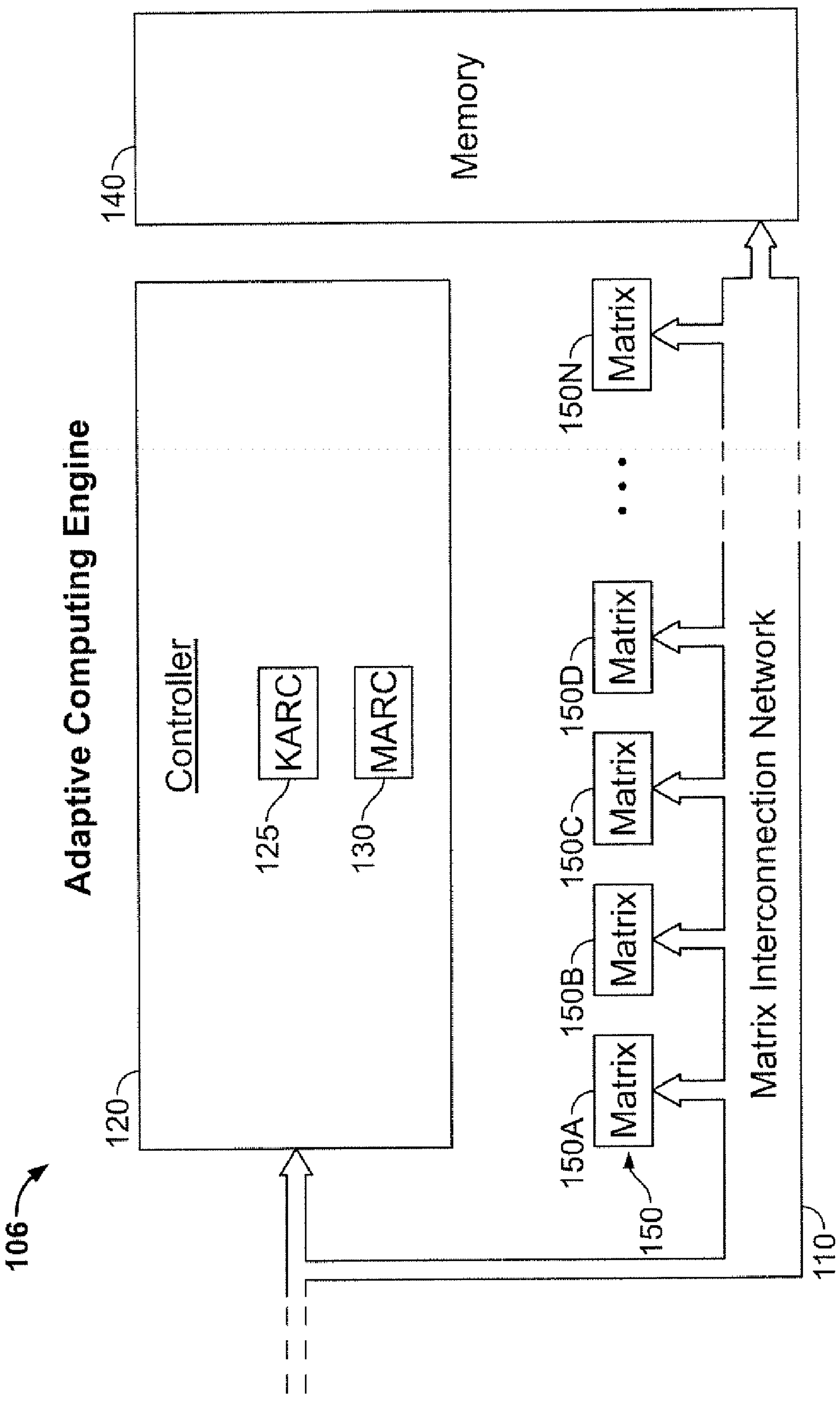

106
Adaptive Computing Engine
120
Controller
125
KARC
130
MARC
140
Memory
150
150A
Matrix
150B
Matrix
150C
Matrix
150D
Matrix
150N
Matrix
Matrix Interconnection Network
110

FIG. 2

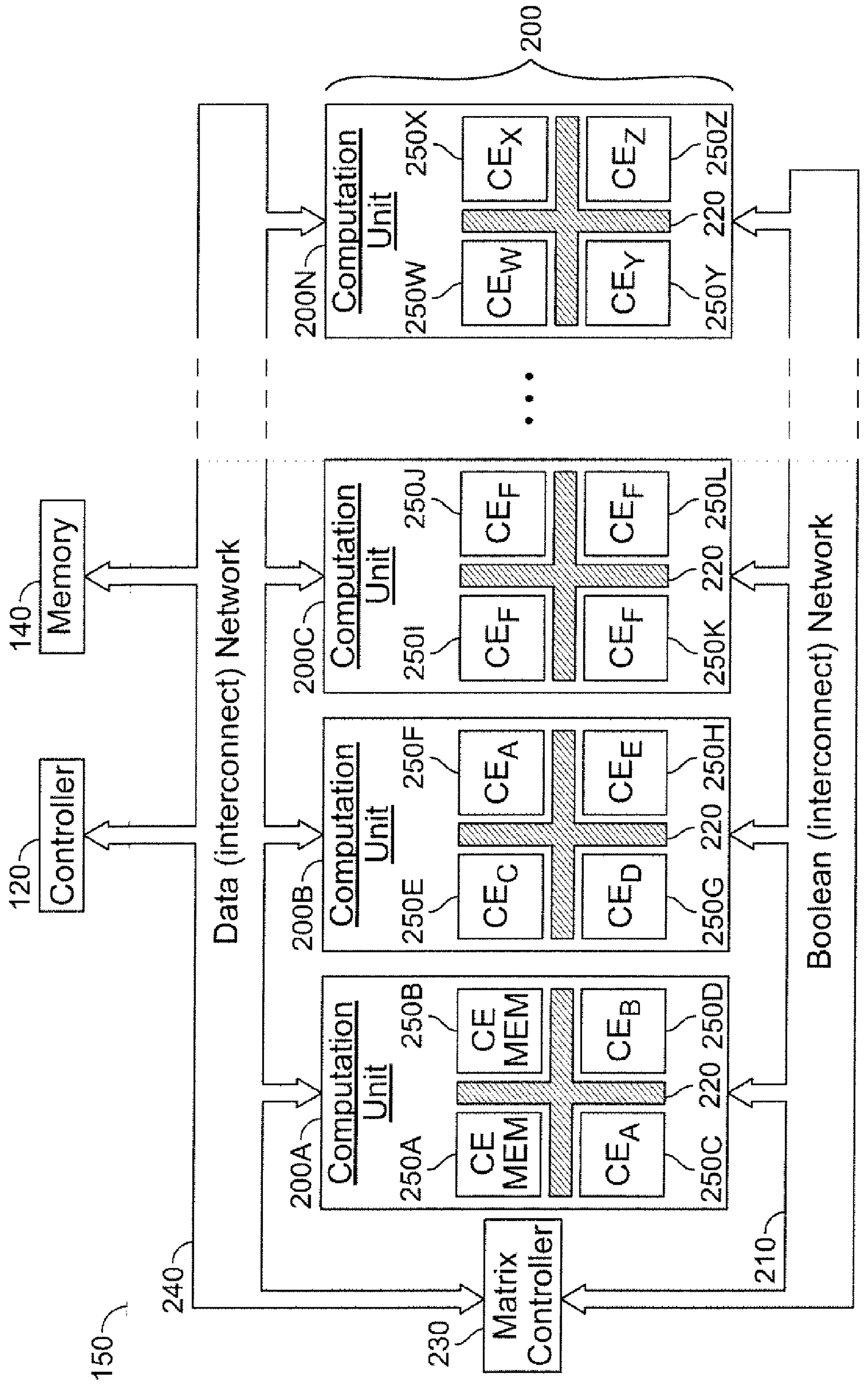

150
240
230
Matrix Controller
210
120
Controller
140
Memory
Data (interconnect) Network
Boolean (interconnect) Network
200
200A
200B
200C
200N
Computation Unit
250A
CE MEM
250B
CE MEM
250C
CE_A
250D
CE_B
250E
CE_C
250F
CE_A
250G
CE_D
250H
CE_E
250I
CE_F
250J
CE_F
250K
CE_F
250L
CE_F
250W
CE_W
250X
CE_X
250Y
CE_Y
250Z
CE_Z
220

FIG. 3

METHOD AND SYSTEM ACHIEVING INDIVIDUALIZED PROTECTED SPACE IN AN OPERATING SYSTEM

FIELD OF THE INVENTION

[0001] The present invention relates to robust operating system protection.

BACKGROUND OF THE INVENTION

[0002] As is generally understood in computing environments, an operating system (O/S) acts as the layer between the hardware and the software providing several important functions. For example, the functionality of an O/S includes device management, process management, communication between processes, memory management, and file systems. Further, certain utilities are standard for operating systems that allow common tasks to be performed, such as file access and organization operations and process initiation and termination.

[0003] Within the O/S, the kernel is responsible for all other operations and acts to control the operations following the initialization functions performed by the O/S upon boot-up. The traditional structure of a kernel is a layered system. Some operating systems use a micro-kernel to minimize a size of the kernel while maintaining a layered system, such as the Windows NT operating system. FIG. **1** illustrates an example diagram of a typical layered structure, such as for the Windows NT operating system. As shown, the applications **10** lie above the O/S **20**, where each application typically resides in its own memory space. The micro-kernel **30** interacts with a hardware abstraction layer **40** (e.g., with device drivers) associated with hardware layer **50**. The line **60** represents a demarcation line indicating the separation between which normally is considered the user space of the applications, and the protected space of the operating system.

[0004] While the typical structure provides a well-understood model for an operating system, some problems remain. One such problem is the potential for crashing the machine once access below the demarcation line **60** is achieved. For example, bugs in programs that are written for performing processes below the demarcation line, e.g., device drivers that interact with the hardware abstraction layer, protocol stacks between the kernel and the applications, etc., can bring the entire machine down. While some protection is provided in operating systems with the generation of exceptions in response to certain illegal actions, such as memory address violations or illegal instructions, which trigger the kernel and kill the application raising the exception, there exists an inability by operating systems to protect against the vulnerability to fatal access.

[0005] An approach to avoiding such vulnerability is to limit which software is trusted within an operating system and utilizing control mechanisms that check all other programming prior to processing. Relying on software to perform such checks reduces the ability to limit the amount of software that is trusted. A hardware solution would be preferable, but, heretofore, has been prohibitive due to the level of instantaneous hardware machine generation that would be necessary.

[0006] Accordingly, what is needed is an ability to achieve a protected operating system through on demand hardware monitoring. The present invention addresses such a need.

SUMMARY OF THE INVENTION

[0007] Aspects for achieving individualized protected space in an operating system are provided. The aspects include performing on demand hardware instantiation via an ACE (an adaptive computing engine), and utilizing the hardware for monitoring predetermined software programming to protect an operating system.

[0008] Through the present invention, all elements outside a system's own code for operating, e.g., all the stacks, abstraction layers, and device drivers, can be readily and reliably monitored. In this manner, the vulnerability present in most current operating systems due to unchecked access below the demarcation line is successfully overcome. Further, the reconfigurability of the ACE architecture allows the approach to adjust as desired with additions/changes to an operating system environment. These and other advantages will become readily apparent from the following detailed description and accompanying drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

[0009] FIG. **1** illustrates a diagram of operating system layers of the prior art.

[0010] FIG. **2** is a block diagram illustrating an adaptive computing engine.

[0011] FIG. **3** is a block diagram illustrating, in greater detail, a reconfigurable matrix of the adaptive computing engine.

[0012] FIG. **4** illustrates an overall block flow diagram illustrates a process for achieving individualized protected space in an operating system in accordance with the present invention.

DETAILED DESCRIPTION OF THE INVENTION

[0013] The present invention relates to achieving individualized protected space in an operating system via an adaptive computing engine (ACE). The following description is presented to enable one of ordinary skill in the art to make and use the invention and is provided in the context of a patent application and its requirements. Various modifications to the preferred embodiment and the generic principles and features described herein will be readily apparent to those skilled in the art. Thus, the present invention is not intended to be limited to the embodiment shown but is to be accorded the widest scope consistent with the principles and features described herein.

[0014] In a preferred embodiment, the processing core of an embedded system is achieved through an adaptive computing engine (ACE). A more detailed discussion of the aspects of an ACE are provided in co-pending U.S. patent application Ser. No. 10/384,486, entitled ADAPTIVE INTEGRATED CIRCUITRY WITH HETEROGENEOUS AND RECONFIGURABLE MATRICES OF DIVERSE AND ADAPTIVE COMPUTATIONAL UNITS HAVING FIXED, APPLICATION SPECIFIC COMPUTATIONAL ELEMENTS, filed Mar. 7, 2003, assigned to the assignee of the present invention, and incorporated herein in its entirety. Generally, the ACE provides a significant departure from the prior art for achieving processing in an embedded system, in that data, control and configuration information are transmitted between and among its elements, utilizing an interconnection network, which may be configured and reconfigured, in real-time, to provide any given connection between and among the elements. In order to more fully illustrate the aspects of the present invention, portions of the discussion of the ACE from the application incorporated by reference are included in the following.

[0015] FIG. **2** is a block diagram illustrating an adaptive computing engine ("ACE") **106** that includes a controller **120**, one or more reconfigurable matrices **150**, such as matrices **150**A through **150**N as illustrated, a matrix interconnection network **110**, and preferably also includes a memory **140**.

[0016] The controller **120** is preferably implemented as a reduced instruction set ("RISC") processor, controller or other device or IC capable of performing the two types of functionality discussed below. The first control functionality, referred to as "kernal" control, is illustrated as kernal controller ("KARC") **125**, and the second control functionality, referred to as "matrix" control, is illustrated as matrix controller ("MARC") **130**.

[0017] FIG. **3** is a block diagram illustrating, in greater detail, a reconfigurable matrix **150** with a plurality of computation units **200** (illustrated as computation units **200**A through **200**N), and a plurality of computational elements **250** (illustrated as computational elements **250**A through **250**Z), and provides additional illustration of the preferred types of computational elements **250** and a useful summary of aspects of the present invention. As illustrated in FIG. **3**, any matrix **150** generally includes a matrix controller **230**, a plurality of computation (or computational) units **200**, and as logical or conceptual subsets or portions of the matrix interconnect network **110**, a data interconnect network **240** and a Boolean interconnect network **210**. The Boolean interconnect network **210** provides the reconfigurable interconnection capability between and among the various computation units **200**, while the data interconnect network **240** provides the reconfigurable interconnection capability for data input and output between and among the various computation units **200**. It should be noted, however, that while conceptually divided into reconfiguration and data capabilities, any given physical portion of the matrix interconnection network **110**, at any given time, may be operating as either the Boolean interconnect network **210**, the data interconnect network **240**, the lowest level interconnect **220** (between and among the various computational elements **250**), or other input, output, or connection functionality.

[0018] Continuing to refer to FIG. **3**, included within a computation unit **200** are a plurality of computational elements **250**, illustrated as computational elements **250**A through **250**Z (collectively referred to as computational elements **250**), and additional interconnect **220**. The interconnect **220** provides the reconfigurable interconnection capability and input/output paths between and among the various computational elements **250**. Each of the various computational elements **250** consist of dedicated, application specific hardware designed to perform a given task or range of tasks, resulting in a plurality of different, fixed computational elements **250**. Utilizing the interconnect **220**, the fixed computational elements **250** may be reconfigurably connected together to execute an algorithm or other function, at any given time.

[0019] In a preferred embodiment, the various computational elements **250** are designed and grouped together, into the various reconfigurable computation units **200**. In addition to computational elements **250** which are designed to execute a particular algorithm or function, such as multiplication, other types of computational elements **250** are also utilized in the preferred embodiment. As illustrated in FIG. **3**, computational elements **250**A and **250**B implement memory, to provide local memory elements for any given calculation or processing function (compared to the more "remote" memory **140**). In addition, computational elements **250**I, **250**J, **250**K and **250**L, are configured (using, for example, a plurality of flip-flops) to implement finite state machines, to provide local processing capability, especially suitable for complicated control processing.

[0020] With the various types of different computational elements **250** which may be available, depending upon the desired functionality of the ACE **106**, the computation units **200** may be loosely categorized. A first category of computation units **200** includes computational elements **250** performing linear operations, such as multiplication, addition, finite impulse response filtering, and so on. A second category of computation units **200** includes computational elements **250** performing non-linear operations, such as discrete cosine transformation, trigonometric calculations, and complex multiplications. A third type of computation unit **200** implements a finite state machine, such as computation unit **200**C as illustrated in FIG. **3**, particularly useful for complicated control sequences, dynamic scheduling, and input/output management, while a fourth type may implement memory and memory management, such as computation unit **200**A as illustrated in FIG. **3**. Lastly, a fifth type of computation unit **200** may be included to perform bit-level manipulation, such as for encryption, decryption, channel coding, Viterbi decoding, and packet and protocol processing (such as Internet Protocol processing).

[0021] The ability to configure the elements of the ACE relies on a tight coupling (or interdigitation) of data and configuration (or other control) information, within one, effectively continuous stream of information. The continuous stream of data can be characterized as including a first portion that provides adaptive instructions and configuration data and a second portion that provides data to be processed. This coupling or comingling of data and configuration information, referred to as a "silverware" module, helps to enable real-time reconfigurability of the ACE **106**, and in conjunction with the real-time reconfigurability of heterogeneous and fixed computational elements **250**, to form different and heterogenous computation units **200** and matrices **150**, enables the ACE **106** architecture to have multiple and different modes of operation. For example, when included within a hand-held device, given a corresponding silverware module, the ACE **106** may have various and different operating modes as a cellular or other mobile telephone, a music player, a pager, a personal digital assistant, and other new or existing functionalities. In addition, these operating modes may change based upon the physical location of the device; for example, when configured as a CDMA mobile telephone for use in the United States, the ACE **106** may be reconfigured as a GSM mobile telephone for use in Europe.

[0022] As an analogy, for the reconfiguration possible via the silverware modules, a particular configuration of computational elements, as the hardware to execute a corresponding algorithm, may be viewed or conceptualized as a hardware analog of "calling" a subroutine in software which may perform the same algorithm. As a consequence, once the configuration of the computational elements has occurred, as directed by the configuration information, the data for use in the algorithm is immediately available as part of the silverware module. The immediacy of the data, for use in the configured computational elements, provides a one or two clock cycle hardware analog to the multiple and separate software steps of determining a memory address and fetching stored data from the addressed registers.

[0023] Referring again to FIG. **2**, the functions of the KARC **125** may be explained with reference to a silverware module. As indicated above, through a silverware module, the ACE **100** may be configured or reconfigured to perform a new or additional function, such as an upgrade to a new technology standard or the addition of an entirely new function, such as the addition of a music function to a mobile communication device. Such a silverware module may be stored in memory **140**, or may be input from an external (wired or wireless) source through, for example, matrix interconnection network **110**.

[0024] While the ability to configure and reconfigure computational elements in real-time is achieved through the ACE, the present invention applies that ability to provide a more robust operating system configuration. In accordance with the present invention, a core amount of programming, such as the kernel space, is the only so-called trusted space within the operating system. All other elements of the operating system that normally would fall within the protected space of the operating system model now receive individualized monitoring. Referring to FIG. **4**, an overall block flow diagram illustrates a process for achieving individualized protected space in an operating system in accordance with the present invention. As shown in FIG. **4**, the process initiates with on demand instantiation of hardware via the ACE in response to a processing call outside of the trusted space of operating system programming (step **1100**). By way of example, when a device driver is input via a silverware module to perform a function on behalf of the operating system, such as a SCSI driver to perform a data transfer to memory, a hardware "machine" is formed by appropriate computational elements. Thus, in the preferred embodiment, one of the matrices **150** is configured to decrypt a module and verify its validity, for security purposes. Next, the machine then monitors the operations of the processing (step **1200**). Thus prior to any configuration or reconfiguration of existing ACE **100** resources, the controller **120**, through the KARC **125**, checks and verifies that the configuration or reconfiguration may occur without adversely affecting any pre-existing functionality. In the SCSI driver example, the machine is configured to perform several checks to protect against invalid operations by the device driver. For example, the machine performs address checking, i.e., it ensures that the device driver interacts with a valid memory address range associated with that driver. The machine may also monitor for resource restriction violations, i.e., it ensures that limits on transfer time are not violated. Additionally, the protocol for the processing is monitored, i.e., it ensures that the hardware interacted with is left in a good state through proper communication of 'start', 'stop', and 'end' signalling. Of course, other types of monitoring may be performed as needed for particular elements, as is well appreciated by those skilled in the art.

[0025] In the preferred embodiment, the system requirements for the configuration or reconfiguration are included within the silverware module for use by the KARC **125** in performing this evaluative function. If the configuration or reconfiguration may occur without adverse affects, the silverware module is allowed to load into memory **140**, with the KARC **125** setting up the DMA engines within the memory **140**. If the configuration or reconfiguration would or may have such adverse affects, the KARC **125** does not allow the new module to be incorporated within the ACE **100**.

[0026] Basic operations that device drivers perform can be broken down into:

[0027] Memory Reads and Writes
- **[0028]** Reads and Writes to main memory address space to set, clear and check status of CSR (control status registers) of devices
- **[0029]** Reads and Writes to Input/Output address space to set, clear and check status of CSR (control status registers) of devices

[0030] Hardware Interrupts
- **[0031]** Setting up interrupt vectors to point to an interrupt service routine
- **[0032]** Servicing interrupt
- **[0033]** Disabling and enabling interrupts
- **[0034]** Setting and Clearing an interrupt

[0035] Direct Memory Addressing (DMA)
- **[0036]** Setting up a DMA transfer by Memory Reads and Writes to DMA CSRs or Memory Mapped CSRs
- **[0037]** Setting Callback routine to be executed when DMA completes
- **[0038]** Setting Interrupt level to be asserted when DMA completes
- **[0039]** Setting up Memory Tables for scatter and gather operations by reads and writes

[0040] Computational Cycles
- **[0041]** Execution of device driver code consumes clock cycles of some processor

[0042] Memory Utilization
- **[0043]** Device driver code requires a certain amount of memory for temporary buffers, scratch pad working space, stacks, constants, data buffers, control sequences, etc. . . .

[0044] Bandwidth
- **[0045]** Device driver code requires a certain amount of bandwidth, typically bus bandwidth, link bandwidth, bandwidth between computation units such as register files, memories, hardware units, as well as bandwidth between low level component building blocks required to construct larger structures such as multipliers, adders, shifters, etc. . . .

Depending on the nature of the device driver, the physical characteristics of the hardware under control of the device driver some to all of the above operations are utilized. Device driver code which has defects (bugs) either intentionally (as in virus) or un-intentionally can effect the system the device driver is installed since device drivers run at the protected kernel level and can thus effect the integrity of the system leading to crashes, freezes, failure to perform as specified, as well as unintentional side effects of other software and hardware in the system.

[0046] In an ACE system, with the ability to construct specialized hardware from lower level building blocks a device driver can be "protected" by uniquely special hardware to protect the system from the device driver. Thus there is no need to trust that the device driver will perform as specified, hardware will ensure that device driver performs correctly. On failure an exception is generated to the OS indicating the failure condition as well as the specific device driver that failed. The OS then has the ability to either terminated the device driver, restart the device driver, resume the device driver from a check pointed (device driver may occasionally save state and thus has a copy of a known good configuration) copy of the device driver, pass the exception upwards to be handled at a higher system level, or even notify the user and request corrective action. Specifically for each of the above basic device operators the ACE can:

[0047] Memory Reads and Writes

[0048] Reads and Writes to main memory address space to set, clear and check status of CSR (control status registers) of devices

[0049] The ACE produces a hardware memory range checking hardware to insure that the address of the memory read/writes are allowed and do not touch any memory that is out of bounds or range. This can range from sophistication from a simple address range checker (ALU) to multiple addresses for scattered CSR addresses (sophisticated multiple ALUs to perform in parallel range checking as well as insuring either read or write protection) to a full Customized MMU (memory management unit for block based address checking). Multiple address checking allows very specific and customized protection above and beyond what traditional MMU systems can provide.

[0050] Reads and Writes to Input/Output address space to set, clear and check status of CSR (control status registers) of devices

[0051] The ACE produces a hardware memory range checking hardware to insure that the address of the Input/Output (I/O) read/writes are allowed and do not touch any memory that is out of bounds or range. This can range from sophistication from a simple address range checker (ALU) to multiple addresses for scattered CSR addresses (sophisticated multiple ALUs to perform in parallel range checking as well as insuring either read or write protection) to a full Customized MMU (memory management unit for block based address checking). Multiple address checking allows very specific and customized protection above and beyond what traditional MMU systems can provide.

[0052] Hardware Interrupts

[0053] Setting up interrupt vectors to point to an interrupt service routine

[0054] The ACE can adapt hardware to produce hardware protection checking to insure that only a specific vector or group of specific vectors may be read or written.

[0055] Servicing interrupt

[0056] The ACE can adapt hardware to produce hardware protection checking to insure that if the device driver does not service the interrupt that a hardware default device driver is executed.

[0057] Disabling and enabling interrupts

[0058] The ACE can adapt hardware to produce hardware protection checking to insure that if the device driver can only enable or disable the interrupt that it has permission for.

[0059] Setting and Clearing an interrupt

[0060] The ACE can adapt hardware to produce hardware protection checking to insure that only the specific CSR bits are read or written by the device driver. In addition, if required, a watchdog timer can be configured to insure that strict timing durations are met in terms of duration of interrupt allowed.

[0061] Direct Memory Addressing (DMA)

[0062] Setting up a DMA transfer by Memory Reads and Writes to DMA CSRs or Memory Mapped CSRs

[0063] The ACE can adapt hardware to produce hardware protection checking to insure that only the specific CSRs or portions of CSRs as well as read/write protection is allowed by the device driver.

[0064] Setting Callback routine to be executed when DMA completes

[0065] The ACE can adapt hardware to produce hardware protection checking to insure that no other code can change the callback routine address to insure that the specific device driver intended to be called back is.

[0066] Setting Interrupt level to be asserted when DMA completes

[0067] The ACE can adapt hardware to produce hardware watchdog timers to insure that the DMA completes.

[0068] Setting up Memory Tables for scatter and gather operations by reads and writes

[0069] The ACE can adapt hardware to produce hardware protection checking to insure that the specific addresses (either in memory or I/O space) are accessed thereby precluding the device driver from accessing memory that it does not have authorization for.

[0070] Computational Cycles

[0071] Execution of device driver code consumes clock cycles of some processor

[0072] The ACE can adapt hardware to produce hardware cycle count checking to insure that the device driver does not exceed the specified maximum number of cycles. This can be used to terminate run-away tasks, or operations that are taking too long and may begin to effect system operation.

[0073] Memory Utilization

[0074] Device driver code requires a certain amount of memory for temporary buffers, scratch pad working space, stacks, constants, data buffers, control sequences, etc. . . .

[0075] The ACE produces a hardware memory range checking hardware to insure that the address of the memory read/writes are allowed and do not touch any memory that is out of bounds or range. This can range from sophistication from a simple address range checker (ALU) to multiple addresses for scattered CSR addresses (sophisticated multiple ALUs to perform in parallel range checking as well as insuring either read or write protection) to a full Customized MMU (memory management unit for block based address checking). Multiple address checking allows very specific and customized protection above and beyond what traditional MMU systems can provide.

[0076] This may include if required hardware resource checking on the amount of memory space used to see if it will exceed a maximum specified limit (for example if the upper limit on stack space is exceeded)

[0077] Bandwidth

[0078] Device driver code requires a certain amount of bandwidth, typically bus bandwidth, link bandwidth, bandwidth between computation units such as register files, memories, hardware units, as well as bandwidth between low level component building blocks required to construct larger structures such as multipliers, adders, shifters, etc. . . .

[0079] The ACE produces a hardware bandwidth checker to insure that the specified amount of bandwidth used on the MIN is not exceeded. This can be as simple as a total number of bytes transferred limit, to an average rate not to exceed limit.

The advantage here is that only the hardware protection that is required for a particular execution of the device driver needs to consume resources. For example, if no DMA is used then no ACE circuitry protecting the DMA is required. Even more resource efficient is if between to different calls to the device driver which use differing levels of operators then only the exact hardware protection is required—e.g. in a single execution no I/O read/writes are used and thus no hardware protection is required, in a second execution there is I/O read/writes and thus hardware protection is instantiated (hardware is configured and reconfigured from lower building block hardware to construct the exact hardware that is required). In a conventional hardware architecture without the ability to reconfigure the hardware the overhead for all this protection circuitry must be paid—by using only what is required during a particular time window (or execution) the ACE can provide exactly what is needed.

[0080] Thus, through the present invention, all elements outside a system's own code for operating, e.g., all the stacks, abstraction layers, and device drivers, can be readily and reliably monitored. In this manner, the vulnerability present in most current operating systems due to unchecked access below the demarcation line is successfully overcome. Further, the reconfigurability of the ACE architecture allows the approach to adjust as desired with additions/changes to an operating system environment.

[0081] From the foregoing, it will be observed that numerous variations and modifications may be effected without departing from the spirit and scope of the novel concept of the invention. Further, it is to be understood that no limitation with respect to the specific methods and apparatus illustrated herein is intended or should be inferred. It is, of course, intended to cover by the appended claims all such modifications as fall within the scope of the claims.

What is claimed is:

1. A method for achieving individualized protected space in an operating system, the method comprising the steps of:

(a) instantiating hardware on demand via an an adaptive computing engine (ACE); and

(b) utilizing the hardware for monitoring predetermined software programming to protect an operating system.

2. The method of claim **1** wherein the utilizing step (b) further comprises the step of (b1), utilizing the hardware to perform memory address range checking.

3. The method of claim **1** wherein the utilizing step (b) further comprises the step of (b1) utilizing the hardware to perform resource restriction checking.

4. The method of claim **3** wherein the resource restriction further comprises a time duration restriction.

5. The method of claim **1** wherein the utilizing step (b) further comprises the step of (b1) monitoring protocol processing programming.

6. The method of claim **1** wherein the utilizing step (b) further comprises the step of (b1) monitoring device driver programming.

7. The method of claim **1** wherein the utilizing step (b) further comprises the step of (b1) monitoring hardware abstraction layer programming.

8. The method of claim **1** wherein step (b) further comprises the step of (b1), utilizing the hardware to monitor only of the portions of the software programming that is used from run to run.

9. A system for achieving individualized protected space in an operating system, the system comprising:

memory for storing trusted operating system programming; and

an adaptive computing engine (ACE) for providing on demand hardware instantiation to individually monitor predetermined software programming interacting with the trusted operating system programming.

10. The system of claim **9** wherein the trusted operating system programming further comprises a kernel level of programming.

11. The system of claim **9** wherein the predetermined software programming further comprises device drivers.

12. The system of claim **9** wherein the predetermined software programming further comprises abstraction layers.

13. The system of claim **9** wherein the predetermined software programming further comprises communication protocol stacks.

14. The system of claim **9** wherein the on-demand hardware instantiation performs memory address range checking.

15. The system of claim **9** wherein the on-demand hardware instantiation performs resource restriction checking.

16. The system of claim **15** wherein the resource restriction further comprises a time duration restriction.

17. A system for achieving individualized protected space in an operating system, the system comprising:

memory for storing trusted operating system programming, wherein the trusted operating system programming further comprises a kernel level of programming and a plurality of device drivers; and

an adaptive computing engine (ACE) for providing on demand hardware instantiation to individually monitor predetermined software programming interacting with the trusted operating system programming, wherein the predetermined software programming further comprises abstraction layers and communication protocol stacks wherein the on-demand hardware instantiation performs memory address range checking and resource restriction checking; and wherein the resource restriction further comprises a time duration restriction.

* * * * *