(54) Title: HIERARCHICAL DYNAMIC SCHEDULING
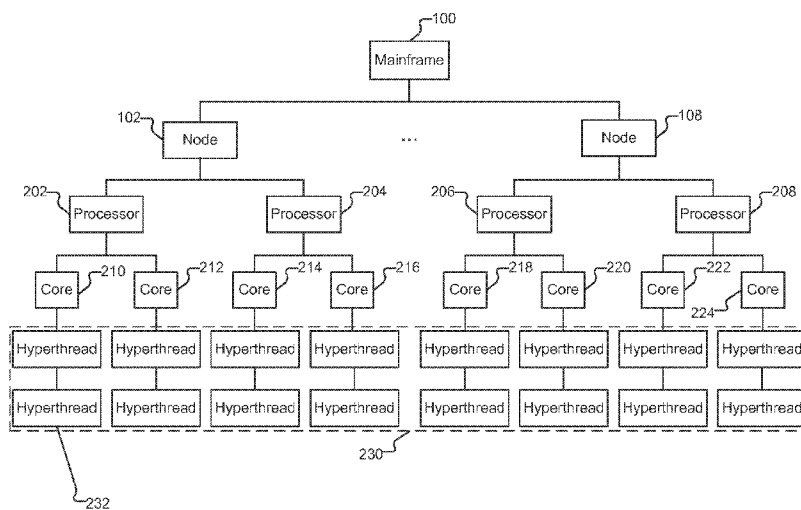
FIG. 2

(57) Abstract: Hierarchical dynamic scheduling is disclosed. A plurality of physical nodes is included in a computer system. Each node includes a plurality of processors. Each processor includes a plurality of hyperthreads. An abstraction of the nodes, processors, and hyperthreads forms a hierarchy. Upon receiving an indication that a hyperthread should be assigned, a dynamic search of the hierarchy is performed, beginning at the leaf level, for a process to assign to the hyperthread.

WO 2015/034508 A1

HIERARCHICAL DYNAMIC SCHEDULING

BACKGROUND OF THE INVENTION

[0001]   Software applications are increasingly operating on large sets of data and themselves becoming increasingly complex. In some cases, distributed computing systems are used to support such applications (e.g., where a large database system distributes portions of data onto a landscape of different server nodes, and optimizes queries into sub-queries that get distributed across that landscape). Unfortunately, significant effort has to be spent managing that distribution both in terms of data placement and data access distribution methods, including the complexities of networking. If the landscape changes, if the data organization changes, or if the workload changes, significant work will be required. More generally, the behavior of complex computing systems changes over time, e.g., with new releases of applications, the addition of new intermediate software layers, new operating system releases, new processor models, and changing structural characteristics of data, increasing amounts of data, and different data access patterns.

BRIEF DESCRIPTION OF THE DRAWINGS

[0002]   Various embodiments of the invention are disclosed in the following detailed description and the accompanying drawings.

[0003]   Figure 1 illustrates an embodiment of a computer system.

[0004]   Figure 2 illustrates the physical structure of the computer system as a hierarchy.

[0005]   Figure 3A depicts a virtualized computing environment in which multiple virtual machines (with respective multiple guest operating systems) run on a single physical machine.

[0006]   Figure 3B depicts a virtualized computing environment in which multiple physical machines collectively run a single virtual operating system.

[0007]   Figure 4A depicts an example of a software stack.

[0008]   Figure 4B depicts an example of a software stack.

[0009]   Figure 5 depicts an example of an operating system's view of hardware on an example system.

[0010]   Figure 6A depicts an example of a hyperthread's view of hardware on a single node.

[0011]   Figure 6B depicts an example of a HyperKernel's view of hardware on an example system.

[0012]   Figure 7 depicts an example of an operating system's view of hardware on an example of an enterprise supercomputer system.

[0013]   Figure 8 illustrates an embodiment of a process for selectively migrating resources.

[0014]   Figure 9 illustrates an embodiment of a process for performing hierarchical dynamic scheduling.

[0015]   Figure 10 illustrates an example of an initial memory assignment and processor assignment.

[0016]   Figure 11 illustrates an updated view of the memory assignment and an unchanged view of the processor assignment.

[0017]   Figure 12 illustrates a memory assignment and an updated view of the processor assignment.

DETAILED DESCRIPTION

[0018]   The invention can be implemented in numerous ways, including as a process; an apparatus; a system; a composition of matter; a computer program product embodied on a computer readable storage medium; and/or a processor, such as a processor configured to execute instructions stored on and/or provided by a memory coupled to the processor. In this specification, these implementations, or any other form that the invention may take, may be referred to as techniques. In general, the order of the steps of disclosed processes may be altered within the scope of the invention. Unless stated otherwise, a component such as a processor or a memory described as being configured to perform a task may be implemented as a general component that is temporarily configured to perform the task at a given time or a specific component that is manufactured to perform the task. As used herein, the term 'processor' refers to one or more devices, circuits, and/or processing cores configured to process data, such as computer program instructions.

[0019]   A detailed description of one or more embodiments of the invention is provided below along with accompanying figures that illustrate the principles of the invention. The

invention is described in connection with such embodiments, but the invention is not limited to any embodiment. The scope of the invention is limited only by the claims and the invention encompasses numerous alternatives, modifications and equivalents. Numerous specific details are set forth in the following description in order to provide a thorough understanding of the invention. These details are provided for the purpose of example and the invention may be practiced according to the claims without some or all of these specific details. For the purpose of clarity, technical material that is known in the technical fields related to the invention has not been described in detail so that the invention is not unnecessarily obscured.

[0020]   Figure 1 illustrates an embodiment of a computer system. System 100 is also referred to herein as an "enterprise supercomputer" and a "mainframe." In the example shown, system 100 includes a plurality of nodes (e.g., nodes 102-108) located in close proximity (e.g., located within the same rack). In other embodiments, multiple racks of nodes (e.g., located within the same facility) can be used in the system. Further, the techniques described herein can also be used in conjunction with distributed systems.

[0021]   The nodes are interconnected with a high-speed interconnect (110) such as 10-gigabit Ethernet, direct PCI-to-PCI, and/or InfiniBand. Each node comprises commodity server-class hardware components (e.g., a blade in a rack with its attached or contained peripherals). In the example shown in Figure 1, each node includes multiple physical processor chips. Each physical processor chip (also referred to as a "socket") includes multiple cores, and each core has multiple hyperthreads.

[0022]   As illustrated in Figure 2, the physical structure of system 100 forms a hierarchy (from the bottom) of hyperthreads (230), cores (210-224), physical processor chips (202-208), and nodes (102-108 (with nodes 104, 106, etc. omitted from the figure and represented as ellipses)). The tree depicted in Figure 2 is of a fixed size, defined by the hardware configuration.

[0023]   As will be described in more detail below, each enterprise supercomputer (e.g., system 100) runs a single instance of an operating system. Both the operating system, and any applications, can be standard commercially available software and can run on system 100. In the examples described herein, the operating system is Linux, however other operating systems can also be used, such as Microsoft Windows, Mac OS X, or FreeBSD.

[0024]   In a traditional virtualized computing environment, multiple virtual machines may run on a single physical machine. This scenario is depicted in Figure 3A. In particular, three

virtual machines (302-306) are running three guest operating systems on a single physical machine (308), which has its own host operating system. In contrast, using the techniques described herein, multiple physical machines (354-358) collectively run a single virtual operating system (352), as depicted in Figure 3B.

[0025]   One example of a software stack is depicted in Figure 4A. Such a stack may typically be used in traditional computing environments. In the stack shown in Figure 4A, an application (402) sits above a database engine (404), which in turn sits upon an operating system (406), underneath which lies hardware (408). Figure 4B depicts a software stack used in some embodiments. As with the stack shown in Figure 4A, an application (452) sits above a database engine (454), which in turn sits upon an operating system (456). However, underneath the operating system and above the hardware is a layer of software (referred to herein as a HyperKernel) that observes the system running in real time and optimizes the system resources to match the needs of the system as it operates. The HyperKernel conceptually unifies the RAM, processors, and I/O (Input Output resources for example Storage, Networking resources) of a set of commodity servers, and presents that unified set to the operating system. Because of this abstraction, the operating system will have the view of a single large computer, containing an aggregated set of processors, memory, and I/O. As will be described in more detail below, the HyperKernel optimizes use of resources. The HyperKernel can also help optimize other I/O system resources such as networks and storage. In some embodiments, based on observations and profiles of running software, performance indicators (hints) are provided to upper layers (e.g., database management systems) about the dynamic performance of the system that can further improve overall system performance.

[0026]   The HyperKernel can be ported to all major microprocessors, memory, interconnect, persistent storage, and networking architectures. Further, as hardware technology evolves (e.g., with new processors, new memory technology, new interconnects, and so forth), the HyperKernel can be modified as needed to take advantage of industry evolution.

[0027]   As shown in Figure 4B, operating system 456 is running collectively across a series of nodes (458-462), each of which has a HyperKernel running on server hardware. Specifically, the operating system is running on a virtual environment that is defined by the collection of HyperKernels. As will be described in more detail below, the view for operating system 456 is that it is running on a single hardware platform that includes all of the hardware resources of the individual nodes 458-462. Thus, if each of the nodes includes 1 TB of RAM, the operating system will have as a view that it is running on a hardware platform that includes 3 TB of RAM. Other

resources, such as processing power, and I/O resources can similarly be collectively made available to the operating system's view.

[0028]   Figure 5 depicts an example of an operating system's view of hardware on an example system. Specifically, operating system (502) runs on top of processors 504-508 and physical shared memory 510. As explained above, an operating system can run on either a traditional computing system or on an enterprise supercomputer such as is shown in Figure 1. In either case, the view of the operating system will be that it has access to processors 504-508 and physical shared memory 510.

[0029]   Figure 6A depicts an example of a hyperthread's view of hardware on a single node. In this example, a node has four hyperthreads denoted H1 (602) through H4 (608). Each hyperthread can access all portions of physical shared memory 612. Physical shared memory 612 is linear, labeled location 0 through a maximum amount, "max." The node also includes three levels of cache (610).

[0030]   Figure 6B depicts an example of a HyperKernel's view of hardware on an example system. In this example, three nodes (652-656) are included in an enterprise supercomputer. Each of the three nodes has four hyperthreads, a physical shared memory, and cache (i.e., each node is an embodiment of node 600 shown in Figure 6A). A hyperthread on a given node (e.g., node 652) has a view that is the same as that shown in Figure 6A. However, the HyperKernel is aware of all of the resources on all of the nodes (i.e., the HyperKernel sees twelve hyperthreads, and all of the physical shared memory). In the example shown in Figure 6B, a given hyperthread (e.g., hyperthread 658, "H1-4") is labeled with its node number (e.g., "1") followed by a hyperthread number (e.g., "4").

[0031]   Figure 7 depicts an example of an operating system's view of hardware on an example of an enterprise supercomputer system. The operating system sees a plurality of "virtualized processors" denoted in Figure 7 as P1 through Pmax (702). The virtualized processors correspond to the total number of hyperthreads across all nodes included in the enterprise supercomputer. Thus, using the example of Figure 6B, if a total of twelve hyperthreads are present across three nodes, a total of twelve virtualized processors would be visible to an operating system running on the enterprise supercomputer. The operating system also sees "virtualized physical memory" (704) that appears to be a large, physical, linear memory of a size equal to the total amount of physical memory across all nodes.

[0032] As will be described in more detail below, the HyperKernel dynamically optimizes the use of cache memory and virtual processor placement based on its observations of the system as it is running. A "virtual processor" is a computing engine known to its guest operating system, i.e., one that has some operating system context or state. As will be described in more detail below, a "shadow processor" is an anonymous virtual processor, i.e., one that had been a virtual processor but has now given up its operating system context and has context known only to the HyperKernel.

[0033] **RESOURCE VIRTUALIZATION**

[0034] Memory Virtualization

[0035] As explained above, in the physical configuration, each node has an array of memory addresses representing locations in memory. As such, in a physical configuration with three nodes (e.g., as depicted in Figure 6B), there are three memory locations each of which has address 0x123456. In contrast, in the virtual configuration, all memory addresses are unique and represent the sum total of all memory contained in those three nodes. In the virtual configuration, all memory is shared, and all memory caches are coherent. In some embodiments, memory is further subdivided into a series of contiguous blocks, with monotonically increasing memory addresses. In the examples described herein, each page has 4K bytes of memory, however, other subdivisions can also be used, as applicable. The term "blocks" is used herein to describe contiguous arrays of memory locations. In some embodiments, the "blocks" are "pages."

[0036] Processor Virtualization

[0037] A virtual processor (e.g., virtual processor 706 of Figure 7), as seen by the operating system, is implemented on a hyperthread in the physical configuration, but can be location independent. Thus, while the operating system thinks it has 500 processors running on a single physical server, in actuality it might have 5 nodes of 100 processors each. (Or, as is shown in Figure 6B, the operating system will think it has twelve processors running on a single physical server.) The computation running on a virtual processor is described either by the physical configuration on a hyperthread when the computation is running, or in a "continuation," when the virtual processor is not running (i.e., the state of an interrupted or stalled computation).

[0038] As used herein, a "continuation" represents the state of a virtual processor. Each continuation:

[0039] • Has processor state (i.e., saved registers, etc.).

[0040]   • Has a set of performance indicators that guide a scheduler object with information about how to intelligently assign continuations to leaf nodes for execution.

[0041]   • Has a virtual-processor identifier that indicates the processor the operating system thinks is the physical processor to which this continuation is assigned.

[0042]   • Has an event on which this continuation is waiting (possibly empty).

[0043]   • Has a state which includes: "waiting-for-event" or "ready."

[0044]   I/O Virtualization

[0045]   I/O systems observe a similar paradigm to processors and memory. Devices have a physical address in the physical configuration and virtual addresses in the virtual configuration. When migrating computations (described in more detail below), if for example, there are memory buffers associated with I/O operations, the I/O devices used will likely perform better if they are co-located with the memory with which they are associated, and can be moved accordingly.

[0046]   **RESOURCE MAPS**

[0047]   Resource maps are used to translate between virtual and physical configurations. The following are three types of resource maps used by enterprise supercomputers in various embodiments.

[0048]   A "physical resource map" is a table that describes the physical resources that are available on each node. It contains, for example, the number and type of the processors on each node, the devices, the memory available and its range of physical addresses, etc. In some embodiments, this table is read-only and is fixed at boot time.

[0049]   An "initial virtual resource map" is fixed prior to the booting of the operating system and describes the virtual resources that are available from the point of view of the operating system. The configuration is readable by the operating system. In some cases, it may be desirable to configure a system (from the viewpoint of the operating system) that does not match, one-to-one, with the underlying hardware resources. As one example, it may be desirable for the operating system to have more memory and fewer cores. This can be accomplished by changing the ratio of memory to cores, i.e., by modifying the initial virtual resource map.

[0050]   A "current resource map" is created and maintained by each HyperKernel instance. This map describes the current mapping between the virtual resource map and the physical resource map from the point of view of each node. For each entry in the virtual resource map, a definition of the physical resources currently assigned to the virtual resources is maintained. Initially (e.g., at boot time), the current resource map is a copy of the initial virtual resource map. The HyperKernel modifies the current resource map over time as it observes the characteristics of the resource load and dynamically changes the mapping of physical resources to virtual resources (and vice-versa). For example, the definition of the location of the Ethernet controller eth27 in the virtualized machine may at different times refer to different hardware controllers. The current resource map is used by the HyperKernel to dynamically modify the virtual hardware resource mappings, such as the virtual memory subsystem, as required.

[0051]   **RESOURCE MIGRATION OVERVIEW**

[0052]   Using the techniques described herein, virtualized resources can be migrated between physical locations. As explained above, the operating system is provided with information about the virtualized system, but that information need not agree with the physical system.

[0053]   In the following example, suppose an enterprise supercomputer holds a large in-memory database, larger than can fit into a single node. Part of the database is in a first node, "node1." Suppose one of the cores on a different node, "node2," is trying to access data that is owned by node1 and that does not reside locally in a cache on node2. The core on node2 will receive a memory access violation because it is trying to access data that it believes it should be able to access (but cannot). As will be described in more detail below, the exception is handled in the HyperKernel.

[0054]   One way that the situation can be resolved is by moving the needed area of memory to node2, and then returning control back to the operating system (which, in turn, returns it back to the database system). The software can then proceed as intended (i.e., as if the access violation never occurred).

[0055]   In many cases, there may be one or more other cores in other nodes (e.g., "node3") that are also trying to access the same area block of memory as needed by node2 above. Node3 might be attempting to access the same data, or it might be accessing different data contained in the memory that was moved (also referred to as "false sharing"). The data could be moved to node3, but if the core on node2 asks for the data a second time, the data would need to be moved back to

node2 (i.e., potentially moving the data back and forth repeatedly), which can be slow and wasteful. One way to avoid moving data back and forth between cores is to recognize that both cores and the associated block of data should be co-located. Using the techniques described herein, the memory and the computation can be migrated so that they reside on the same node. Doing so will result in a higher likelihood of faster access to data, and a higher probability of sharing data stored in local caches.

[0056]   When the access violation occurs, an event is triggered (in a system dependent way) to which the HyperKernel responds. One example of how such an event can be handled is by the invocation of a panic routine. Other approaches can also be used, as applicable. As will be described in more detail below, the HyperKernel examines the cause of the event and determines an appropriate strategy (e.g., low level transaction) for handling the event. As explained above, one way to handle the event is for one or more blocks of HyperKernel virtualized memory to be transferred from one node's memory to another node's memory. The transfer would then be initiated and the corresponding resource maps would be updated. A continuation would be built poised to be placed in a local table in shared memory called the event table (discussed below) so that the next thing the continuation does when it is resumed would be to return control to the operating system after the transfer is completed. A decision could also be made to move the virtual processor to the node that contains the memory being requested or to move the virtualized memory (and its virtualized memory address) from one node to another. In various embodiments, the HyperKernel makes three decisions when handling an event: which (virtual) resources should move, when to move them, and to where (in terms of physical locations) they should move.

[0057]   **TIDALTREE**

[0058]   The physical hierarchical structure depicted in Figure 2 has an analogous software hierarchy comprising a set of "scheduler objects" (i.e., data structures), each of which has a set of characteristics described below. The scheduler objects form a "TidalTree," which is an in-memory tree data structure in which each node of the tree is a scheduler object. Each scheduler object corresponds to an element of the physical structure of the supercomputer (but not necessarily vice versa), so there is one node for the entire machine (e.g., node 100 as shown in Figure 2), one node for each physical node of the system (e.g., node 102 as shown in Figure 2), one node for each multicore socket on the physical nodes that comprise the entire machine (e.g., node 202 as shown in Figure 2), one node for each core of each socket (e.g., node 210 as shown in Figure 2), and one node for each hyperthread on that core (e.g., node 232 as shown in Figure 2).

[0059]   Each scheduler object s:

[0060]   • Is associated with a physical component (e.g., rack, blade, socket, core, hyperthread).

[0061]   • Except for the root of the tree, has a parent scheduler object which is partly responsible for directing its operations (as explained in more detail below).

[0062]   • Has a set of children each of which is a scheduler object. This is the null set for a leaf (e.g., hyperthread) node. As explained in more detail below, it is the responsibility of a scheduler object s to manage and assign (or re-assign) work to its children, and indirectly to its grandchildren, etc. (i.e., s manages all nodes in the subtree rooted at s).

[0063]   • Has a work queue, which is a set of continuations (as described earlier).

[0064]   • Has a (possibly empty) set of I/O devices that it also has the responsibility to manage and assign (or re-assign) work.

[0065]   Each node can potentially be associated with a layer of some form of cache memory. Cache hierarchy follows the hierarchy of the tree in the sense that the higher the scheduler object is, the slower it will usually be for computations to efficiently utilize caches at the corresponding level of hierarchy. The cache of a scheduler object corresponding to a physical node can be a cache of memory corresponding to that node. The memory on the physical node can be thought of as a cache of the memory of the virtual machine.

[0066]   **RESOURCE MIGRATION – ADDITIONAL INFORMATION**

[0067]   The HyperKernel simulates part of the virtual hardware on which the virtual configuration resides. It is an event-driven architecture, fielding not only translated physical hardware events, but soft events, such as receipt of inter-node HyperKernel messages generated by HyperKernel code running on other nodes.

[0068]   As explained above, when an interrupt event significant to the HyperKernel occurs, the HyperKernel makes a decision of how to respond to the interrupt. Before control is returned to the operating system, any higher priority interrupts are recognized and appropriate actions are taken. Also as explained above, the HyperKernel can make three separate decisions: (1) which resources to migrate upon certain events, (2) when to migrate them, and (3) to where those resources should move.

[0069]   In the following example, suppose a scheduler object "s" in a virtual machine is in steady state. Each scheduler object corresponding to a physical node has a set of physical processor sockets assigned to it. Hyperthreads in these sockets may or may not be busy. The physical node also has some fixed amount of main memory and a set of I/O devices, including some network devices. Scheduler object s, when corresponding to a node, is also responsible for managing the networks and other I/O devices assigned to nodes in the subtree rooted at s. The following is a description of how resources can migrate upon either synchronous or asynchronous events.

[0070]   Migrations Triggered by Synchronous Events

[0071]   In the following example, suppose there exists a leaf node scheduler object s, and virtual processor p assigned to s. Leaf node schedule object s is assumed to be executing an application or operating system code on behalf of an application. Assuming the leaf node is not in an infinite loop, p will eventually run out of work to do (i.e., stall) for some reason (e.g., waiting for completion of an I/O operation, page fault, etc.). Instead of allowing p to actually stall, the HyperKernel decides whether to move the information about the stalled computation to some other node, making one of that other node's processors "responsible" for the stalled continuation, or to keep the "responsibility" of the stalled computation on the node and instead move the relevant resources to the current node.

[0072]   The stall is thus handled in either of two ways: either the computation is moved to the physical node that currently has the resource, or else the resource is moved to the physical node that has requested the resource. Example pseudo code for the handling of a stall is provided below (as the "OnStall" routine) in the "EXAMPLE ROUTINES" section below.

[0073]   Decisions such as how to handle a stall can be dependent on many things, such as the order of arrival of events, the state of the computation running on the virtual machine, the state of the caches, the load on the system or node, and many other things. Decisions are made dynamically, i.e., based on the best information available at any given point in time.

[0074]   Recording Stalled Computations

[0075]   Stalled computations are recorded in a data structure referred to as a "continuation." A continuation has a status that can be, for example, "waiting-for-event" or "ready." A stalled computation gets recorded as a newly created continuation with status "waiting-for-event." Once the reason for stalling is satisfied (e.g., due to detection of the event), the status of the

corresponding continuation is changed to "ready." Each continuation with status "ready" is stored in a "wait queue" of a scheduler object so that eventually it gets scheduled for execution. In contrast, any continuation with status "waiting-for-event" will not be stored in any scheduler object's wait queue. Instead, it is stored in the local shared memory of the physical node where the hardware event that stalled the corresponding computation is expected to occur, such as receipt of a missing resource.

[0076]   Additionally, the newly created continuation is associated with the stalling event that caused its creation. This mapping between (stalling) events and continuations awaiting these events permits fast dispatch of asynchronous events (see the "handleEvent" described below). The mapping between continuations and events is stored in a table called "event table" and is kept in the shared memory of the corresponding physical node. Each physical node has its own event table, and an event table of a physical node is directly addressable by every core on that physical node. All anticipated events recorded in an event table of a physical node correspond to hardware events that can occur on that physical node. The scheduler object s mapped to a physical node n represents n, and the event table of n is associated with s. In some cases, several continuations may be waiting on the same event, and so some disambiguation may be required when the event is triggered.

[0077]   Continuations are built using the "InitContinuation" routine. If a decision is made to move the computation, the remote physical node holding the resource will build a continuation that corresponds to the stalled computation and will store it in the remote physical node's event table. When that continuation resumes, the resource will be available. In effect, the HyperKernel has transferred the virtual processor to a different node.

[0078]   In the case where a decision is made to move the resource, the node that has experienced the stall requests the transfer of the resource and builds a continuation using InitContinuation and stores it in the local event table. Upon receipt of the resource, the continuation is attached to an appropriate node in the TidalTree, and when that continuation is resumed, the resource will be generally be available and visible. In effect, the virtual resource has been transferred to the node that requested it.

[0079]   Note that by placing continuations in event tables, it is guaranteed that the processor that receives the event will quickly find the related continuations in its local event table. The reason for the stall in the computation will have been satisfied.

[0080] Having dealt with the stall, the virtual-processor p will in effect be suspended. In between processing the stall and finding a new continuation to resume, p becomes an "anonymous shadow processor," i.e., a processor with no identity known to the operating system. This shadow processor then looks for a new continuation to resume. An example of this is shown below in the "assignProcessor" routine described in more detail below.

[0081] Notation

[0082] Let e be the event that stalled virtual processor p. Assume that e is triggered by local hardware of some physical node n. In particular, assume r is the resource, which caused the stalling event to occur. Resource r could be a block of memory, or an I/O operation, or a network operation. Assume that p is assigned to scheduler object s, which belongs to the subtree rooted at the scheduler object that represents physical node n.

[0083] On-Stall

[0084] Pseudo code for an example on-stall routine is provided below in the "EXAMPLE ROUTINES" section. The migration-continuation function returns true if and only if processor p in node n decides that the resource should not move, i.e., the computation should move. This can be determined by a number of factors such as history and frequency of movement of r between nodes, the type of r, the cost of movement, the number of events in n's local event table waiting for r, system load, etc. For example, it may not be desirable to move a resource if there is a continuation stored in n's local event table that is waiting for it.

[0085] A variety of patterns of events that would benefit from migrations exist. One approach to describing these patterns of events, like access violations, is in formal language theory. Regular (i.e., Chomsky type-3) languages can be recognized by finite state automata. In addition, using a compact and flexible notation, a description of the events that are observed can be made as sentences (or Chomsky sequences) in the regular language, and the recognition modeled as state transitions in the corresponding finite state automaton. When the full Chomsky sequence of events is seen, migration-continuation gets evaluated accordingly: if the finite state automaton accepts the Chomsky sequence, the condition is met, otherwise, it is not met. The length of the minimized finite state machine defines the amount of history that needs to be kept.

[0086] In various embodiments, all events happen locally, and the HyperKernel on the physical node receiving the event must handle it –truly synchronous events are not assumed to occur between physical nodes. To coordinate migration strategy between nodes, "messages" are

used. Message "sends" are synchronous from a node's point of view, but message "receives" are asynchronous, in that a processor or shadow processor, in general, does not wait for receipt of a message. When messages arrive, they are dealt with by the HyperKernel as a virtual interrupt. In one embodiment, the HyperKernel will not allow a processor to resume a continuation while there are messages waiting to be handled. Therefore, before control is transferred back to the operating system, the queue is checked, and any messages are dealt with prior to the transfer of control back to the operating system.

[0087]   For scheduler object s and continuation c, a cost function cost(s,c) can be used to guide the search up the tree. If multiple ancestors of p have non-empty queues, then p may not want to stop its search at the first ancestor found with a nonempty wait queue. Depending on the metrics used in the optimizing strategy, p's choice may not only depend on the distance between p and its chosen ancestor but on other parameters such as length of the wait queues.

[0088]   A function, find-best-within(s), can be used to return the "best-fit" continuation in a (non-empty) wait queue of a scheduler object. Examples of parameters that can be considered include:

[0089]   1. Position in the queue

[0090]   2. The relationship between p and the last location recorded in the continuation (the closer those locations are the better it may be for reusing cache entries).

[0091]   3. Performance indicators recorded in the continuations in the queue.

[0092]   The cost and find-best-within functions can be customized as applicable within a given system.

[0093]   Migrations Triggered by Asynchronous Events

[0094]   Examples of asynchronous events include: receipt of a packet, completion of an I/O transfer, receipt of a resource, receipt of a message requesting a resource, etc. Generally, a HyperKernel that receives an event corresponding to a hardware device managed by the operating system needs to deliver a continuation associated with that event to a scheduler object s. By doing so, s will make this continuation available to an appropriate scheduler object and then ultimately to the computation managed by the operating system represented by that continuation. If, on the other hand, the event is the receipt of a message from a HyperKernel on another physical node, the HyperKernel can handle it directly.

[0095] To simplify explanation, in the examples described herein, an assumption is made that there is only one continuation associated with an event. The procedures described herein can be generalized for the case where multiple continuations are associated with the same event, as needed.

[0096] In some embodiments, the search for a scheduler object on which to place the continuation starts at the leaf of the tree that built the continuation and then proceeds upward (if the computation previously executed on this node). By doing so, the likelihood of reusing cache entries is increased.

[0097]   Handle-Event

[0098] Pseudo code for an example handle-event routine is provided below in the "EXAMPLE ROUTINES" section. The cost function, cost(s,c), is a function that helps determine the suitability of assigning c to scheduling object s. The cost function can depend on a variety of parameters such as the size of the wait queues, the node traversal distance between s and the original scheduling node for c (to increase the probability that cache entries will be reused), and the history of the virtual processor, the physical-processor, and the continuation. If the wait queues of the scheduler objects close to s already contain too many continuations, then it may take a relatively longer time until any newly added continuation is scheduled for execution. Example conditions contributing to cost(s,c) are described below, and the conditions can be customized as applicable.

[0099]   **COSTS**

[00100] Cost functions are used to evaluate options when selecting continuations and scheduling objects. Cost functions can be expressed as the summation of a sum of weighted factors:

[00101] $cost = w_1 f_1^{x_1} + w_2 f_2^{x_2} + \ldots + w_n f_n^{x_n},$

[00102] where w_i indicates the importance of the corresponding factor and x_i indicates an exponential.

[00103] Examples of factors f_i are listed for each of the costs below. Weights w_i and exponents x_i can be determined in a variety of ways, such as empirically and by simulation. Initial weights and exponents can be tuned to various application needs, and can be adjusted by an

administrator to increase performance. The weights can be adjusted while the system is active, and changing weights does not change the semantics of the HyperKernel, only the operational performance characteristics.

[00104]   Examples of the factors that can be considered include:

[00105]   • Length of time since the last processor evacuated this scheduler object.

[00106]   • Height of the scheduler object in the TidalTree.

[00107]   • Length of the work queue.

[00108]   • Reservation status (i.e., it may be the case that some application has reserved this resource for a specific reason).

[00109]   • Node specification (i.e., the node itself might have been taken out of service, or is problematic, has in some way a specialized function, etc.).

[00110]   • Age of the continuation in the queue.

[00111]   • Last physical processor to run this continuation.

[00112]   • Last virtual processor to run this continuation.

[00113]   • Node on which this continuation was last executing.

[00114]   • The "temperature" of the cache. (A cache is "warm" when it has entries that are likely to be reused. A cache is "cold" when it is unlikely to have reusable cache entries.)

[00115]   • Group membership of the continuation (i.e., the continuation may be part of a computation group, each element of which has some affinity for other members of the group).

[00116]   • Performance Indicators (Hints) and special requirements.

[00117]   **EXAMPLES**

[00118]   "OnStall" and "assignProcessor"

[00119]   Figure 8 illustrates an embodiment of a process for selectively migrating resources. In some embodiments, process 800 is performed by a HyperKernel, such as in conjunction with the OnStall routine. The process begins at 802 when an indication is received that a core (or

hyperthread included in a core, depending on whether the processor chip supports hyperthreads) is blocked. As one example, suppose a hyperthread receives a request, directly or indirectly, for a resource that the hyperthread is not able to access (e.g., RAM that is located on a different node than the node which holds the hyperthread). When the hyperthread fails to access the resource (i.e., an access violation occurs), an interrupt occurs, which is intercepted, caught, or otherwise received by the HyperKernel at 802. In particular, the HyperKernel receives an indication at 802 that the hyperthread is blocked (because it cannot access a resource that it has been instructed to provide). In addition to reporting its blocked state, the hyperthread provides information such as the memory address it was instructed to access and what type of access was attempted (e.g., read, write, or modify).

[00120]   At 804, the HyperKernel determines whether the needed memory should be moved (e.g., to the node on which the blocked hyperthread is located), or whether the requesting process should be remapped (i.e., the virtual processor should be transferred to a different node). The decision can be based on a variety of factors, such as where the needed memory is located, the temperature of the cache, the workload on the node holding the hyperthread, and the workload on the node holding the needed memory (e.g., overworked or underworked). In some embodiments, the workload of a node is determined based at least in part on the average queue length in the TidalTree.

[00121]   If the HyperKernel determines that the memory should be moved, the HyperKernel uses its current resource map to determine which node is likely to hold the needed memory and sends a message to that node, requesting the resource. The HyperKernel also creates a continuation and places it in its event table. The hyperthread that was blocked at 802 is thus freed to take on other work, and can be assigned to another virtual processor using the assignProcessor routine.

[00122]   The HyperKernel checks its message queue on a high-priority basis. When the HyperKernel receives a message from the node it contacted (i.e., the "first contacted node"), in some embodiments, one of two responses will be received. The response might indicate that the first contacted node has the needed resource (and provide the resource). Alternatively, the message might indicate that the contacted node no longer has the resource (e.g., because the node provided the resource to a different node). In the latter situation, the first contacted node will provide the identity of the node to which it sent the resource (i.e., the "second node"), and the HyperKernel can send a second message requesting the resource – this time to the second node. In various embodiments, if the second node reports to the HyperKernel that it too no longer has the resource (e.g., has provided it to a third node), the HyperKernel may opt to send the continuation to the third

node, rather than continuing to request the resource. Other thresholds can be used in determining whether to send the continuation or continuing the resource (e.g., four attempts). Further, a variety of criteria can be used in determining whether to request the resource or send the continuation (e.g., in accordance with a cost function).

[00123]   In the event the HyperKernel determines that the continuation should be transferred (i.e., that the computation should be sent to another node rather than receiving the resource locally), the HyperKernel provides the remote node (i.e., the one with the needed resource) with information that the remote node can use to build a continuation in its own physical address space. If the remote node (i.e., the one receiving the continuation) has all of the resources it needs (i.e., is in possession of the resource that caused the initial access violation), the continuation need not be placed into the remote node's event table, but can instead be placed in its TidalTree. If the remote node needs additional resources to handle the continuation, the received continuation is placed in the remote node's event table.

[00124]   Figure 9 illustrates an embodiment of a process for performing hierarchical dynamic scheduling. In some embodiments, process 900 is performed by a HyperKernel, such as in conjunction with the assignProcessor routine. The process begins at 902 when an indication is received that a hyperthread should be assigned. Process 900 can be invoked in multiple ways. As one example, process 900 can be invoked when a hyperthread is available (i.e., has no current work to do). This can occur, for example, when the HyperKernel determines (e.g., at 804) that a continuation should be made. The previously blocked hyperthread will become available because it is no longer responsible for handling the computation on which it blocked (i.e., the hyperthread becomes an "anonymous shadow processor"). As a second example, process 900 can be invoked when a message is received (e.g., by the HyperKernel) that a previously unavailable resource is now available. The HyperKernel will need to locate a hyperthread to resume the computation that needed the resource. Note that the hyperthread that was originally blocked by the lack of a resource need not be the one that resumes the computation once the resource is received.

[00125]   At 904, the TidalTree is searched for continuations that are ready to run, and one is selected for the hyperthread to resume. In various embodiments, the TidalTree is searched from the leaf-level, upward, and a cost function is used to determine which continuation to assign to the hyperthread. As one example, when a hyperthread becomes available, the continuation that has been queued for the longest amount of time could be assigned. If no continuations are waiting at the leaf level, or are outside a threshold specified by a cost function, a search will be performed up the TidalTree (e.g., the core level, then the socket level, and then the node level) for an appropriate

continuation to assign to the hyperthread. If no appropriate continuations are found for the hyperthread to resume at the node level, the HyperKernel for that node contacts the root. One typical reason for no continuations to be found at the node level is that there is not enough work for that node to be fully utilized. In some embodiments, the node or a subset of the node can enter an energy conserving state.

[00126]   Time Sequence

[00127]   For expository purposes, in the example, a "swapping" operation is used to transfer continuations and memory, but in fact that's not necessary in all embodiments.

[00128]   Figure 10 illustrates an example of an initial memory assignment and processor assignment. Specifically, region 1002 of Figure 10 depicts a HyperKernel's mapping between physical blocks of memory (on the left hand side) and the current owner of the memory (the center column). The right column shows the previous owner of the memory. As this is the initial memory assignment, the current and last owner columns hold the same values. Region 1004 of Figure 10 depicts a HyperKernel's mapping between system virtual processors (on the left hand side) and the physical nodes (center column) / core numbers (right column).

[00129]   Suppose virtual processor P00 makes a memory request to read location 8FFFF and that the HyperKernel decides to move one or more memory blocks containing 8FFFF to the same node as P00 (i.e., node 0). Block 8FFFF is located on node 2. Accordingly, the blocks containing 8FFFF are transferred to node 0, and another block is swapped out (if evacuation is required and the block is valid), as shown in Figure 11.

[00130]   Next, suppose virtual processor P06 makes a memory request to read location 81FFF. The contents of this block have been moved (as shown in Figure 11) to node 0. The HyperKernel may determine that, rather than moving the memory again, the computation should be moved. Accordingly, virtual processor P06 is moved to node 0, and may be swapped with virtual processor P01, as shown in Figure 12.

[00131]   **PERFORMANCE INFORMATION**

[00132]   Locks and Other Synchronizers

[00133]   In various embodiments, the use of synchronization mechanisms like locks is minimal. Locks are used, for example, to insert queue and remove queue continuations on scheduler objects and to maintain the event table.

[00134]   Code Path Lengths

[00135]   In some embodiments, the (maximum) length of all code paths is determined through a static code analysis, resulting in estimable and bounded amounts of time spent in the HyperKernel itself. All data structures can be pre-allocated, for example, as indexed arrays. The nodes of the TidalTree are determined at boot time and are invariant, as are the number of steps in their traversal. One variable length computation has to do with the length of the work queues, but even that can be bounded, and a worst-case estimate computed. In other embodiments, other variable length computations are used.

[00136]   Static Storage

[00137]   In various embodiments, all data structures needed in the HyperKernel are static, and determined at boot time, so there is no need for dynamic memory allocation or garbage collection.

[00138]   Physical Memory

[00139]   All memory used by the HyperKernel is physical memory, so no page tables or virtual memory is required for its internal operations (except, e.g., to manage the virtual resources it is managing), further helping the HyperKernel to co-exist with an operating system.

[00140]   **SHARING DATA AND MAINTAINING CONSISTENCY**

[00141]   In some cases, e.g., to preserve the conceptual integrity of the virtual machine being presented to the operating system, changes in one node's data structures are coordinated with corresponding ones in a different node. Many of the data structures described herein are "node local," and either will not need to move, or are constant and replicated. The data structures that are node local are visible to and addressable by all hyperthreads on the node. Examples of data structures that are not node local (and thus require coordination) include the current resource map (or portions thereof), the root of the TidalTree, and migratory continuations (i.e., continuations that might have to logically move from one node to another).

[00142]   A variety of techniques can be used to maintain a sufficient degree of consistency. Some are synchronous and assume all changes are visible at the same time to all nodes (i.e., "immediate consistency"). Others allow a more relaxed solution and strive for "eventual consistency." As mentioned above, physical nodes of an enterprise supercomputer are connected

via one or more high speed interconnects. Multiple instances of HyperKernels are interconnected to pass messages and resources back and forth between physical nodes.

[00143]   Updating the Current Resource Map

[00144]   Each physical node n starts off (e.g., at boot time) with the same copy of the physical resource map, the initial virtual resource map, and the current resource map. Each node maintains its own copy of the current resource map.

[00145]   In some embodiments, each entry for resource r in the current resource map has the following:

[00146]   1. A local lock, so that multiple hyperthreads on a physical-node cannot modify r at the same time.

[00147]   2. A node number specifying the node that currently owns the resource.

[00148]   3. A count k of the number of times n has requested r since the last time it owned r.

[00149]   4. A boolean which when set signifies that this node n wants r.

[00150]   5. A boolean which when set signifies that this node has r but is in the process of transferring it, in which case the node number specifies the new owner.

[00151]   In some embodiments, the count k is used to deal with unbounded chasing of resources. If k exceeds a threshold, a determination is made that it is better to move the newly built continuation rather than chasing the resource around the system.

[00152]   The following is an example of a mechanism for initiating migration of resources and receiving resources. Key transactions include the following:

[00153]   1. Node n sends a request for resource r to n'.

[00154]   2. Node n' receives a request for resource r from n.

[00155]   3. Node n' may send a "deny" message to n under certain circumstances, otherwise it can "accept" and will send the resource r.

[00156]   4. Node n will receive a "deny" message from n' if the resource r cannot be sent by n' at this point in time. It may be that r is needed by n', or it may be that r is being transferred

somewhere else at the arrival of the request. If the request is denied, it can send a "forwarding" address of the node to which it's transferring the resource. It may be that the forwarding address is n' itself, which is the equivalent of "try again later." When node n receives the deny message, it can resend the request to the node suggested by n', often the new owner of the resource. To avoid n chasing the resource around the system, it can keep track of the number of attempts to get the resource, and switches strategy if the number of attempts exceeds a threshold.

[00157]   5. Node n will receive the resource r if n' can send the resource. In this case, n needs to schedule the continuation c that was awaiting r, so that c can be resumed.

[00158]   TidalTree Root

[00159]   In some embodiments, one physical node of the set of nodes in the system is designated as a "master node." This node has the responsibility at boot time for building the initial virtual resource map and other data structures, replicating them to the other nodes, and booting the operating system (e.g., Linux). The master node can be just like any other node after the system is booted up, with one exception. At least one physical node needs to store the root of the TidalTree. The master node is one example of a place where the root can be placed. Updates to the event queue of the TidalTree root scheduling object are handled in each node by sending a message to the master node to perform the update.

[00160]   Over time, the HyperKernel will adapt and locality will continually improve if resource access patterns of the operating system and the application permit.

[00161]   Continuations

[00162]   As explained above, physical memory addresses across all nodes are not unique. In some embodiments, the inclusion of physical memory addresses in continuations can be avoided by using partitioned integer indices to designate important data structures in the HyperKernel. In the event an addresses needs to be put into a continuation, care is taken in the move, since the address is a physical address of the source, and bears no relationship with the physical address in the destination. Moving a continuation means copying its contents to the destination node as discussed above, and remapping any physical addresses from the source to the target.

[00163]   Timestamps

[00164]   In some embodiments, access to a free-running counter is visible to all of the nodes. In the absence of this, free-running counters on each node can also be used. Counters in continuations are mapped between the source and destination.

[00165]   Handling of Disks and Persistent Flash

[00166]   Where a needed resource is on disk (or persistent flash), in some embodiments, such resources are treated as having a heavier gravitational field than a resource such as RAM. Accordingly, disk/flash resources will tend to not migrate very often. Instead, continuations will more frequently migrate to the physical nodes containing the required persistent storage, or to buffers associated with persistent storage, on a demand basis.

[00167]   Operating System Configuration

[00168]   There are many ways to configure an operating system. For servers, an assumption can be made that its operating system is configured to only require a small set of resource types from the virtual machine implemented by the HyperKernel: storage that includes linear block arrays, networks, processors, memory, and internode interconnects. As a result, the complexity of the operating system installation can be reduced.

[00169]   **EXAMPLE DATA STRUCTURES AND FUNCTIONS**

[00170]   The following section provides a list of examples of data structures and functions used in various embodiments.

[00171]   init-continuation: Initializes a continuation when a computation is stalled.

[00172]   assignProcessor: Routine that assigns a new continuation to a shadow processor (if possible).

[00173]   on-stall(r): Stalling event occurs for resource r.

[00174]   migrate-computation(computational-state,r,n): Message to request migration of a computational state to another node n which you hope has resource r.

[00175]   on-interrupt(i): Software interrupt i occurs.

[00176]   handle-event(e): Routine executed when the HyperKernel is called on to handle an asynchronous event.

[00177]   request-resource(r,n): Request transfer of resource r from node n.

[00178]   initiate-send-resource(r,n): Start sending resource r to node n.

[00179]   on-request-transfer-response(r,n,b): The requested transfer of r from n was accepted or rejected. b is true if rejected.

[00180]   on-transfer-requested (r,m): Receive a request from m for resource r.

[00181]   on-resource-transferred(r,n): Ack of resource r has been received from n.

[00182]   on-receive-resource (r,n): Resource r has been received from n.

[00183]   migration-continuation(r): True if and only if it is better to migrate a continuation than move a resource.

[00184]   parent(s): Returns the parent scheduler-object of scheduler object s.

[00185]   cost(s,c): Used to evaluate placement of continuation c in the wait-queue of scheduler-object s.

[00186]   find-best-within(s): A cost function that returns a continuation stored in the wait-queue of scheduler-object s.

[00187]   conserve-energy: Enter low power mode.

[00188]   resume-continuation(c): Resume the computation represented by c in the processor executing this function at the point.

[00189]   valid(i): Boolean function that returns true if and only if interrupt i is still valid.

[00190]   initialize(best-guess): Initializes cost variable best-guess.

[00191]   insert-queue(s,c): Insert continuation c into the wait-queue of scheduler-object s.

[00192]   return-from-virtual-interrupt: Resume execution that was temporarily paused due to the interrupt.

[00193]   r.owner: Returns the node where resource r is local.

[00194]   r.e: Resource r is awaiting this event.

[00195]   e.r: This event is for resource r.

[00196]   e.continuation: When this event occurs, need to resume continuation.

[00197]   get-state(): Returns processor's state.

[00198]   scheduler-object(p): Returns scheduler-object currently associated with processor p.

[00199]   on–request-transfer-response(r,m, response): Response to request of transferring
resource r from node m. Response can be either true if "rejected" or false if "accepted."

[00200]   **EXAMPLE ROUTINES**

[00201]   The following are pseudo-code examples of routines used in various embodiments.
In the following, functions that start with "on-" are asynchronous events or messages coming in.

[00202]   ===========================

[00203]   init-continuation(computational-state)

[00204]   ===========================

[00205]   /* InitContinuation by processor p awaiting resource r with hints h */

[00206]   c = allocate continuation

[00207]   c.state = computational-state

[00208]   c.last = scheduler-object(p)

[00209]   c.state = waiting-for-event

[00210]   c.hints = h

[00211]   e = allocate event in event-table

[00212]   e.resource = r

[00213]   e.continuation = c

[00214]   return c

[00215]   end InitContinuation

[00216]   ==========================

[00217]   assignProcessor

[00218]   ==========================

[00219]   /* Once processor p in physical node n becomes a shadow processor, it gives up its O/S identity and starts looking for a continuation with which to resume execution. p will look for such a continuation in wait-queues as follows: */

[00220]   s = scheduler-object (p)

[00221]   initialize (best-guess)

[00222]   best-s = nil

[00223]   /* traverse upwards, keeping track of best candidate */

[00224]   /* assume there is a locally cached copy of the root */

[00225]   repeat

[00226]   guess = cost (s)

[00227]   if guess > best-guess

[00228]   then

[00229]   best-guess = guess

[00230]   best-s = s

[00231]   s = parent (s)

[00232]   until s = nil

[00233]   if best-s <> nil

[00234]   then

[00235]   c = find-best-within (best-s)

[00236]   resume-continuation (c)

[00237]   else conserve-energy

[00238]   end assignProcessor

[00239]   ===========================

[00240]   on-stall(r)

[00241]   ===========================

[00242]   /* OnStall is invoked when the hardware detects an inconsistency between the virtual and physical configurations. More specifically, node n requests resource r which the hardware cannot find on node n. */

[00243]   if migration-continuation (r)

[00244]   then

[00245]   /* send the computation to node n */

[00246]   nn = owner(r)

[00247]   /* node n believes resource is probably at node nn */

[00248]   migrate-computation (r,nn)

[00249]   else

[00250]   /* request the resource r */

[00251]   c = init-continuation(get-state())

[00252]   /* insert code here to insert c into the local event-table */

[00253]   request-resource(r, owner(r))

[00254]   assignProcessor   /* At this point, p is an anonymous shadow processor */

[00255]   /* p needs to find some work to do */

[00256]   end OnStall

[00257]   ===========================

[00258]   on-migrate-computation(computational-state, r,n)

[00259]   ===========================

[00260]   /* the remote node gets the message from n to receive a continuation. Note: c in this case is the contents of the continuation, not the continuation itself. */

[00261]   c = InitContinuation /* with the information in the request */

[00262]   c.state = computational-state

[00263]   e = insert c into the local event-table

[00264]   handle-event (e)

[00265]   end on-migrate-computation

[00266]   ===========================

[00267]   on-interrupt(i)

[00268]   ===========================

[00269]   /*When a processor p (in subtree of physical node n) is interrupted by i (using a very low level mechanism specific to the particular hardware design), p does the following: */

[00270]   while valid (i)

[00271]   e = event-table (i)   /* find the event corresponding to i */

[00272]   handle-event (e)

[00273]   i = next-queued-interrupt

[00274]   end while

[00275]   /* resume prior execution */

[00276]   return-from-virtual-interrupt

[00277]   end on-interrupt

[00278]   ===========================

[00279] handle-event(e)

[00280] ==========================

[00281] /* An event occurred. Move it from the event table to the best scheduler-object. */

[00282] c = e.continuation   /* find the continuation for event e */

[00283] event-table (i).clear = true   /* remove the event from the table */

[00284] e.complete = true   /* mark e as completed */

[00285] c.state = ready

[00286] /* now find out the best place to put c */

[00287] s = c.last

[00288] initialize (best-guess)

[00289] /* look for best choice */

[00290] /* assume there is a locally cached copy of the root */

[00291] repeat

[00292] guess = cost (s,c)

[00293] if guess > best-guess

[00294] then

[00295] best-guess = guess

[00296] best-s = s

[00297] s = parent (s)

[00298] until s = nil

[00299] insert-queue (best-s,c)/* queue up c in the wait-queue of best-s */

[00300] end handle-event

[00301]   =========================

[00302]   request-resource (r,n)

[00303]   =========================

[00304]   /* When a node n needs a resource r owned by node n' the resource is requested, but the request may not be satisfied because someone else might have beaten you to request it or n' is currently using it. */

[00305]   current-resource-map(r).wanted = true

[00306]   request-transfer(owner(r),r)   /* send a request to the owner of r */

[00307]   /* requesting r's transfer */

[00308]   return

[00309]   =========================

[00310]   on-request-transfer-response (r, m, is-rejected)

[00311]   =========================

[00312]   /* Now, consider that you are a node getting a response from a previous request to a node for a resource r. When the response to this request comes in, it can be accepted or rejected. */

[00313]   if is-rejected

[00314]   then /* resource has been transferred to m */

[00315]   increment k

[00316]   if k > threshold

[00317]   then

[00318]   /* you don't want to go chasing around forever*/

[00319]   /* trying to get the resource. Give up */

[00320]   migrate-computation(r,m)

[00321]   return

[00322]   else

[00323]   request-transfer(r,m)   /* try again */

[00324]   return

[00325]   else

[00326]   /* request was not rejected and r is the resource */

[00327]   r.k = 0

[00328]   r.wanted = false   /* resource has been moved */

[00329]   r.owner = me   /* set the owner to n (i.e., "me") */

[00330]   if the resource is memory,

[00331]   update the hardware memory map with the new memory

[00332]   return

[00333]   =============================

[00334]   on-transfer-requested (r,n)

[00335]   =============================

[00336]   /* When a resource request for r comes from node n, if transfer in progress to owner(r), deny the request */

[00337]   if r.being-transferred

[00338]   then

[00339]   send-request-response (r, owner(r), true)

[00340]   else

[00341]   /* transfer of resource is accepted */

[00342]   r.transferring = true

[00343]   initiate-send-resource(r)

[00344]   if type(r) = memory

[00345]   then update local memory map

[00346]   send-request-response (r, owner(r), false)

[00347]   return

[00348]   =========================

[00349]   on-resource-transferred (r,n)

[00350]   =========================

[00351]   /* When an acknowledgement comes in that the transfer is complete */

[00352]   r.owner = n

[00353]   r.transferring = false

[00354]   return

[00355]   =========================

[00356]   on-receive-resource(r,n)

[00357]   =========================

[00358]   /* Now we receive a message with the requested resource r from n*/

[00359]   r.k = 0

[00360]   r.wanted = false/* clear the bit saying that it's wanted */

[00361]   r.owner = me   /* set the owner to n (i.e., "me") */

[00362]   if the resource is memory,

[00363]   update the memory map with the new memory

[00364]   send-resource-transferred(r,n)

[00365]   handle-event(r.e)   /* the event we've been waiting for has occurred */

[00366]   return

[00367]   Although the foregoing embodiments have been described in some detail for purposes of clarity of understanding, the invention is not limited to the details provided. There are many alternative ways of implementing the invention. The disclosed embodiments are illustrative and not restrictive.

[00368]   WHAT IS CLAIMED IS:

CLAIMS

1.   A computer system, comprising:

a plurality of physical nodes, wherein each node includes a plurality of processors;

wherein each of the processors includes a plurality of hyperthreads;

wherein an abstraction of the nodes, processors, and hyperthreads forms a hierarchy; and

wherein upon receiving an indication that a hyperthread should be assigned, a dynamic search of the hierarchy is performed for a process to assign to the hyperthread, wherein the search begins at the leaf level of the hierarchy.

2.   The system of claim 1 wherein the indication is received in response to the hyperthread becoming available for assignment.

3.   The system of claim 1 wherein the indication is received in response to a previously unavailable resource becoming available.

4.   The system of claim 1 wherein the indication is received in response to a hyperthread becoming blocked.

5.   The system of claim 4 wherein, as a result of performing the search, the blocked hyperthread is designated an anonymous processor.

6.   The system of claim 4 wherein upon receiving notification that the hyperthread is blocked, a continuation is dynamically created and stored in an event table.

7.   The system of claim 6 wherein the event table is stored on the node on which the hyperthread is located.

8.   The system of claim 6 wherein the event table includes a plurality of continuations.

9.   The system of claim 1 wherein performing the dynamic search includes evaluating a cost function.

10.   The system of claim 1 wherein the search is performed by a HyperKernel.

11.   The system of claim 10 wherein the HyperKernel is configured to dynamically assign a process included in an event table to the hyperthread.
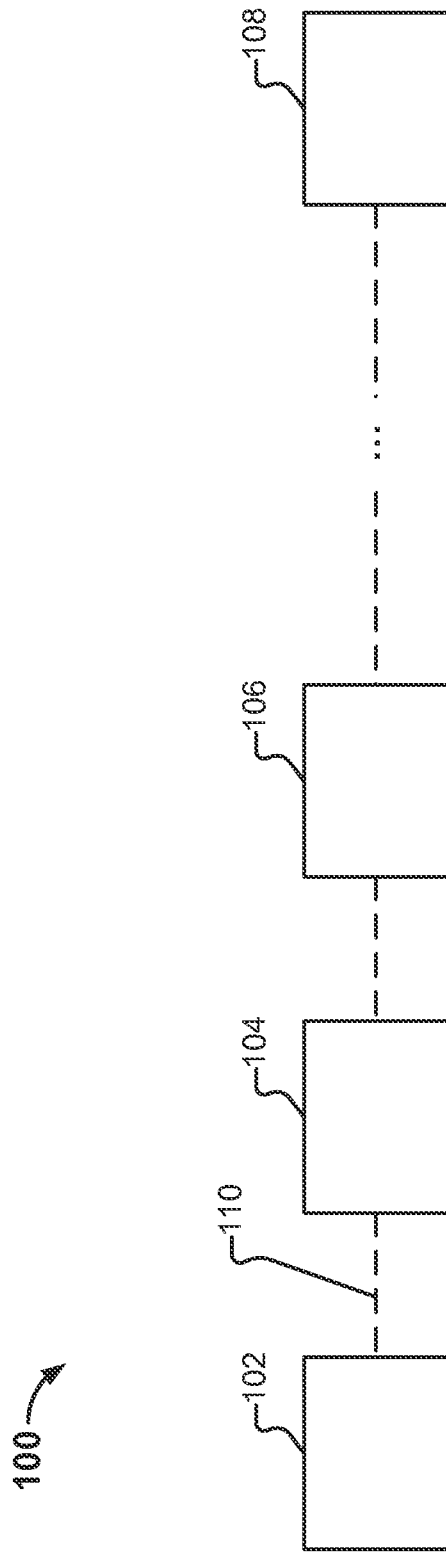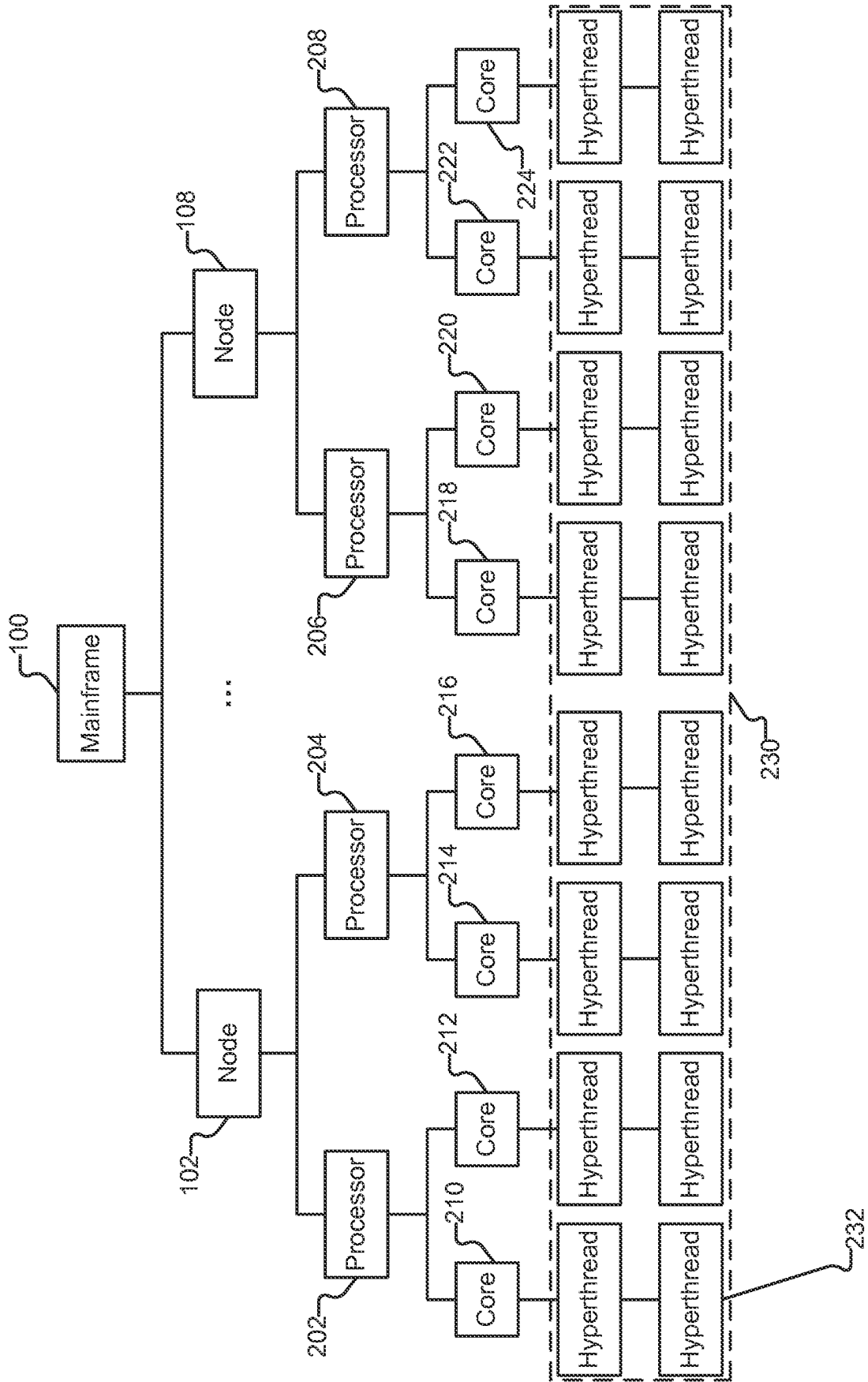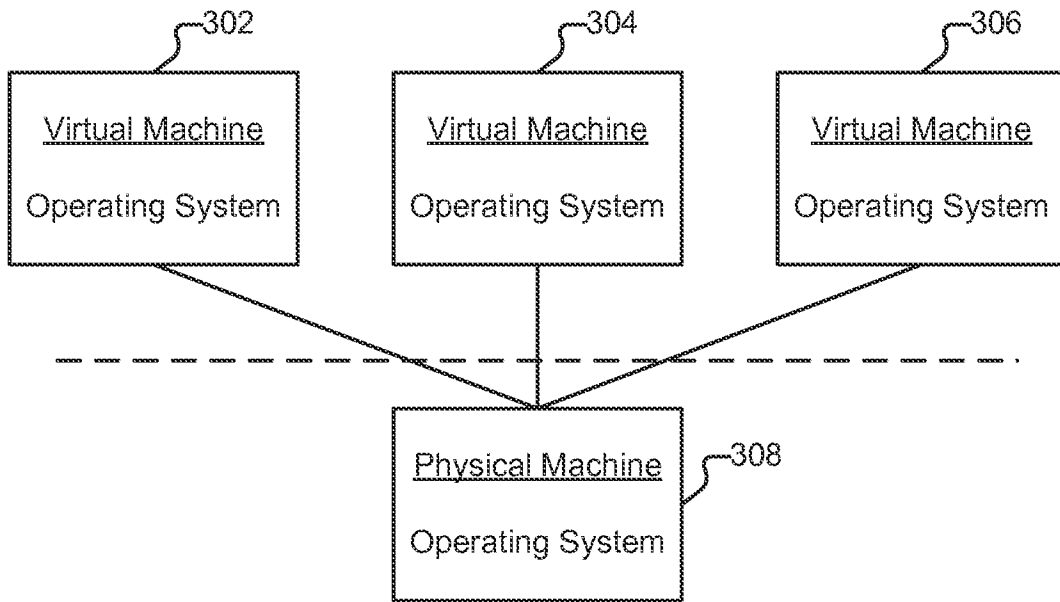
12.   A method, comprising:

receiving an indication that a hyperthread be assigned; and

dynamically searching a hierarchy for a process to assign to the hyperthread, wherein the search begins at a leaf level of the hierarchy; and

wherein the hierarchy comprises an abstraction of a set of physical nodes, processors, and hyperthreads.

13.   The method of claim 12, wherein the indication is received in response to the hyperthread becoming available for assignment.

14.   The method of claim 12, wherein the indication is received in response to a previously unavailable resource becoming available.

15.   The method of claim 12, wherein the indication is received in response to a hyperthread becoming blocked.

16.   The method of claim 15, wherein upon receiving notification that the hyperthread is blocked, a continuation is dynamically created and stored in an event table.

17.   The method of claim 16, wherein the event table includes a plurality of continuations.

18.   The method of claim 12, wherein performing the dynamic search includes evaluating a cost function.

19.   The method of claim 12, wherein the dynamic search is performed by a HyperKernel.

20.   The method of claim 19, wherein the HyperKernel is configured to dynamically assign a process included in an event table to the hyperthread.

FIG. 1

FIG. 2

302

Virtual Machine

Operating System

304

Virtual Machine

Operating System

306

Virtual Machine

Operating System

Physical Machine

Operating System

308

FIG. 3A

Virtual Machine

Operating System

352

Physical Machine

Operating System

354

Physical Machine

Operating System

356

Physical Machine

Operating System

358

FIG. 3B

Application	402
Database	404
Operating System	406
Hardware	408

FIG. 4A

Application	452
Database	454
Operating System	456

HyperKernel	HyperKernel	HyperKernel
Hardware	Hardware	Hardware
458	460	462

FIG. 4B

Operating System ⌇—502

Processor 1 ⌇—504   Processor 2 ⌇—506   Processor 3 ⌇—508

Physical Shared Memory ⌇—510

FIG. 5

600

602 — H1 | Cache 1 | | Cache 2 | | Cache 3 |

604 — H2 | Cache 1 | | Cache 2 | | Cache 3 | Physical   0
                                                          Shared   ⌐612
606 — H3 | Cache 1 | | Cache 2 | | Cache 3 | Memory

608 — H4 | Cache 1 | | Cache 2 | | Cache 3 | MAX

610
(Invisible)

FIG. 6A

652 — Node 1
        H1-1 | Cache 1 | | Cache 2 | | Cache 3 |
        H1-2 | Cache 1 | | Cache 2 | | Cache 3 | Physical   0
        H1-3 | Cache 1 | | Cache 2 | | Cache 3 | Shared
        H1-4 | Cache 1 | | Cache 2 | | Cache 3 | Memory   MAX
                        658

654 — Node 2
        H2-1 | Cache 1 | | Cache 2 | | Cache 3 |
        H2-2 | Cache 1 | | Cache 2 | | Cache 3 | Physical   0
        H2-3 | Cache 1 | | Cache 2 | | Cache 3 | Shared
        H2-4 | Cache 1 | | Cache 2 | | Cache 3 | Memory   MAX

656 — Node 3
        H3-1 | Cache 1 | | Cache 2 | | Cache 3 |
        H3-2 | Cache 1 | | Cache 2 | | Cache 3 | Physical   0
        H3-3 | Cache 1 | | Cache 2 | | Cache 3 | Shared
        H3-4 | Cache 1 | | Cache 2 | | Cache 3 | Memory   MAX

FIG. 6B

706

704

P1

P2

P3

P4

P5

P6

702

P7

P8

P9

P10

...

Pmax

FIG. 7

800 ⌐

Receive indication that a core, associated with a
virtual processor, cannot access a needed portion of
physical memory.   ⌐802

Selectively move contents of the needed memory
closer to the core, or remap the virtual processor to
another core.   ⌐804

FIG. 8

900 ⌐

```
┌─────────────────────────────────────────┐
│                                          │⌐902
│  Receive an indication that a hyperthread should be │
│               assigned.                  │
│                                          │
└─────────────────────────────────────────┘
                      │
                      │
                      ▼
┌─────────────────────────────────────────┐
│                                          │⌐904
│  Search a hierarchy for a process to assign to the │
│               hyperthread.               │
│                                          │
└─────────────────────────────────────────┘
```

FIG. 9

INITIAL (Time 0) MEMORY ASSIGNMENT

BLOCK		Current Owner	Last Owner
0	– FFFF	Node 0	Node 0
10000	– 1FFFF	Node 0	Node 0
20000	– 2FFFF	Node 0	Node 0
30000	– 3FFFF	Node 1	Node 1
40000	– 4FFFF	Node 1	Node 1
50000	– 5FFFF	Node 1	Node 1
60000	– 6FFFF	Node 2	Node 2
70000	– 7FFFF	Node 2	Node 2
80000	– 8FFFF	Node 2	Node 2

• • •

1002

INITIAL (Time 0) PROCESSOR ASSIGNMENT

O/S Processor	Node	# on Node
P00	0	0
P01	0	1
P02	0	2
P03	0	3
P04	1	0
P05	1	1
P06	1	2
P07	1	3
P08	2	0
P09	2	1
P10	2	2
P11	2	3
P12	3	0

• • •

1004

FIG. 10

Time 1 PROCESSOR ASSIGNMENT (Unchanged)

O/S Processor	Node	# on Node
P00	0	0
P01	0	1
P02	0	2
P03	0	3
P04	1	0
P05	1	1
P06	1	2
P07	1	3
P08	2	0
P09	2	1
P10	2	2
P11	2	3
P12	3	0
...		

Time 1 MEMORY ASSIGNMENT (Updated)

BLOCK	Current Owner	Last Owner
0 - FFFF	Node 0	Node 0
10000 - 1FFFF	Node 0	Node 0
20000 - 2FFFF	Node 2	Node 0
30000 - 3FFFF	Node 1	Node 1
40000 - 4FFFF	Node 1	Node 1
50000 - 5FFFF	Node 1	Node 1
60000 - 6FFFF	Node 2	Node 2
70000 - 7FFFF	Node 2	Node 2
80000 - 8FFFF	Node 0	Node 2
...		

SWAP

FIG. 11

Time 2 MEMORY ASSIGNMENT
(Same as Time 1)

BLOCK			Current Owner	Last Owner
0	-	FFFF	Node 0	Node 0
10000	-	1FFFF	Node 0	Node 0
20000	-	2FFFF	Node 2	Node 0
30000	-	3FFFF	Node 1	Node 1
40000	-	4FFFF	Node 1	Node 1
50000	-	5FFFF	Node 1	Node 1
60000	-	6FFFF	Node 2	Node 2
70000	-	7FFFF	Node 2	Node 2
80000	-	8FFFF	Node 0	Node 2
		...		

Time 2 PROCESSOR ASSIGNMENT
(Updated)

O/S Processor	Node	# on Node
P00	0	0
P06	0	1
P02	0	2
P03	0	3
P04	1	0
P05	1	1
P01	1	2
P07	1	3
P08	2	0
P09	2	1
P10	2	2
P11	2	3
P12	3	0
...		

SWAP

FIG. 12

A. CLASSIFICATION OF SUBJECT MATTER

IPC(8) - G06F9/46 (2014.01)
USPC - 718/1

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
IPC(8) - G06F9/46, G06F9/45, G06F9/50 (2014.01)
USPC - 718/1, 718/104, 718/102

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched
CPC - G06F9/45533, G06F9/5066, G06F9/5077 (2013.01)

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)

PatBase, Google Patents, Proquest Dissertations & Theses: The Sciences and Engineering Collection; ProQuest Technology Collection Google Scholar

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
Y	US 2011/0179132 A1 (MAYO et al) 21 July 2011 (21.07.2011) entire document	1-20
Y	US 2012/0020370 A1 (SONNIER et al) 26 January 2012 (26.01.2012) entire document	1-20
Y	US 2005/0223382 A1 (LIPPETT) 06 October 2005 (06.10.2005) entire document	4-8, 10, 11, 15-17, 19, 20
A	US 20040226026 A1 (GLASS et al) 11 November 2004 (11.11.2004) entire document	1-20

☐ Further documents are listed in the continuation of Box C.   ☐

* Special categories of cited documents:	"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention
"A" document defining the general state of the art which is not considered to be of particular relevance	
"E" earlier application or patent but published on or after the international filing date	"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone
"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)	"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art
"O" document referring to an oral disclosure, use, exhibition or other means	
"P" document published prior to the international filing date but later than the priority date claimed	"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
03 March 2014	**1 8 MAR 2014**

Name and mailing address of the ISA/US	Authorized officer:
Mail Stop PCT, Attn: ISA/US, Commissioner for Patents P.O. Box 1450, Alexandria, Virginia 22313-1450	Blaine R. Copenheaver
Facsimile No. 571-273-3201	PCT Helpdesk: 571-272-4300 PCT OSP: 571-272-7774

Form PCT/ISA/210 (second sheet) (July 2009)