Hardware Implemented Scheduler, Placer, Inter-Task Communications and IO System Functions for Manycore Processors Dynamically Shared among Multiple Applications

Mark Sandstrom
ThroughPuter, Inc.
mark@throughputer.com

ABSTRACT

To enable maximizing on-time processing throughput across multiple internally pipelined/parallelized applications on dynamically shared manycore processors by eliminating system software overhead, a hardware automated implementation of the parallel execution system functions is presented. In the presented implementation scenario, the manycore processor hardware provides, besides the processing cores, IO and memories, the system functions of monitoring the applications' processing loads, periodically (e.g. at microsecond intervals) allocating processing resources (cores) among the applications based on their processing load variations and contractual entitlements, prioritizing application task instances for execution, mapping selected task instances for execution on their assigned cores, and accordingly dynamically configuring the inter-task communications, IO and memory access subsystems (and on programmable hardware, the core slot types). The result pursued is a realtime application load and type adaptive manycore processor architecture, enabling scalable, secure, high-performance and resource-efficient, dynamic parallel cloud computing.

Categories and Subject Descriptors

C.2.4 [**Cloud computing**]

General Terms

Algorithms, Management, Performance, Design, Economics, Experimentation, Security, Standardization, Theory.

Keywords

Dynamic parallel execution, application load adaptive processing, hardware-automation of operating system functions.

1. INTRODUCTION

Traditionally, advancements in computing technologies have fallen into two categories. First, in the field conventionally referred to as high performance computing, the main objective has been maximizing the processing speed of one given computationally intensive program running on a dedicated hardware comprising a large number of parallel processing resources. Second, in the field conventionally referred to as utility or cloud computing, the main objective has been to most efficiently share a given pool of computing hardware resources among a large number of user application programs. Thus, in effect, one branch of computing technology advancement effort has been seeking to effectively use a large number of parallel processors to accelerate execution of a single application program, while another branch of the effort has been seeking to efficiently share a single pool of computing capacity among a large number of user applications to improve the capacity utilization.

However, there have not been major synergies between these two efforts; often, pursuing any one of these traditional objectives rather happens at the expense of the other. For instance, dedicating an entire parallel processor based (super) computer per individual application causes severely sub-optimal computing resource utilization, as much of the capacity would be idling much of the time. On the other hand, seeking to improve utilization of computing systems by sharing their processing capacity among a number of user applications using conventional technologies will cause non-deterministic, compromised performance for the individual applications, along with security concerns. As such, the overall cost-efficiency of computing is not improving as much as improvements toward either of the two traditional objectives would imply: traditionally, single application performance maximization comes at the expense of system utilization efficiency, while overall system efficiency maximization comes at the expense of individual application performance.

There thus exists a need for a new parallel computing architecture, which, at the same time, enables increasing the speed of executing application programs, including through execution of a given application in parallel across multiple processor cores, as well as improving the utilization of the available computing resources, thereby maximizing the collective application processing on-time throughput for a given cost budget. Moreover, even outside traditional high performance computing, the application performance requirements will increasingly be exceeding the processing throughput achievable from a single CPU core, e.g. due to the practical limits being reached on the CPU clock rates. This creates an emerging requirement for intra-application parallel processing (at ever finer grades) also for mainstream programs. Notably, these internally parallelized enterprise and web applications will be largely deployed on dynamically shared cloud computing infrastructure. Accordingly, the emerging form of mainstream computing calls for technology innovation supporting executing large number of internally parallelized applications on dynamically shared parallel processing resource pools.

Generally, dynamically optimizing resource usage in a large capacity parallel processing system among a large number of applications and their instances and tasks, in pursuing both predictable, high performance for each individual application as well as efficient system resource utilization, does present a complex problem, resolving which would consume plenty of the system's resources if handled in software. It is not trivial to answer the question: *To which application task instance should any given processing resource be assigned at any given time, to achieve optimal system-wide application processing throughput?*

2. MULTI-STAGE PARALLEL PROCESSING ARCHITECTURE

2.1 Overview

To address the above challenges, this paper presents an architecture for extensible, application program load and type adaptive, multi-stage manycore processing systems (Fig. 1). The presented architecture takes the following approach to enable scaling the dynamic resource optimization for increasing numbers (and types) of pooled processing resources and application programs (apps), their instances (insts) and tasks sharing the pooled resources:

1) The processing resources and app processing is partitioned into (manycore processor based) processing stages, which, per any given app, can be arranged to support various combinations of pipelined and parallelized processing. This brings the following benefits:
   a. The system has to support, per each processing stage, only one task per each of the apps dynamically sharing the system. At each processing stage though, there will be a dynamically optimized number of active insts of the local tasks of each app. The resource management for each stage is thus simpler than it would be for the full system, where there are multiple tasks per each app.
   b. The resource management is done independently for any given stage, which, besides being simpler due to there being just one task per app, limits the scope of the function, adding to the scalability of the architecture. Note that the dynamic resource optimization at each processing stage of the system, while done independently, is adaptive to the apps' processing load variations (incl. the processing input volumes received by any given stage from the other stages/external network inputs), so that the per-stage distributed dynamic resource management still achieves full system scope resource usage optimization.
2) The processing core resource management at each manycore based processing stage is further partitioned as follows:
   a. First, the allocation of the cores (of the local manycore processor) among the apps (i.e. their local tasks at that stage) is optimized periodically, based (in part) on the input processing load variations among the apps.
   b. Based on such core allocations, highest priority insts of the local app tasks are assigned for processing on a number of cores allocated to each given app. To minimize task switching overhead, continuing app-task insts are kept at their existing cores, and activating app-task insts are mapped to cores occupied by de-activating app-task insts -- on processors supporting multiple (dynamically reconfigurable) core types, so that the core types demanded by incoming app-task insts match, to the extent possible, the core type of their assigned core slots occupied by outgoing app-task insts.

By partitioning the system-wide dynamic resource management functionality per above, the individual functions of resource management for dynamically shared manycore arrays become feasible (e.g. in terms complexities of data structures needed) for direct hardware (e.g. FPGA) implementation. The all-hardware implementation of such system functions further adds to the scalability of the architecture (per Figs. 1-5) via system software overhead reduction. Since the hardware automated system functions do not consume any of the system processor capacity no matter how frequently the capacity is reallocated, and since the hardware algorithms run in just a few clock cycles, as well as since hardware automated task switching for the processor cores is non-visible to software, this architecture also enables re-optimizing the system resource assignment as frequently as needed to accommodate the apps' processing load variations. The main structures and elements of the architecture, and their operation, are described in the following.

2.2 Multi-stage Pipelined/Parallel Processing

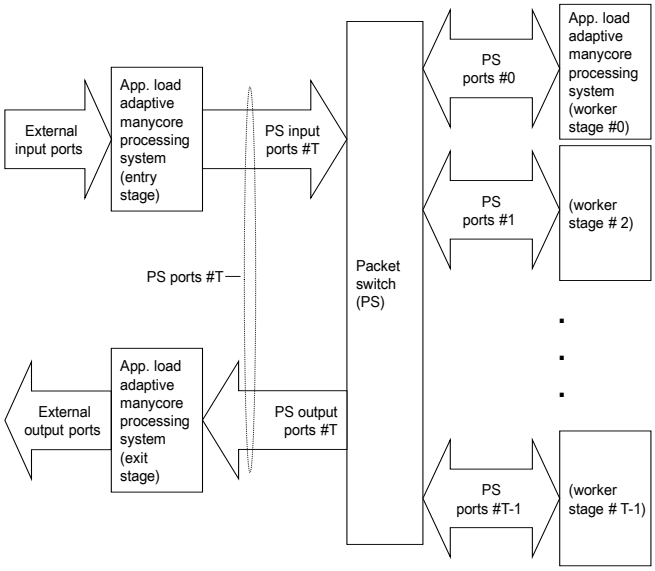

Figure 1. Multi-stage manycore processor system architecture.

General operation of the application load adaptive, multi-stage parallel data processing system per FIG 1., focusing on the main I/O data flows, is as follows: The system provides data processing services to be used by external parties (e.g. by client apps) over networks. The system receives data packets from its users through its network input ports, and transmits the processing results to the relevant parties through its network output ports. Naturally the network ports of the system of Fig. 1 can be used also for connecting with other resources and services (e.g. storage, data bases etc.) as/if necessary to produce the requested processing results. The app tasks executing on the entry stage manycore processor are typically of 'master' type for parallelized/pipelined apps, i.e., they manage and distribute the processing workloads for 'worker' type tasks running on the worker stage manycore processing systems (note that the processor system hardware is similar for all instances of the processing system). The insts of master tasks typically do pre-processing (e.g. message/request classification, data organization) and workflow management based on input packets, and then typically involve appropriate worker tasks at their worker stage processors to perform the data processing called for by the given input packet(s), potentially in the context of and in connection with related input and/or stored data elements. (The processors can have access to system memories through interfaces additional to the IO ports shown in the Figs.) Accordingly, the master tasks typically pass on the received data units (using direct connection techniques to allow most of the data volumes being transferred to bypass the actual processor cores) through the inter-stage packet-switch (PS) to the worker stage processors, with the destination app-task inst identified for each data unit.

2.3 Inter-Stage Data Flow and Processing Load Balancing

The any-to-any connectivity among the app-tasks of all the processing stages provided by the PS (Fig.1) enables organizing the worker tasks (located at the array of worker stage processors) flexibly to suit the individual needs (e.g. task inter-dependencies) of any given app on the system: the worker tasks can be arranged to conduct the work flow for the given app using any desired combinations of parallel and pipelined processing. E.g., it is possible to have copies of a particular (data parallelizable) task of a given app located on any number of the worker stages in the architecture per Fig. 1, to provide a desired number of parallel copies of a given app task. The set of apps configured to run on the system have their tasks identified by (intra-app) IDs according to their descending order of relative workload levels. The sum of the intra-app task IDs (with each ID representing the workload ranking of its task within its app) of the app-tasks hosted at any given processing system is equalized by appropriately locating the tasks of differing ID#s, i.e. of differing workload levels, across the apps for each processing stage, to achieve optimal overall load balancing. For instance, in case of four worker stages, if the system is shared among four apps and each of that set of apps has four tasks, for each app of that set, the busiest task (i.e. the worker task most often called for or otherwise causing the heaviest processing load among tasks of the app) is given task ID#0, the second busiest task ID#1, the third busiest ID#2, and the fourth ID#3. To balance the processing loads across the apps among the worker stages of the system, the worker stage #t gets task ID#t+m (rolling over at 3 to 0) of the app ID #m (t=0,1,…T-1; m=0,1,…M-1). In this example scenario of four apps, four worker tasks per app as well as four worker stages, the above scheme causes the task IDs of the set of apps to be placed at the processing stages per Tbl. 1 below:

App ID# m (to right) / **Worker stage# t (below)**	**0**	**1**	**2**	**3**
0	0	1	2	3
1	1	2	3	0
2	2	3	0	1
3	3	0	1	2

Table 1.

As seen in the example of Tbl. 1, the sum of the task ID#s (with each task ID# representing the workload ranking of its task within its app) is the same for any row i.e. for each worker stage. Applying this load balancing scheme for differing numbers of processing stages/tasks and apps is straightforward based on this example, so that the overall task processing load is to be, as much as possible, equal across all worker-stage processors of the system. Advantages of such schemes include optimal utilization efficiency of the processing resources and minimizing the possibility or effects of any of the worker-stage processors forming system-wide performance bottlenecks.

2.4 Application-Load Adaptive Manycore Processor Architecture

From here, we continue by exploring the internal structure and operation of a given processing stage, a high level functional block diagram of which is shown in Fig. 2 below.

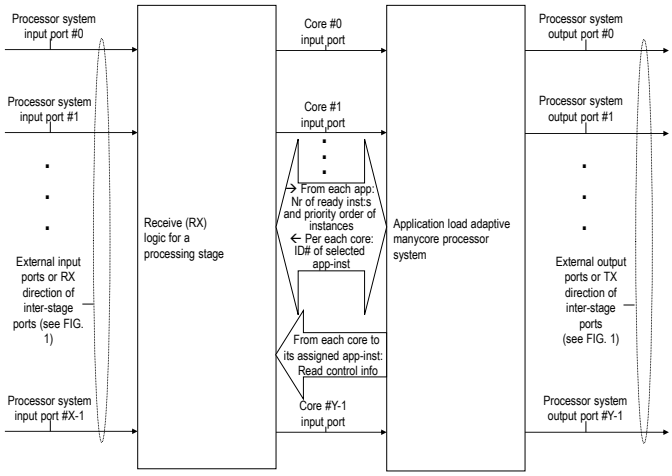

Figure 2. Top-level diagram for any of the processing stages in the multi-stage parallel processing system in Fig. 1.

Per Fig. 2, any of the processing stages of the system (Fig. 1) has, besides the manycore processor system (Figs. 3-5), an RX logic subsystem, which connects input data packets from any of the input ports to any of the processing cores of the processing stage, according to at which core the indicated destination app-inst of any given packet may be executing at any given time. Moreover, the monitoring of the buffered input data load levels per each destination app-inst at the RX logic subsystem enables optimizing the allocation of processing core capacity of the local manycore processor among the app tasks hosted on that processing stage. Internal elements and operation of the application load adaptive manycore processor system are illustrated in Fig. 3. Since there is one task per app per processing stage (though there can be multiple insts of any app-task at its local processing stage), the term app-inst in the context of a single processing stage means an instance of an app-task hosted at the processing stage under study.

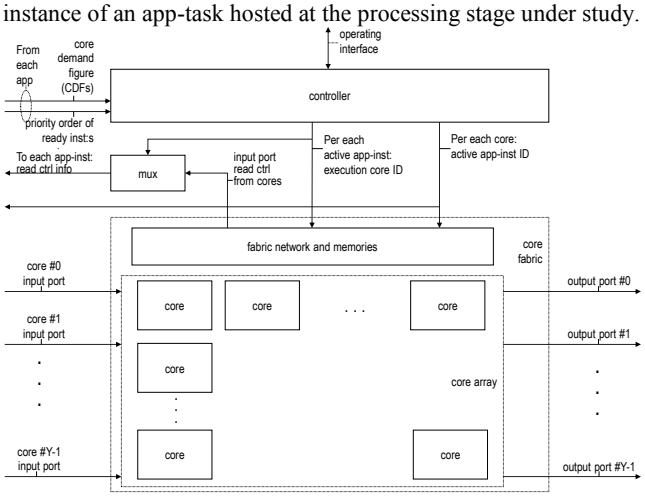

Figure 3. Application load adaptive manycore processor for the processing stage per Fig. 2 (within the multi-stage parallel processing system per Fig. 1).

Fig. 3 provides a block diagram for the manycore processor system dynamically shared among insts of the locally hosted app-tasks, with capabilities for application processing load adaptive allocation of the cores among the apps, as well as for dynamically reconfigured IO and memory access by the app-task insts. Any of the cores of a processor per Fig. 3 can comprise any types of

processing hardware resources, e.g. central processing units (CPUs), graphics processing units (GPUs), digital signal processors (DSPs) or application specific processors (ASPs) etc., and in programmable logic (FPGA) implementation, the core type for any core slot is furthermore reconfigurable per expressed demands of its assigned app-task. App specific logic modules at the RX module (Fig. 2) write their associated apps' capacity demand indicators, core-demand-figures (CDFs), to the controller of the local manycore processor. The CDFs express how many cores their associated app is presently able to utilize for its ready to execute insts. Each app's capacity demand expressions for the controller further include a list of its ready insts in an execution priority order. Criteria for prioritizing app-insts for execution includes whether a given inst has available to it such input data and fast-access memory contents that enable it to execute at the given time. The hardware logic based controller module within the processor system, through a periodic process, allocates and assigns the cores of the processor among the set of apps and their insts (in part) based on the CDFs of the apps. This app-inst to core assignment process is exercised periodically, at intervals such as once per a defined number (e.g. 1024) of processing core clock or instruction cycles. Fig. 4 below provides a data flow diagram for the hardware implemented controller, which periodically, e.g. once per microsecond, selects app-insts for execution, and places each selected-to-execute app-inst to one of the cores of the local manycore processor. As shown in Figs. 2 and 3, the app-inst to core mapping info also directs muxing of input data from the RX buffers of an appropriate app-inst to each core of the array, as well as muxing of the read control signals from the core array to the RX buffers of the app-inst that is assigned for any given core at any given time.

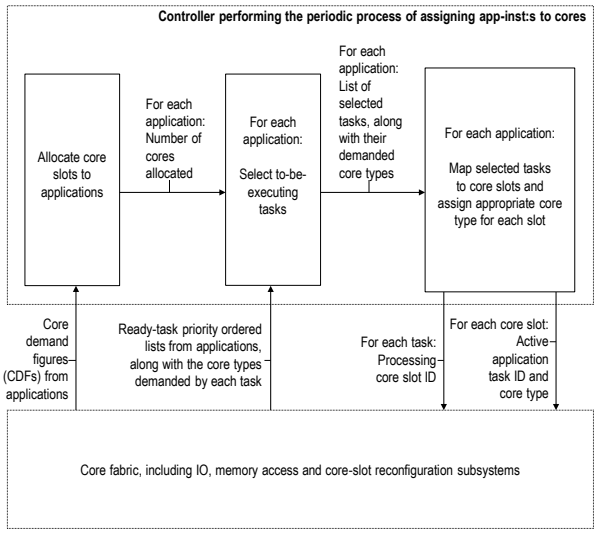

Figure 4. App-inst to core mapping process for the manycore processor per Fig. 3.

Fig. 4 presents major phases of the app-inst to core mapping process, used for maximizing the value-add of the app processing throughput of the manycore fabric shared among a number of apps. This process, periodically selecting and mapping the to-be-executing insts of the set of app-tasks to the array of processing cores of the local processor, involves the following steps:

(1) allocating the array of cores among the set of apps, based on CDFs and contractual entitlements of the apps, to produce for each app a number of cores allocated to it (between the current and the next run of the process); and
(2) based at least in part on the allocating, for each given app that was allocated one or more cores:
   (a) selecting, according to the inst priority list of the given app, the highest priority insts of the app for execution corresponding to the number of cores allocated to the given app, and
   (b) mapping each selected app-inst to one of the available cores of the array, to produce,
      i) per each core of the array, an identification of the app-inst that the given core was assigned to, and
      ii) per each app-inst selected for execution on the fabric, an identification of its assigned core.

The periodically produced and updated outputs of the controller process are used for periodically re-configuring connectivity through the RX subsystem (Fig. 2) as well as the fabric memory access subsystem (Fig. 5).

2.5 Fabric Memory Access Subsystem for Dynamically Allocated Manycore Processor

Fig. 5 and related specifications below, along with the reference [1] (in particular its figures 8-10) describe the manycore processor on-chip memory access subsystem providing non-blocking processing memory access (incl. for program instructions and interim processing results) between the app-insts dynamically assigned to cores of the array and the app-inst specific memories at the memory array of the core fabric. The capabilities per Fig. 5 provide logic, wiring, memory etc. system resource efficient support for executing any app-inst at any core within the processor at any given time (as controlled by the controller that periodically optimizes the allocation and assignment of cores of the array among the locally hosted app-insts), while keeping each given app-inst transparently connected to its own (instruction and interim data containing) memory block at memory array.

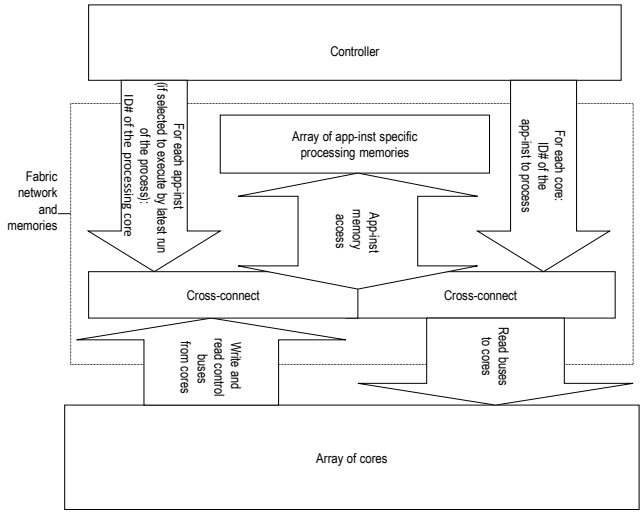

Figure 5. Dynamically reconfigured access by app-insts dynamically assigned for execution at the core array to app-inst specific memory blocks within the core fabric.

Per Fig. 5, to direct write and read control access from the array of cores to the array of app-inst specific memories, the controller identifies, for app-inst specific muxes at the cross-connect (XC)

between the core array and memory array, the presently active source core (if any) for write and read control access to each given app-inst specific segment within the fabric memory array. Similarly, to direct read access by the array of cores to the array of app-inst specific memories, the controller identifies, for core specific muxes at the XC, the memory segment of the app-inst presently assigned for each given core of the array. Based on the control by the controller for a given core indicating that it will be subject to an app-inst switchover, the currently executing app-inst is made to stop executing and its processing state from the core is backed up to the segment of that exiting app-inst at the memory array, while the processing state of the next app-inst assigned to execute on the given core is retrieved to the core from the memory array. Cores not indicated by controller as being subject to app-inst switchover continue their processing uninterruptedly through the core allocation period transitions. Note that applying of updated processing core ID# configurations for the app-inst specific mux:s at the XC and app-inst ID# configurations for the core specific mux:s of the XC (Fig. 5) as well as of the RX logic (Fig. 2) can be safely and efficiently done by the hardware logic without software involvement, since none of the app-insts needs to know whether or at which core itself or any other app-inst is executing within the system at any given time. Instead of relying on knowledge of the their respective previous, current (if any at a given time) or future execution cores by either the application or any system software, the architecture enables flexibly running any insts of any app-tasks at any core of their local processing stages.

2.6 Specifics of the Application Instance to Core Assignment Process

2.6.1 Hardware automation of dynamic resource management

To enable rapidly re-optimizing the allocation and assignment of the system processing core capacity among the insts and tasks of the apps sharing the processing system per Fig. 1 according to the realtime processing load variations among the app-task-insts, the dynamic resource management processes are implemented by hardware logic in the manycore processor controller modules per Fig. 4. Similar processes are run (independently) for each of the processing stages of a given multi-stage manycore processor system per Fig. 1. The application processing load adaptive, dynamic core assignment process per Fig. 4 comprises algorithms for core allocation, app-inst selection and mapping, as detailed in the following.

2.6.2 Algorithm for allocating the cores among the applications

Objectives for the core allocation algorithm include maximizing the processor core utilization (i.e., generally minimizing, and so long as there are ready app-insts, eliminating, core idling), while ensuring that each app gets at least up to its entitled (e.g. a contract based minimum) share of the processor core capacity whenever it has processing load to utilize such amount of cores. Each app sharing a given manycore processor (Fig. 3) is specified its entitled quota of the cores, at least up to which number of cores it is to be allocated whenever it is able to execute on such number of cores in parallel. Naturally, the sum of the apps' core entitlements (CEs) is not to exceed the total number of core slots in the given processor. Each app on the processor gets from each run of the core allocation algorithm:

(1) at least the lesser of its (a) CE and (b) core demand figure (CDF) worth of the cores; plus
(2) after condition (1) is met for all apps sharing the processor, as many additional cores to match its CDF as is possible while maintaining fairness among apps whose CDF is not fully met; plus
(3) the app's fair share of any cores remaining unallocated after conditions (1) and (2) are met for all the apps.

This algorithm allocating the cores to apps runs as follows:

(i) First, any CDFs by all apps up to their CE of the cores within the array are met. E.g., if a given app #P had its CDF worth zero cores and entitlement for four cores, it will be allocated zero cores by this step (i). As another example, if a given app #Q had its CDF worth five cores and entitlement for one core, it will be allocated one core by this stage of the algorithm. However, to ensure that each app-task will be able at least to communicate at some defined minimum frequency, the step (i) of the algorithm allocates for each app, regardless of the CDFs, at least one core once in a specified number (e.g. sixteen) of the core allocation periods.
(ii) Following step (i), any processing cores remaining unallocated are allocated, one core per app at a time, among the apps whose CDF had not been met by the amounts of cores so far allocated to them by preceding iterations of this step (ii) within the given run of the algorithm. For instance, if after step (i) there remained eight unallocated cores and the sum of unmet portions of the app CDFs was six cores, the app #Q, based on the results of step (i) per above, will be allocated four more cores by this step (ii) to match its CDF.
(iii) Following step (ii), any processing cores still remaining unallocated are allocated among the apps evenly, one core per app at time, until all the cores of the array are allocated among the set of apps. Continuing the example case from steps (i) and (ii) above, this step (iii) will allocate the remaining two cores to certain two of the apps (one for each). Apps with zero existing allocated cores, e.g. app #P from step (i), are prioritized in allocating the remaining cores by this step (iii).

Moreover, the iterations of steps (ii) and (iii) per above are started from a revolving app ID# within the set, so that the app ID# to be served first by these iterations is incremented by one (and returning to 0 after reaching the highest app ID#) for each successive run of the algorithm.

Accordingly, all cores of the array are allocated on each run of the above algorithm according to apps' processing load variations while honoring their contractual entitlements. I.e., the allocating of the array of cores by the algorithm is done in order to minimize the greatest amount of unmet demands for cores (i.e. greatest difference between the CDF and allocated number of cores for any given app) among the set of apps, while ensuring that any given app gets its CDF at least within its CE met on each successive run of the algorithm.

2.6.3 Algorithm for assigning app-insts for the cores

Following the allocation of the array of cores among the apps, for each app on the processor that was allocated one or more cores by the latest run of the core allocation algorithm, the individual ready-to-execute app-insts are selected and mapped to the number of cores allocated to the given app. One of the selected app-insts is assigned per one core by each run of this algorithm.

The app-inst to core assignment algorithm for each given app begins by keeping any continuing app-insts, i.e., app-insts selected to run on the core array both on the present and the next core allocation period, mapped to their current cores. After that rule is met, any newly selected insts for the given app are mapped to available cores. Assuming that a given app was allocated k (a positive integer) cores beyond those used by its continuing app-insts, k highest priority not-yet-mapped app-insts of the app are chosen to be mapped to the remaining available cores allocated to the given app, starting from the insts that are ready-to-execute.

When the app-inst to core mapping module of the controller (Fig. 4) gets an updated list of selected insts for the apps (following a change in either or both of core to app allocations or app-inst priority lists of one or more apps), it identifies from them the following:

I. The set of activating, to-be-mapped, app-insts, i.e., selected app-insts that were not mapped to any core by the previous run of the placement algorithm;
II. The set of deactivating app-insts, i.e., app-insts that were included in the previous, but not in the latest, selected app-inst lists; and
III. The set of available cores, i.e., cores which in the latest assignment table were assigned to the set of deactivating app-insts (set II above).

The sets I and II can be obtained as the incoming and outgoing app-insts for each of the cores for which the two are different. The app-inst to core assignment algorithm uses the info from the above sets to map the active app-insts to cores of the array so as to keep the continuing app-insts executing on their present cores, thus maximizing the utilization of the core array for user app processing, and by mapping the individual app-insts within the set I of activating app-insts for processing at the set III of available cores (according to their increasing app-inst and core IDs).

Moreover, regarding placement of activating app-insts (set I as discussed above) on processors with reconfigurable core slots, the assignment algorithm seeks to minimize the amount of core slots for which the activating app-inst demands a different execution core type than the deactivating app-inst did. I.e., the app-inst to core assignment algorithm will, to the extent possible, place activating app-insts to such core slots (within the core array of the local processor) where the deactivating app-inst had the same execution core type. E.g., activating app-inst demanding the DSP type execution core will be placed to the core slots where the deactivating app-insts also had run on DSP type cores. This sub-step in placing the activating app-insts to their target core slots uses as one of its inputs the new and preceding versions of the core slot ID indexed active app-inst ID and core type arrays, to allow matching the activating app-insts and the available core slots according to the core type, in order to minimize the need for core slot reconfigurations. For details on the core slot dynamic reconfiguration, please see [2].

3. CONCLUSIONS

Optimizing dynamic resource allocation on parallel processing resource pools shared among a number of internally parallelized and/or pipelined applications is a complex challenge, particularly when pursuing predictable, high performance (on-time processing throughput) for each of the individual applications as well as system-wide cost-efficiency, including in terms of efficient resource usage. Moreover, the resource allocation is merely a starting point for the overall challenge of orchestrating the execution of multiple concurrent applications on a dynamically shared parallel processing hardware: in addition, there needs to be a solution for handling the dynamic parallel execution routines, such as appropriately connecting the inter-task communications among the tasks of the application instances, and keeping each executing application task instance connected to its own processing context, while such application task instances are dynamically scheduled and placed on the shared pool of processing cores.

Conventional computing paradigms have relied on system software for handling the dynamic resource management etc. parallel execution routines. However, by considering the data volumes and processing intensiveness of handling the functions per above in software when trying to scale up the number of pooled processing resources as well as the number of applications and their tasks sharing such resource pools, and while trying to increase the frequency of resource allocation optimization, it becomes clear that the system software would eventually begin consuming a disproportionately high amount of the processing capacity of the given pool, to the degree that plain scaling of conventional architectures will lead not only to reducing resource utilization efficiency, but eventually also to decreasing system-wide application on-time processing throughput: after some point, the incremental processing resources, applications and tasks would begin to increase the overhead rate per a processing core so severely that the incremental scaling units would begin to reduce the app throughput of all the cores in the given pool by a factor greater than they would increase the system-wide app processing throughput capacity.

The presented architecture is designed to provide hardware logic based approach to the above scalability challenge being faced when seeking to improve both the individual application on-time processing throughput as well as the system-wide cost-efficiency and scalability of high volume, multi-user (e.g. cloud) computing. To the description herein, the reference [3] adds descriptions of (i) billing methods with incentive system for maximizing the amount of processing resources available to meet processing load demand peaks of the user applications sharing the given system, (ii) a memory access system that both seeks to keep the on-chip fast-access memory contents optimal w.r.t. to the presently active application-task instances' needs as well as uses the readiness of app-task insts fast-access memory contents as a factor in optimally scheduling such insts for execution, (iii) inter-application performance isolation for inter-task communications, and (iv) hardware logic based load balancers for a cluster of multi-stage manycore processing systems per this paper.

4. REFERENCES

[1] Sandstrom, M. 2012. US patent application #13684473. Application Load Adaptive Multi-stage Parallel Data Processing Architecture.
[2] Sandstrom, M. 2012. US patent application #13717649. Application Load and Type Adaptive Manycore Processor Architecture.
[3] Sandstrom, M. 2014. US patent application #61934747. Dynamic Parallel Execution.
[4] Sandstrom, M. 2014. US patent application #14318512. Concurrent Program Execution Optimization.