Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2010, Article ID 261434, 21 pages
doi:10.1155/2010/261434

Research Article

Hardware-Enabled Dynamic Resource Allocation for Manycore Systems Using Bidding-Based System Feedback

Theocharis Theocharides,[1] Maria K. Michael,[1] Marios Polycarpou,[1] and Ajit Dingankar[2]

[1] *Department of Electrical and Computer Engineering, KIOS Research Center for Intelligent Systems and Networks, University of Cyprus, 1678 Nicosia, Cyprus*
[2] *Client Components Group, Intel Corporation, Folsom, CA, USA*

Correspondence should be addressed to Theocharis Theocharides, ttheocharides@ucy.ac.cy

Received 28 May 2010; Revised 6 October 2010; Accepted 13 October 2010

Academic Editor: Shuvra Bhattacharyya

Manycore architectures are expected to dominate future general-purpose and application-specific computing systems. The ever-increasing number of on-chip processor cores and the associated interconnect complexities present significant challenges in the design, optimization and operation of these systems. In this paper we investigate the applicability of intelligent, dynamic system-level optimization techniques in addressing some manycore design challenges such as dynamic resource allocation. In particular, we introduce hardware enabled system-level bidding-based algorithms as an efficient and real-time on-chip mechanism for resource allocation in homogeneous and heterogeneous (MPSoC) manycore architectures. We have also developed a low-level simulation framework, to evaluate the proposed bidding-based algorithms in several on-chip network-connected manycore configurations. Experimental results indicate performance improvements between 8%–44%, when compared to a standard on-chip static allocation, while achieving a balanced workload distribution. The proposed hardware was synthesized to show that it imposes a very small hardware overhead to the overall system. Power consumption of the embedded mechanism as well as energy consumption due to additional network traffic for collecting system feedback are also estimated to be very small. The obtained results encourage further investigation of the applicability of such intelligent, dynamic system-level algorithms for addressing additional issues in manycore architectures.

1. Introduction

Manycore architectures are expected to become the dominant trend in both general purpose and application-specific processor architectures. It is anticipated that the number of on-chip processing cores will increase significantly in the near future, with the possibility of hundreds (and even thousands) of cores placed on a single die. Manycore architectures essentially consist of a large number of processor cores (typically, greater than eight), possibly heterogeneous, interconnected together and behaving as a massively parallel computer system [1, 2]. Core communication in these systems will be facilitated via packet-based, high bandwidth, on-chip interconnection networks (NoCs) [3, 4].

While microarchitectural mechanisms and *instruction level parallelism* (ILP) have improved the performance of uniprocessor systems to great extents, it is evident that future manycore systems will require alternative/additional mechanisms for performance optimization in order to fully utilize the large number of available processing cores [2, 5]. Manycore systems are expected to operate on the principle of *thread-level parallelism* (TLP) (or task-level parallelism for application-specific systems) rather than ILP. Hence, system-level approaches for performance optimization become more attractive than fine grained core-level mechanisms for these systems. In addition, the increased on-chip communication and synchronization complexity among the large number of cores in a manycore chip give rise to new paradigms for modeling, designing, and optimizing such systems [1, 2, 5].

A particular aspect when investigating the design of next generation manycore architectures involves porting of certain operations currently performed by the Operating System

(O/S) or the runtime system in dedicated hardware units (or cores). Control operations such as synchronization and resource allocation, which are vital operations in manycore architectures and for which the O/S is currently predominantly responsible, could potentially be designed using either dedicated hardware mechanisms or hardware/software co-design, with small hardware overhead. An advantage of this scenario is the use of truly dynamic, system-level algorithms for such operations, where feedback, received from the cores and the associated interconnect and input/output units, can potentially be used in real-time adjustment of the system operation, acting as a self-adjusting dynamic knob [6]. One critical design optimization challenge in manycore systems is the efficient assignment of program tasks to each of the processing elements (PEs/cores) available. Traditionally known as resource allocation or task assignment in off-chip multiprocessor architectures and monolithic processors, it has been a part of the O/S, and has been extensively researched (e.g., see [7, 8], among many others). Implementing a resource allocation engine using dedicated hardware, allows for the utilization of real-time knowledge of the system status (in our case, the system status is given in terms of PE/core utilization and on-chip network traffic), which in turn can lead to more intelligent decisions by the resource allocation algorithm, especially as the number of cores and the complexity of the on-chip network are increased.

In this paper, we consider intelligent, runtime, system-level optimization mechanisms, embedded on-chip, that can be used to address some key manycore challenges. We focus on the problem of intelligent resource allocation for manycore architectures (homogeneous and heterogeneous) and propose a hardware enabled, system-level dynamic mechanism that assigns processes to cores based on *bidding algorithms* [9–12], in an attempt to improve the overall system performance. We focus on a simple bidding-based approach, since it requires small hardware resources and, therefore, can be implemented and utilized as a real-time on-chip resource allocation algorithm. Furthermore, we have developed a low-level simulation framework in order to accurately evaluate the proposed mechanisms. We experimented with both homogeneous chip multiprocessor architectures (CMPs) as well as heterogeneous multiprocessor systems-on-chip architectures (MPSoCs), using various on-chip network sizes and topologies. In both cases, we observe significant improvement in system performance from a comparable (on-chip) standard mechanism such as static round robin allocation. The implemented bidding algorithm is very simple, with small overall hardware overhead (between 2.5%–6%). Anticipating that future manycore applications will emphasize thread-level speculation and target thread-level parallelism, we form a set of benchmarks based on small, general-purpose, and application specific highly parallel tasks in order to evaluate the proposed mechanisms. Hence, performance is evaluated using both real-world applications and representative synthetic benchmarks.

This paper extends our previous work in [13, 14], expanding in the process the benchmark applications, the interconnection architectures and topologies, and the scale (number of cores) of the system. Additionally, the paper presents a more detailed view of the hardware architecture, along with the associated hardware performance metrics. Moreover, we detail the benchmark suites that we developed for evaluating the presented algorithms as well as the multimedia applications that were used as part of the evaluation process. We expect that this work will provide motivation for further research on dynamic, system-level optimization algorithms, addressing various manycore issues such as power-aware resource allocation, energy consumption, and interconnect optimization.

The rest of this paper is organized as follows. Section 2 discusses some preliminary concepts related to system-level optimization, focusing on the resource allocation problem on CMPs and MPSoCs as well as some basic concepts of bidding-based algorithms. Previous work on resource allocation for CMPs and MPSoCs is discussed in Section 3. Section 4 presents the targeted underlying system architecture considerations, explaining the motivations behind the architectural assumptions taken. Section 5 introduces the considered resource allocation framework and describes the proposed system-level algorithms and their hardware implementation. Section 6 details the experimental procedure and the developed simulation framework, and explains the benchmarking approach taken. Section 7 reports and discusses the obtained results, and Section 8 concludes the paper and gives future research directives.

2. Background

2.1. System-Level Optimization. Runtime system-level optimization algorithms have been employed with the purpose of dynamically adjusting the system's parameters through knowledge of the system's status, in order to improve the system operation at that given time. In the manycore case, the improvement can target several issues such as performance, and reduction in energy consumption (and subsequently temperature) as well as system reliability.

In general, system level optimization algorithms operate on the principle of feedback received from the system in periodic intervals and system operational constraints at a given time. These are subsequently used to update the algorithm's optimization criteria, in order to optimize the global system performance subject to certain system constraints. Such algorithms can be embedded as part of the on-chip hardware (such as a custom processor). Figure 1 shows one such scenario, where the on-chip optimization unit is encapsulated as an individual processing element, part of the manycore system (light-shaded block), with its internal operation illustrated on the left and below. The optimization unit receives runtime information $y(t)$ about the status of the system (processing elements and the communication fabric) at time t and operational constraints ($u(t)$) given by the application requirements also at time t, and in turn, decides, based on one or more optimization goals, how to modify the system's operation $z(t+dt)$ at the future time $t+dt$. There can be obviously more than one dedicated unit, depending on the underlying architecture, optimization algorithm, feedback, and application operational constraints. Additionally, the granularity of the optimization interval (dt) is an important

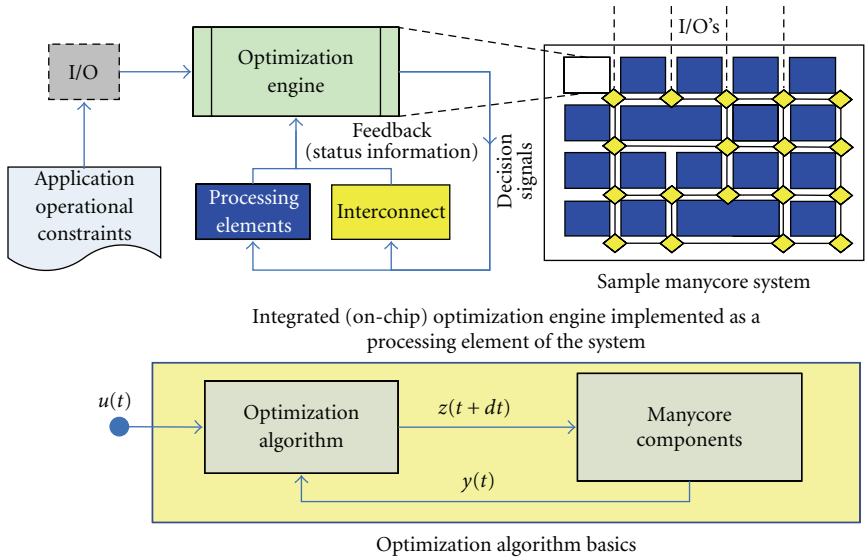

Figure 1: Integrated system-level optimization algorithm (above) and its basic operation (below).

parameter in determining the location of the mechanism in the network topology.

On-chip implementation favors cases where frequent optimization actions are necessary, since system feedback can be collected quickly and decision and optimization steps can happen in real time. Real-time information such as the utilization of each component and the state of the communication network can result in efficient workload distribution; this is particularly important in MPSoC environments where tasks are usually small and periodic, and thus frequent (and predictable) allocation takes place. It also holds for future large-scale CMPs systems with simple PEs/cores, running parallelized code. Clearly, for efficient on-chip implementation, the chosen optimization algorithm needs to be fast and simple (its implementation has to impose small hardware overhead). In cases where the optimization actions can be taken during sparse intervals, real-time feedback may not be as crucial. In such cases, more sophisticated and complex optimization algorithms can be employed using software (as part of the O/S or runtime system), avoiding a possibly expensive hardware implementation.

Our focus in this work is to investigate the applicability of different on-chip system-level optimization algorithms and the impact of introducing real-time intelligent processing in such algorithms (using the problem of resource allocation as an example) in networked manycore systems.

2.2. Resource Allocation. Resource allocation in computing systems deals with the allocation of available system resources to the various tasks ready to be executed. This is a process that significantly affects the overall performance of the system. Typically, resource allocation algorithms take as input a list of tasks or processes that are ready to be executed at some particular time as provided by a system scheduler. The scheduler considers a task flow graph in order to resolve task dependencies. Traditionally, resource allocation in off-chip multiprocessor systems concentrates on the allocation of software tasks to each of the processor nodes (usually individual processors with local caches and memory), such that the overall performance of the system is maximized. This is a well-known problem, with a large amount of research contributions [7, 8] towards efficient utilization of the massive hardware parallelism available in such systems using various exact and heuristic approaches [11, 15–17]. In the case of manycore systems, important on-chip constraints such as limited buffer capacity for on-chip communication, on-chip network congestion, power density, and limited I/O bandwidth [5] necessitate the evolution of existing algorithms or even the development of new algorithms, in an effort to integrate the emerging challenges [3, 5]. In such dense systems, efficient workload distribution could benefit from real-time system information knowledge such as the status/utilization of each core and, additionally, the status of the interconnection network. Interconnect-associated delays are an important factor in efficient decision-making and the problem is further complicated when control and memory-related traffic, such as cache misses and synchronization data is taken into consideration.

Whilst dealing with a similar problem, research looks at general-purpose systems (CMPs) from a slightly different viewpoint when compared to application-specific systems (MPSoCs). General purpose systems face runtime uncertainties, where cache misses, data hazards, and unpredictable interconnect behavior can potentially alter the expected execution time of one task significantly. On the other hand, application-specific manycore systems (typically heterogeneous MPSoCs) usually deal with predictable schedules and execution times. Consequently, MPSoC-related research focuses mostly on finding optimal, static, design time allocation, where the mapping of tasks to the cores can either take place as part of the compilation, or mapped prior to execution on the processor cores. However, as the number of cores increases, MPSoCs are expected to contain groups of cores, where all cores in a group are of the same type. Any

core inside a group can potentially execute some task, shifting the task allocation process towards finding the best core that can execute one type of task among the cores in a group. As a result, allocation in future massively parallel CMPs and MPSoCs is expected to face similar issues and should not necessarily be treated independently.

2.3. Bidding. This work uses the concept of bidding to decide how to allocate processes to the various cores of the system. Bidding algorithms have been widely used to solve several optimization problems as part of auction-based algorithms [9–12]. Typically, for a specific number of items, there are n possible "bidders", with each "bidder" placing a bid in an attempt to "buy" a number of items. Usually, the highest bidder claims one or more items, and bidding continues until there are no more items or no more bidders [10]. Such algorithms can also be used in more complex scenarios, under various constraints. Bidding-based algorithms traditionally offer load balancing across distributed systems and networks, optimizing performance and utilization [10]. In the proposed algorithms, cores actively decide based on their workload and utilization of their related communication resources (on-chip network) whether, and to what extent, to bid for (request) additional process (task) assignment. Each core (or group of cores), computes its bid independently and sends it, through the on-chip network, to the system-level on-chip allocation engine. Transferring this decision to the cores is done with minimal overhead; it also improves flexibility and scalability of the system. Subsequently, the allocation engine decides where to assign the list of pending processes/tasks dispatched by the system scheduler, based on the placed bids. In this work, we present two different simple bidding-based algorithms for performing on-chip resource allocation in a manycore system, where processes are the "auctioned" items, and a processor core (or a group of cores) places its "bid" based on its status (or the status of each core in the group). We are aiming for a simple and fast bidding-based solution, rather than an optimal one, since we are targeting hardware implementation of the algorithm where speed and simplicity are critical factors. Still, the obtained experimental results demonstrate significant performance improvement and highly balanced utilization among the various cores. The presented algorithms improve the system's performance when compared to a standard (static) allocation mechanism such as round robin implemented in hardware. We use round robin as a hardware reference algorithm in evaluating our optimization algorithms, due to its simplicity to be implemented (very low hardware overhead) as well as the lack of other existing comparable solely hardware-based solutions for the specific problem under consideration.

3. Related Work

Dynamic system-level optimization has been extensively studied in software-based algorithms where an O/S routine acts as the optimization engine, addressing several system-level issues. For example, software-based, dynamic system-level thermal-aware optimizations involving scheduling and task allocation are presented in [18–21]. The authors in [22] use a two-level mapping technique for online allocation of streaming applications, while [23–26] introduce other dynamic techniques such as agent-based techniques, task migration, and spatial mapping. Additional examples involve runtime configuration of the on-chip interconnect for priority among application traffic [27], or on-demand core "merging" and "splitting", offering the necessary facilities for efficient application execution and different granularities of parallelism [28]. The concept of distributed system status gathering and real-time adjustments was also examined in [28], where the authors consider the problem of allocating a variable number of resources to each process, something that increases the hardware complexity. Nevertheless, this is a different problem to the one examined in this work, since a process is allocated to some resource out of a fixed number of resources, whereas in [28] the authors allocate a variable number of resources to a fixed number of processes.

The problem of resource allocation has been widely addressed, with a large number of proposed methodologies applicable to traditional multiprocessor systems and networks [11, 12, 15–17, 22]. There has also been extensive study of resource allocation applicable to single chip multiprocessors, that targets the same problem, but under different constraints, depending on the underlying architecture: general purpose CMPs or application-specific MPSoCs. As mentioned earlier, most of the related work focuses on software techniques encapsulated within the O/S.

Recent works propose the use of dedicated architectural (hardware) mechanisms to support task allocation on manycore systems, however the allocation algorithm remains still as part of the O/S. Preemptive techniques for task scheduling and allocation which involve thread migration, as a solution to balance the workload effectively, are proposed in [29, 30]. A hardware mechanism that is used to support the computation of a scheduling algorithm for symmetric CMPs has been proposed in [31]. Additionally, scheduling and resource allocation with emphasis on other design constraints such as energy and temperature has also been investigated, again using software-based algorithms supported by dedicated hardware mechanisms in [8, 18, 32]. Such preemptive methods usually require additional control hardware and are often associated with extra delays, in contrast to the simpler hardware support required for the nonpreemptive version of the problem that is examined in this work. Some other dynamic scheduling algorithms are also proposed in [11, 15, 16, 22, 33–39], however they are assumed to be maped on dedicated processor cores rather than custom hardware and take into consideration only the on-chip network status in reallocating tasks, but not the processing element status. Similar to previous methods, the majority of these techniques also apply preemption.

Task allocation for MPSoCs focuses mainly on meeting real-time constraints, which most MPSoC applications are bounded by, while maintaining operational constraints such as temperature, performance, and reliability. The majority of the existing work involve software-based mechanisms dealing with static MPSoC task allocation as part of the O/S or the runtime system and focuses on proposing algorithms which can efficiently optimize resource allocation, improving

the algorithm itself as well as the performance of the MPSoC. Recent works propose the use of runtime, multiobjective optimization algorithms such as genetic algorithms and multilevel optimization algorithms for thermal management, reliability and meeting worst-case execution times [15, 18, 22, 24–26, 40]. These works are also implemented as software routines, part of the runtime system or the O/S, and utilize system feedback (temperature and workload) in attempting to optimize the performance, energy consumption, and reliability of the system.

In this work, we propose a fully hardware-enabled allocation engine, which receives real-time system data relating to the workload of each core and the network traffic and dynamically allocates tasks coming from the system scheduler. Unlike previous works, the proposed algorithm does not impact the instruction set architecture or the microarchitecture of each core, making it applicable to both general purpose CMPs as well as MPSoCs. Moreover, the proposed hardware implementation alleviates cores from running the allocation task, allowing more execution time to user applications. By porting the algorithm completely in the system hardware, system feedback is received in real time, allowing the allocation engine to deal with frequent system changes as well as receive well-informed allocation decisions.

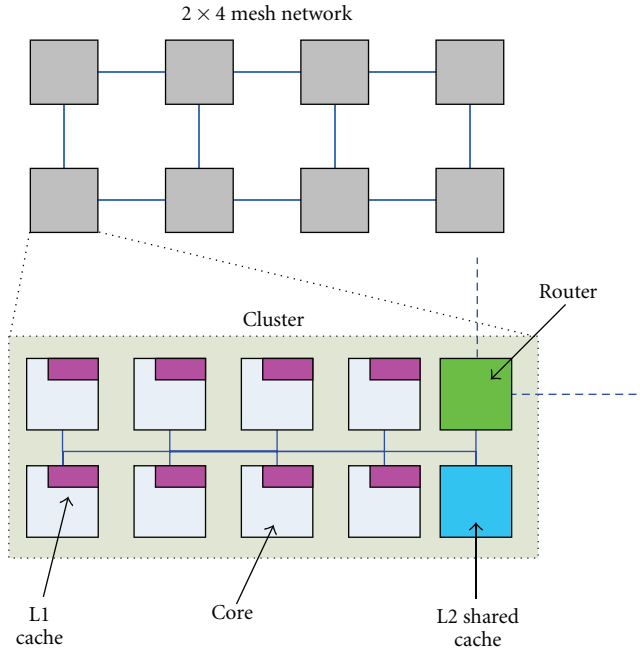

Figure 2: An example of a hierarchical NoC-based 64-core architecture.

4. Architectural Considerations

While architectural exploration of future manycore systems is not the objective of this paper, we aim in designing an algorithm that will be implementable and applicable in generic homogeneous and heterogeneous manycore architectures. It is assumed that these architectures will facilitate the execution of multiple, parallel processes/tasks. Cores are less likely to contain complicated control or resource and data dependency resolving hardware. Performance penalty associated with stalled pipelines in individual cores can be offset by the increased systemwide parallelism, which is achieved by running multiple processes in parallel as well as the elimination of complicated control flow hardware that traditional uniprocessor architectures employ. Furthermore, future manycore systems could utilize *thread-level speculation* (*TLS*) [41, 42], with speculative threads inserted to offset performance penalties associated with control and data dependences. By such methods such as TLS, emerging research suggests that future manycore systems may feature small parallel processes independent of each other, in an effort to minimize communication and parallelize computation [43, 44].

In general-purpose architectures, multiple parallel threads can be issued on processor cores, where threads from the same process can share data, but processes themselves are independent from each other. In an effort to compromise on thread level dependencies, we target an architecture that can facilitate execution of threads using a shared address space, and processes using a different address space. Motivated by the issues surrounding memory coherency protocols and memory organization [45], we consider a hierarchically connected networked architecture. In this scenario, groups of a small number of cores, called *clusters*, are connected via a local bus; clusters are in turn connected using an on-chip network. Each core has its own private L1 cache, and all cores within a cluster share an L2 cache. The size of the cluster is typically smaller than 8 cores, limited by the bus connecting the cores. Clusters are interconnected using hierarchical, packet-based NoC through on-chip routers [3, 4]. This architecture allows a process (which consists of a variable number of threads) to be dispatched to a cluster of cores, where shared address space threads belonging to the process can be executed in parallel (on different cores of the cluster) using standard cache coherency protocols. Thread communication is done at the cluster level, where threads share data and communicate with each other. This, given that clusters will not share common data between them (since sharing is done inside the cluster), alleviates overhead coherency traffic over the network. By dispatching processes inside each cluster, threads can execute on that cluster using the L1 and L2 caches, with only L2 misses being serviced by the on-chip network. Figure 2 shows a sample 2×4 clustered architecture, in which clusters consist of 8 cores and a shared L2 cache, and each core consists of a simple RISC core with L1 cache. The bus connecting all cores inside a cluster, is responsible for cache coherency, via an MSI-(modified-shared-invalid) based cache protocol. It must be noted that even if clusters share data between them, the algorithm still operates on the same principle, as memory traffic as well as intercluster communication and computation due to cluster stalls will be taken into consideration by the algorithm in each allocation interval.

In the proposed framework, scheduling and resource allocation are treated separately; the input to the considered

problem is a list of processes, as scheduled by the O/S and the compiler. In this paper, we only target the allocation part, the dispatching of a process to a cluster. Each process consists of a variable number of threads, to be executed by the allocated cluster's cores. The compiler provides precedence between processes, and for each process, the O/S and compiler provide an estimated execution time which is used as input to the bidding algorithm. For example, if we assume that the targeted cores consist of RISC CPUs with minimal hardware addressing control and data dependences, the instruction count for each process can be used as a metric. The actual execution time for each process may of course vary from the estimated one due to cache misses, and pipeline stalls (data and control hazards). The accuracy of the estimated time is not crucial; the relative time between the processes is utilized by the allocation algorithm. More importantly, the system feedback (bids from clusters or cores) considers stalls, memory misses and network delay in an implicit manner (as explained in the following section).

It is also assumed that at a given time, the scheduler sends independent processes to the system, but each process can contain threads dependent on each other. Therefore, branches within threads either target instructions from the same process (possibly the same thread as part of a loop) or result in a context switch. If the latter happens, it is assumed that a thread or a process has finished and a new one waits to be executed. Once a process terminates, the scheduler is dynamically informed. We anticipate that this flow model will be preferred for single-address space manycore architectures, as it removes costly prediction hardware from cores and eliminates unnecessary traffic from the on-chip network (such as invalid instructions, etc.). We must mention that L2 misses, that based on this scheme are a result of a context switch between processes, will result in bursts of memory traffic over the network, as most of these misses will happen concurrently. This is however handled as part of memory-oriented scheduling and is out of the scope of this paper. The considered architecture is based on generally accepted assumptions about general-purpose manycore architectures. The proposed allocation engine can be adopted for alternate (nonhierarchical) architectures [2], where single cores operate as the bidders. We experiment with both hierarchical cluster-based and non-hierarchical (traditional NoC-based) architectures in our simulations (see Section 7).

Similar assumptions are also taken when targeting MPSoC architectures in which heterogeneous components communicate with each other through the on-chip interconnection network. MPSoCs contain several types of cores, many of them replicated, in attempts to provide parallel execution such as processing concurrent video and audio streams. MPSoCs are typically application specific, with each component designed specifically for a certain computation. Additionally, they may include general-purpose embedded processors for running generic tasks and for control and management purposes. We clarify that the use of "task" and "process" is similar; the term *task* is typically used in MPSoCs to describe a certain application task, which contains instructions and data necessary to complete the task. During resource allocation, each task dispatched by the scheduler can be bound to a certain PE (or embedded processor) for execution, along with its associated data. As such, we consider a non-hierarchical (nonclustered) architecture in the MPSoC scenario. In the general-purpose case, we use the term *process*, as described earlier. Under the context of this work, we consider a *task* to be one individual entity, whereas a *process* is broken down into multiple small threads. As such, an application mapped on an MPSoC is broken down into small, data-independent tasks following an application taskgraph. Intertask communication for control purposes is facilitated using the on-chip interconnection network. This work does not address the memory mapping and allocation problem in MPSoCs. It is assumed that all task data will be accessible to on-chip PEs through dedicated memory elements (MEs) and that two tasks that share the same data will contain exclusive copies of each data variable inside the MEs. Clearly, this is a conservative approach which however can be easily relaxed using message passing via the on-chip network or even more sophisticated memory coherency protocols for intratask communication. The important point here is that any processor delays or network congestion due to memory accesses are taken into consideration during the algorithm allocation stage.

We classify a manycore system into three major components: the input/output blocks (I/O), the interconnection network (IN), and the PEs (cores). This abstract viewpoint subsequently treats the three individual units as black boxes, using only information transferred between the three units as input and feedback to optimize the system's operation. It also maintains the simplicity of the algorithm, and potentially makes the applicability of the algorithm scalable since all system expansion will still be viewed using the three subunit models.

5. System-Level Bidding-Based Resource Allocation Algorithms and Implementation

5.1. Problem Formulation and Basic Principles. The proposed bidding-based algorithms apply to both heterogeneous and homogeneous systems as well as clustered and nonclustered architectures, as explained in the previous section. In this section, we present the bidding-based allocation concept and algorithms as applied in the homogeneous clustered case. While extending the algorithms to the non-clustered homogeneous case (where each core bids for processes) is trivial, the heterogeneous MPSoC case (where typically cores of the same type bid for tasks) requires some modifications. This case is discussed in Section 5.4.

Two different system-level resource allocation algorithms are presented, both based on *bidding*. In this context, each individual cluster computes independently and submits a *bid* to the allocation engine, which centrally decides which resource (cluster) to allocate to each pending process. Here, we consider the clusters as the computational resources; however, the proposed bidding-based algorithms can also be applied hierarchically inside each cluster, to allocate threads

to cores (in our experiments, since the amount of cores inside each cluster is small, a round robin allocation of cores to process threads is considered).

Let the system be composed of n clusters, denoted by the list $\mathcal{C} = \{C_1, C_2, \ldots, C_n\}$. Each cluster consists of a number of cores given by $c(C_i) = \{c_1^i, c_2^i, \ldots, c_{mi}^i\}$, $C_i \in \mathcal{C}$. The number of cores per cluster (m_i) can be either the same or different among the clusters. The input to the allocation engine is a list of processes from the system scheduler, denoted by $\mathcal{P} = \{P_1, P_2, \ldots, P_x\}$, with an estimated execution time for each process as given by the compiler (measured in clock cycles), denoted by $t(P_i)$, $P_i \in \mathcal{P}$. Additionally, the engine receives feedback from the system in the form of a list of bids, one per cluster, given by $\mathcal{B} = \{b(C_1), b(C_2), \ldots, b(C_n)\}$. A bid $b(C_i)$ for some cluster C_i gives the *estimated new workload that the cluster can handle* (in terms of clock cycles), and it is calculated based on the remaining workload in the queue of each core in $c(C_i)$ as well as the network flow on the path towards the cluster. A zero-value bid ($b(C_i) = 0$) indicates that the cluster C_i is full and does not desire any additional processes (Section 5.2 gives details on how the bids are calculated).

Time is divided into time intervals of size dt, where the size of each interval (based on the system clock cycle) controls the granularity of the algorithm. For every time interval, the engine receives from the scheduler the list of processes $\mathcal{P}$ and $t(P_i)$ for all $P_i \in \mathcal{P}$, along with the list of bids $\mathcal{B}$ from each cluster. The number of processes in $\mathcal{P}$ can vary between intervals whereas n, the number of clusters and, hence, bids, is clearly fixed since it depends on the underlying architecture.

Both of the proposed algorithms accept the same input and use the same bid calculation mechanism, and proceed in a similar rationale to perform the allocation employing a *highest bidder* scenario. Their difference is on the highest bidder calculation method. In general, the following problem is targeted: *for some time interval* $[T, T + dt]$*, create a bind between each process* $P_i \in \mathcal{P}$, $1 \le i \le x$*, and some cluster* $C_j \in \mathcal{C}$, $1 \le j \le n$*, using a highest-bidder order where the cluster with the highest bid receives the process with the highest execution time* ($t(P_i)$). A cluster can be allocated to more than one process. A cluster with a zero bid is not allocated to any process. In the case where one or more processes cannot be bound to any cluster during the current interval (all clusters are full), the scheduler can be stalled, indicating that the system is full, until all scheduled processes can be bound to some cluster (in subsequent time intervals). This is of course a pessimistic approach, used to maintain precedence constraints between processes in different time intervals that could be further optimized. At this point, it is used in order to maintain the simplicity of the solution.

5.2. Bid Calculation. At some time instance T, the bid $b(C_i)$ of some cluster C_i is calculated based on two main parameters, both expressed in terms of clock cycles: (i) the remaining, already allocated, workload $w(C_i)$ at the cluster, and (ii) the network delay $d(C_i)$ on the path towards the cluster (estimated time for a new process to reach the cluster).

Let $w(c_j^i)$, $c_j^i \in c(C_i)$ denote the remaining workload at some processor core c_j^i of cluster C_i, which is calculated by counting the instructions in c_j^i's queue awaiting to be executed. An ideal execution time of one instruction per cycle is considered (as it is typical in RISC processors); the number of waiting instructions in a core's queue can then be used to represent the workload, in number of cycles, for that core. In this way, core stalls due to cache misses as well as branches of already executed instructions (that caused additional clock cycles to the ones estimated by the compiler) are implicitly considered. Once every core c_j^i calculates its corresponding $w(c_j^i)$, the cluster's workload is computed by

$$w(C_i) = \sum_{j=1}^{m_i} w\left(c_j^i\right). \tag{1}$$

The workload is computed through the use of dedicated hardware counters inside each core and through dedicated counters in the router, for computing the cluster workload. After a cluster C_i computes its workload $w(C_i)$, it sends its bidding message to the resource allocation engine that includes two values: $w(C_i)$ and a timestamp $s(C_i)$ of the time the packet was sent. When the engine receives the bidding packet from a cluster, it uses its own timestamp S and computes the amount of time (estimated in clock cycles) that the packet needed to travel through the network. If the bidding packet is constrained to travel on the exact network path that process data and instructions use to travel towards the targeted cluster, the delay of the network can be estimated by

$$d(C_i) = S - s(C_i). \tag{2}$$

Enforcing the bidding packet to use the same path that data/instruction packets use, can be done by reversing the routing algorithm for those packets marked as bidding packets; for example, in the case of an XY routing algorithm with X-first priority, the bidding packets would have to travel using Y-first priority instead, going through the same routers that the incoming data/instructions will travel in the next allocation interval. Bids also utilize a dedicated virtual channel for deadlock avoidance, but are subject to the same port arbitration as the regular process packets (data and instructions), so they are exposed to the network delay that regular packets are exposed. This method, while simple, gives a good estimate of the network delay. Relevant NoC literature suggests that busy routers do tend to remain busy for a number of cycles, and if a router has one blocked output port, there is increased probability that the other ports will block faster as well due to hotspot formation [26].

In order to calculate the estimated *new* workload that the cluster C_i can handle, which is actually the bid $b(C_i)$, it is necessary to establish the maximum amount of workload each cluster can be assigned. A straightforward measure is the total instruction queue space available in the cluster, which is

the sum of the instruction queue space per core in the cluster. Let this be denoted by $b_o(C_i)$. Then, a cluster's bid is given by

$$b(C_i) = \begin{cases} b_o(C_i) - w(C_i) - d(C_i) & \text{if } b_o(C_i) \geq w(C_i) + d(C_i), \\ 0 & \text{if } b_o(C_i) < w(C_i) + d(C_i). \end{cases} \quad (3)$$

When the system is entirely empty, $w(C_i) = 0$ and $b(C_i) = b_o(C_i) - d(C_i)$, for all $C_i \in \mathcal{C}$. In this case, the larger bidders will be the ones closer to the allocation engine and they will be allocated to the larger processes. On the other end, if $b(C_i) \leq 0$ then cluster C_i cannot accept any new workload and is out of the bidding contest. The latter is *always* encoded with a sentinel value of zero, indicating no bid.

5.3. System-Level Bidding-Based Algorithms. Both of the proposed system-level resource allocation algorithms use the same bid calculation method described in the previous subsection and follow a highest bidder in order to decide which cluster to allocate to a process. However, each uses a different method in determining the order among the various clusters.

The first algorithm, *Necessary Resorting* (NRS), is simpler in terms of operations performed and, therefore, faster. At some time instance T, it starts by sorting the list of clusters $\mathcal{C}$ and the list of processes $\mathcal{P}$ in decreasing order, based on the bid value $b(C_i) \in \mathcal{B}$ per cluster $C_i \in \mathcal{C}$ and the process size $t(P_j)$ per process $P_j \in \mathcal{P}$, respectively. Then, it binds the highest bidder (cluster at the top of $\mathcal{C}$) with the largest process (the one at the top of $\mathcal{P}$), the second highest bidder with the second largest process, and so on. This scenario tends to distribute the various processes among the available clusters in such a way that clusters with smaller, already allocated, workload and/or smaller network traffic are allocated to larger processes. Hence, workload balancing is inherently achieved without been explicitly targeted. This is demonstrated consistently by the obtained experimental results. Observe that it is possible for a cluster C_i to be allocated to a process P_j with size greater than the cluster's bid (i.e., $b(C_i) < t(P_j)$). This occurs only if no other cluster with a bid greater than $b(C_i)$ exists and continues to follow the overall rationale of the algorithm (larger processes are bound to clusters with smaller workload and/or network traffic). Since this is a binding (and not a dispatching) phase, allocating a cluster to a process whose size is greater than the cluster's bid does not create any problems.

In the case where the number of processes is greater than the number of clusters with positive bids, the algorithm updates the bids based on the allocation done so far and then performs a cluster resorting step in order to determine which clusters to allocate to the remaining processes. Bid update is carried out only for the *allocated* clusters by $b(C_i) = b(C_i) - t(P_j)$, where P_j is the process that has been already bound to cluster C_i. After re-sorting the cluster list $\mathcal{C}$ based on the updated bids, the allocation phase continues in the same manner as before the re-sorting. Bid update and re-sorting occur as many times as necessary to bind all processes. The algorithm terminates when either all processes have been bound or no more clusters with positive bids exist.

Algorithm 1 gives an outline of this algorithm. $\mathcal{P}[1]$ indicates the top of list $\mathcal{P}$ (contains the largest, not yet bound process) and $\mathcal{C}[k]$ indicates the kth cluster in the list of available (i.e., with positive bids) clusters $\mathcal{C}$, which is sorted based on the clusters' bid values. Once a process is bound to a cluster the process is removed from $\mathcal{P}$ (line 10). Lines 12–16 are executed only if the number of processes is larger than the number of available clusters. Clusters with positive bids are never removed from the list of available clusters allowing, in the case where lines 12–16 are executed, for a cluster to be allocated to more than one process.

The second algorithm, *Dynamic Resorting* (DRS), follows a similar rationale as the NRS algorithm with the exception that a cluster's bid is re-calculated every time a process is bound to the cluster and the list of available clusters in $\mathcal{C}$ is resorted in order to reflect the allocation. Hence the allocation is more dynamic in DRS than in NRS. In contrast to NRS that binds at least one process to each available cluster before considering binding additional processes to a cluster, the DRS algorithm can bind several processes to a cluster, and possibly none to others, based on the dynamically recomputed bids. Algorithm 2 gives an outline of the DRS method. $\mathcal{P}[1]$ and $\mathcal{C}[1]$ indicate the top of list $\mathcal{P}$ (largest, not yet bound process) and the top of list $\mathcal{C}$ (highest bidder an any time), respectively.

The complexity of both algorithms depends on the complexity of the sorting algorithm chosen. Given that we target hardware implementation, the sorting algorithm that can be easily implemented is the insertion sort. The big hardware advantage of the insertion sort is the fact that sorting can happen *in-place*, requiring only constant amount of memory space.

Bidding, in this context, offers several inherent benefits. Bid computation is distributed inside the cores/clusters, eliminating unnecessary traffic. Also, if the clusters cannot respond due to network congestion or them being busy, their bid value is assumed to be zero and, hence, these clusters are excluded from the allocation during the busy intervals. The bidding process is scalable, since an increase in the bidders can easily be integrated by increasing the lists of bids and tasks as well as using more than one allocation units (each managing groups of clusters/cores). As the network size grows, network delay, a more important factor in large networks, is a linear component of the bid. Similarly, core simplicity and core clustering allow for hierarchical multi-level allocation engines, which can take into consideration more detailed intra-cluster conditions.

5.4. Task Allocation in Heterogeneous Manycore Systems. As mentioned previously, we consider a non-clustered architecture in the MPSoC case, however, the algorithm can be applied in clustered MPSoC architectures as well. Additionally, we use the term *task* in the same context we used the term *process*, for the CMP case. In contrast to CMPs, MPSoC applications are bounded by a maximum execution

Algorithm Necessary Re-Sorting()
Inputs: $\mathcal{C} = \{C_1, C_2, \ldots, C_n\}$ // list of clusters
$\mathcal{B} = \{b(C_1), b(C_2), \ldots, b(C_n)\}$ // list of bids, per cluster
n // number of clusters
$\mathcal{P} = \{P_1, P_2, \ldots, P_x\}$ // list of processes
$t(P_i)$, for all $P_i \in \mathcal{P}$ // process size
Output: A bind between all processes and available clusters

01: Sort processes in list $\mathcal{P}$ in decreasing order of $t(P_j)$, $\forall P_j \in \mathcal{P}$
02: Remove clusters with non-positive bid values from $\mathcal{C}$
03: Sort clusters in list $\mathcal{C}$ in decreasing order of $b(C_i) \in \mathcal{B}$, $\forall C_i \in \mathcal{C}$
04: **if** $\mathcal{C} == \emptyset$ **then**
05: **exit** // no (more) allocation possible, system is full
06: **else**
07: $k = 1$
08: **while** $\mathcal{P} \neq \emptyset$ **do**
09: Bind process $P_j = \mathcal{P}[1]$, $P_j \in \mathcal{P}$ to cluster $C_i = \mathcal{C}[k]$, $C_i \in \mathcal{C}$
10: Remove process P_j from $\mathcal{P}$
11: $k = k + 1$
12: **if** $(k > n)$ and $(\mathcal{P} \neq \emptyset)$ **then**
13: **for each** cluster $C_i \in \mathcal{C}$ bound to a process $P_j \in \mathcal{P}$ in step 09
14: $b(C_i) = b(C_i) - t(P_j)$ // re-calculated bid for cluster C_i
15: **goto** step (2)
16: **end if**
17: **end while**
18: **end if**

Algorithm 1: The Necessary Re-Sorting (NRS) Algorithm.

Algorithm Dynamic Re-Sorting()
Inputs: $\mathcal{C} = \{C_1, C_2, \ldots, C_n\}$ // list of clusters
$\mathcal{B} = \{b(C_1), b(C_2), \ldots, b(C_n)\}$ // list of bids, per cluster
n // number of clusters
$\mathcal{P} = \{P_1, P_2, \ldots, P_x\}$ // list of processes
$t(P_i)$, $\forall P_i \in \mathcal{P}$ // process size
Output: A bind between all processes and available clusters

01: Sort processes in list $\mathcal{P}$ in decreasing order of $t(P_j)$, $\forall P_j \in \mathcal{P}$
02: Remove clusters with non-positive bid values from $\mathcal{C}$
03: Sort clusters in list $\mathcal{C}$ in decreasing order of $b(C_i) \in \mathcal{B}$, $\forall C_i \in \mathcal{C}$
04: **if** $\mathcal{C} == \emptyset$ **then**
05: **exit** // no (more) allocation possible, system is full
06: **else**
07: **while** $\mathcal{P} \neq \emptyset$ **do**
08: Bind process $P_j = \mathcal{P}[1]$, $P_j \in \mathcal{P}$ to cluster $C_i = \mathcal{C}[1]$, $C_i \in \mathcal{C}$
09: Remove process P_j from $\mathcal{P}$
10: **if** $\mathcal{P} \neq \emptyset$ **then**
11: $b(C_i) = b(C_i) - t(P_j)$ // re-calculated bid for cluster C_i
12: **if** $b(C_i) \leq 0$ **then**
13: Remove cluster C_i from $\mathcal{C}$
14: **end if**
15: Re-Sort list $\mathcal{C}$ in decreasing order of $b(C_i) \in \mathcal{B}$, $\forall C_i \in \mathcal{C}$
16: **end if**
17: **end while**
18: **end if**

Algorithm 2: The Dynamic Re-Sorting (DRS) Algorithm.

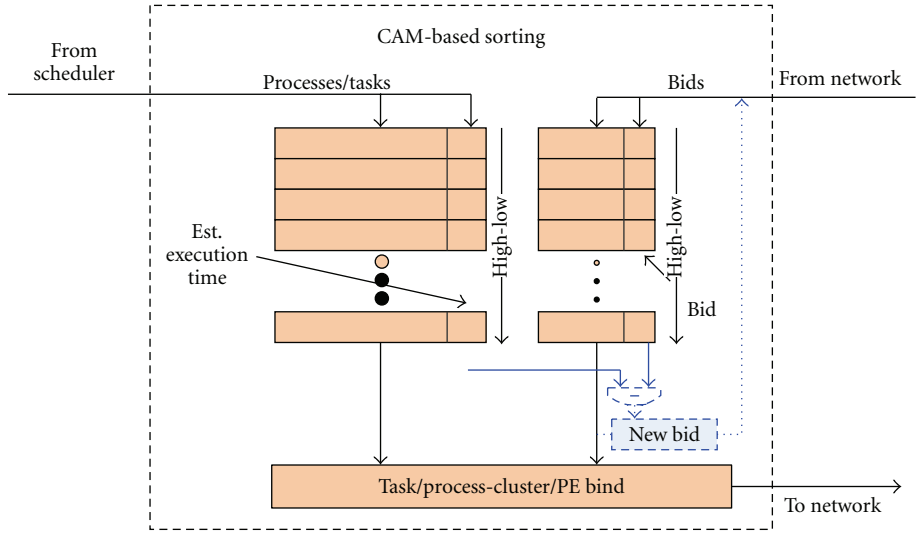

Figure 3: Block diagram of proposed Allocation Engine (AE). Dotted lines indicated hardware necessary for re-sorting. There are two CAMs: one contains bids and cluster/PE ID number, and the other estimated execution times with process/task ID number.

time for real-time response. Moreover, in MPSoCs the processing capacity of a PE (i.e., its ability to execute i tasks during an interval $[T, T+dt]$) can be determined and utilized such that, at a given time, the PE can give a representative value of its available workload capacity in relation to other PEs. A task's estimated execution time is statically derived, by virtue of the task's maximum execution time and the processing capability of the targeted PE. Obviously, the number of tasks already assigned to the targeted PE and the time taken to transfer each task's instructions and data through the on-chip network affect the realistic capability of the targeted PE. These are the real-time situations addressed by the proposed dynamic allocation.

Let an MPSoC consist of k types of PEs (a PE type denotes the functionality of a PE). Multiple cores, say n, of a particular PE type can exist (the number of cores of a particular PE type varies among the various types). In the considered MPSoC architecture, each incoming task can be executed by certain types of PEs, out of the k possible PE types in the system. Since our algorithm does not investigate task assignment based on the type of each PE, but instead based on the available workload, we present the algorithm here under the assumption that each task can be executed by a specific PE (of a certain type), for simplicity purposes. This assumption can actually be trivially relaxed, without any impact on the design and implementation of the proposed bidding algorithm. For some task type i, the list of available resources can include cores of different types, instead of being restricted to the available cores of only one particular type. The proposed allocation scheme does not change, as it will receive bids based on the same criteria and dispatch applications to clusters using the same strategy. Hence, let us assume that the input to the allocation algorithm for the MPSoC case consists of k lists of incoming tasks $(\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_k)$ and k lists of possible PEs $(\mathcal{E}_1, \mathcal{E}_2, \ldots, \mathcal{E}_k)$, in contrast to the CMP case where the input consists of a single list of incoming processes $\mathcal{P}$ and a single list of possible clusters $\mathcal{C}$ (or single cores). All tasks in a list $\mathcal{T}_i$ must be bound to one or more PEs in the corresponding list of PEs $\mathcal{E}_i$ (i.e., the index denotes the task and PE type). The allocation procedure is identical to the one for the generic CMP case, with the difference that the overall process is partitioned into k allocation engines, one per PE and task type. For example, to use the algorithm of Algorithm 2 for a particular task and PE type i, we consider the list of tasks $\mathcal{T}_i = P$ and its corresponding lists of PEs $\mathcal{E}_{i.} = \mathcal{C}$, with $|\mathcal{E}_{i.}| = n$. Time is similarly divided in time intervals of size dt, where the size of each interval controls the granularity of the algorithm.

A PE's bid is calculated based on the same feedback information as in the CMP case: (i) the remaining workload of the PE, and (ii) the estimated network delay for the path from the corresponding allocation engine to the PE. This is done through the use of dedicated hardware inside each PE and the associated router, using hardware counters, just like the general-purpose CMP case. When a PE computes its available workload (in clock cycles) using its processing capacity, it sends a packet to the allocation engine that includes the two values discussed earlier: the workload and a timestamp of the time the packet is sent.

5.5. Hardware Implementation. The proposed allocation engine (AE) was designed using content-addressable memory (CAM), where sorting can be done very fast [46]. The size of the CAM tables can be set relative to the number of processor cores and application demands; however, if more processes per interval are dispatched by the scheduler than the CAM can support, the AE blocks, and dispatches the remaining processes in the next interval. Figure 3 shows the hardware block diagram for the AE; re-sorting can be selectively done (i.e., NRS versus DRS) via the dotted (blue-colored) blocks. Bid packets are collected and assembled inside the network router attached to each cluster/PE. The NoC router is also configured for routing bidding packets in

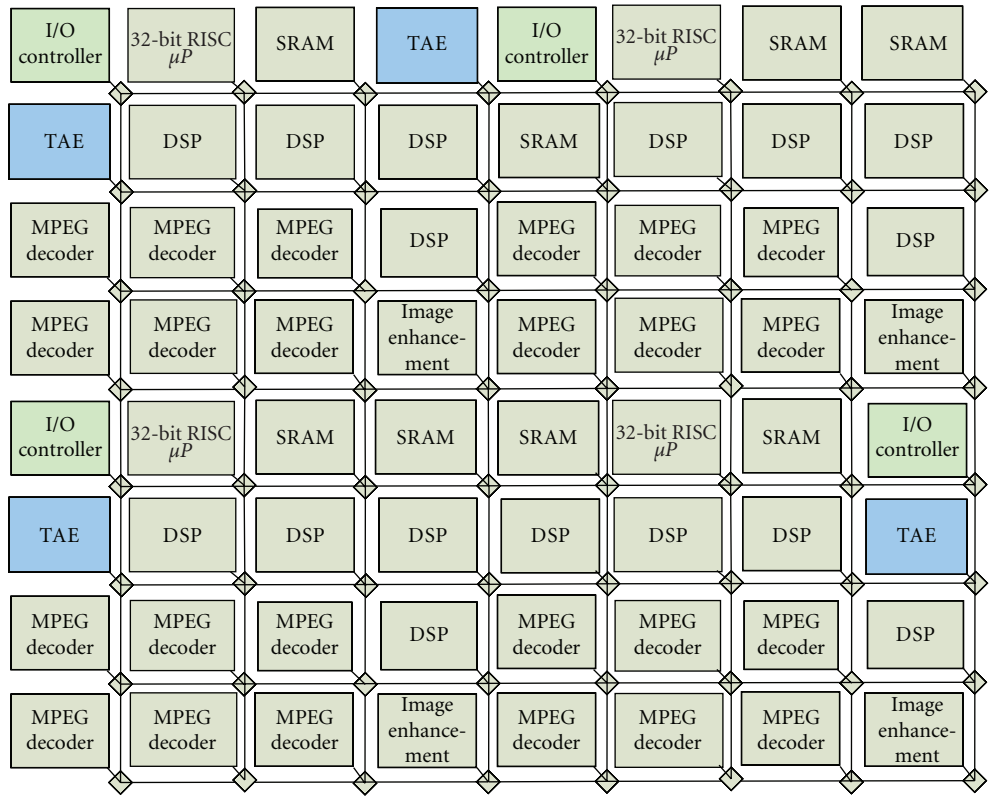

Figure 4: (a) Example Multistream Video Pattern Recognition MPSoC showing 4 I/O controllers and 4 AEs (*toroidal links not shown for readability purposes*) and (b) example MPEG2 Decoding MPSoC showing 4 AEs.

a separate manner as described earlier in Section 5.2. We have also designed an allocation engine based on static, round robin allocation where tasks (or processes) are assigned on each PE (or cluster) using a round robin order, regardless of the PE's status, used as a reference standard engine in our experiments.

The three AEs (Round-Robin (RR), DRS, and NRS) were synthesized using Synopsys Design Compiler and TSMC's 65 nm CMOS standard cell library, with targeted clock frequency of 800 MHz. The RR unit can have a faster cycle, but since the AE is invoked during distinct intervals, its overall delay is amortized. The hardware overhead for the

Table 1: Types of synthetic processes and applications used in CMPs.

SYNTHETIC PROCESSES			
Process Type	Description/Contribution	Number of Threads (Manually Set)	Range of number of Instructions/Thread
Matrix Multiplication	Heavy and Light Computation depending on Matrix size and content (MIPS performs MUL with repeated additions)	Variable, depends on matrix size and values	100's–10000's
Producer/Consumer	Process consists of threads producing data values, which are then consumed by other threads, and vice versa. Gives a varied workload of of ALU/Memory instructions, as well as cache misses.	Variable (Range from 16–64)	100's–1000's
Sequential Memory Access	Fetches a number of variables from memory, performs addition, storing result in memory. Provides a balanced workload in terms of threads, as it can be parallelized, without data dependency between threads. Creates L1 and L2 cache misses.	Max of 64	10's–1000's
Random Memory Access	Similar to sequential access, but with random memory locations. Creates L1 and L2 Cache Misses	Max of 64	10's–1000's
APPLICATIONS			
Process Type	Input Data	Number of Threads (Manually selected)	Range of number of Instructions/Thread
Quicksort 1 QS1	1024 integers	Max of 32	10's–100's
Quicksort 2 QS2	16384 integers	Max 64	100's–1000's
Histogram Equalization HEQ1	Image Sizes of 320 × 240 and 640 × 480 pixels	Max of 16 (320 × 240)	10's–100's
Histogram Equalization HEQ2		Max of 32 (640 × 480)	
Canny Edge Detector CED1	Image Sizes of 320 × 240 and 640 × 480 pixels	Max of 16 (320 × 240)	10's–100's
Canny Edge Detector CED2		Max of 32 (640 × 480)	

DRS and the NRS AE that supports 64 processes/tasks (i.e., CAM size) is very small, ∼2400 and ∼2200 logic gates, respectively. Both DRS and NRS are slightly bigger (approximately 3%) than an RR unit (which also contains memory to keep track and support up to 64 processes/tasks). Blocking was used in all three cases, whenever the number of active processes/tasks was larger than the CAM/memory size.

For a single allocation of 128 tasks to 64 bidders, the DRS AE consumes ∼0.064 mW of total power (including leakage) at 0.8 V power supply voltage, while the NRS AE consumes ∼0.042 mW. The RR AE consumes the least power, at ∼0.02 mW, as it involves simpler computations than NRS and DRS. The overall impact of the bidding packets on the network traffic consists of less than 1% of the total traffic on a 16 × 16 network, so it is anticipated that the proposed dynamic, on-chip-based allocation will not contribute significantly towards the overall energy consumption of the network.

6. Experimental Methodology

With emphasis on modularity, we developed a simulation framework that would allow us to maintain the generic nature of the architecture. Existing high-level system simulation frameworks, such as SIMICS-based simulators [47], may limit the ability in modifying the underlying hardware architecture, especially in including dedicated hardware, besides the very long simulation time necessary for full system-level simulation. Hence, we chose the use of synthesizable SystemC and Verilog to target RTL hardware simulation which allowed us to perform synthesis (using Synopsys CoCentric SystemC Compiler) and derive detailed hardware overhead and power estimates. To make the simulator as generic as possible, we partitioned the manycore system in three major units, the cores/clusters/PEs, the interconnection network, and the I/O controller (in which the AE was also implemented). Given the modularity of the system, we chose to create our framework using an existing NoC simulator [48] and models for each of the units, which we have merged into one system-level simulator using SystemC.

The proposed mechanisms were evaluated using both real-world applications and representative synthetic benchmarks, using various on-chip network sizes and topologies. Specifically, for MPSoC architectures, we experimented with several computational kernels as well as popular MPSoC applications; for general-purpose architectures (CMPs), we used standard applications as well as various synthetic benchmarks, developed specifically for our evaluation

Table 2: Scenarios details for synthetic processes for CMPs.

SCENARIO	Process Types	Processes Per Interval
Scenario 1	15% Heavy Computation (MM), 30% Light Computation (MM), 52% Sequential Memory Accesses, 3% Producer/Consumer	2–32
Scenario 2	8% Heavy Computation (MM), 47% Light Computation (MM), 24% Sequential Memory Accesses, 21% Producer/Consumer	8–64
Scenario 3	22% Heavy Computation (MM), 26% Light Computation (MM), 31% Random Memory Accesses, 21% Producer-Consumer	0–16
Scenario 4	57% Heavy Computation (MM), 31% Random Memory Accesses, 12% Sequential Memory Accesses	0–64
Scenario 5	12% Heavy Computation (MM), 32% Light Computation (MM), 47% Sequential Memory Accesses, 9% Producer/Consumer	8–64
Scenario 6	18% Heavy Computation (MM), 29% Light Computation (MM), 38% Sequential Memory Accesses, 15% Producer/Consumer	16–64

Table 3: Applications used in MPSoC simulations.

Benchmark Name	Description	Benchmark Code
IMAGE PROCESSING MPSoC		
Histogram Equalization	Histogram Equalization on a 320 × 240 image and on a 640 × 480 image	HEQ320
Canny Edge Detector	Uses Canny edge detection algorithm on a 320 × 240 image and on a 640 × 480 image	CED320 CED640
Noise Removal	Uses a gaussian mask for noise removal from a 320 × 240 image and a 640 × 480 image	NR320 NR640
ALL IN PARALLEL	Runs all six applications in parallel (equal priority)	ALL
DSP MPSoC		
Noise Removal	Uses a gaussian mask for noise removal from a 320 × 240 image and a 640 × 480 image	NR320 NR640
Face Detector	Uses a neural network for detecting faces in an image of 320 × 24 and an image of 640 × 480	NNFD320 NNFD640
Low Pass Filter	Performs a low-pass filter operation on a digital signal	LPF
ALL IN PARALLEL	Runs all five applications in parallel (equal priority)	ALL
PAR MPSoC		
Parallel Object Detection	Detects circular shapes using a neural network in 4 parallel video streams of frames 320 × 240 and 640 × 480	OD320 OD640
MPEG2 MPSoC		
MPEG2 Decoding Algorithm	Performs the MPEG2 decoding algorithm on encoded frames	MPEG2-320 MPEG2-640

purposes (see Section 6.3). The latter was necessary since currently there are no publically available standard low-level benchmarks (in Assembly language) for multi/manycore architectures, which would be appropriate for our targeted architecture and simulation framework. As the simulation framework receives binary data, assembly language is easily converted to input vectors for the simulator, eliminating the need for a cross-compiler. Recall that we are targeting the concept of small, parallel threads; research in future manycore software is currently an extremely active area, with the goal of extracting parallelism. As mentioned earlier in Section 4, emerging literature suggests that trends in software development favor the concept of small parallel process and threads [41–44], a concept not readily available in present day low-level software benchmarks. Hence, we created in-house evaluation benchmarks that can be used in conjunction with an MIPS assembler to construct relatively small and parallel processes, each of which can also be broken down into several small-scale threads.

The proposed algorithms in this paper are non-preemptive; if a process is assigned to a cluster/PE, then the process is queued for that cluster/PE until its execution. If the cluster/PE is busy, then the process remains in queue until the cluster/PE completes its current workload.

Also for evaluation purposes, we need a reference point with which to compare the proposed mechanisms. Given the lack of alternative, non-preemptive (without any task/process migration), hardware-based mechanisms for runtime system-level resource allocation in manycore systems, we compare with a Round Robin (RR) allocation algorithm, which was also implemented in hardware. The RR algorithm was selected for the following reasons: (i) it is a static algorithm, that is, it does not receive any system feedback: this will allow us to evaluate the impact in performance, when introducing intelligence during the resource allocation process, (ii) it is very simple and, therefore, very easy to implement in hardware: this will reveal the specific hardware overhead that is imposed to the system by a dynamic method.

Comparing the proposed techniques with existing dynamic algorithms such as the ones in [11, 15, 16, 22, 33–39] would be impractical for several reasons. In contrast to existing state-of-the-art techniques, our algorithm is purely implemented using a custom hardware architecture. The majority of existing methods are implemented at the O/S or runtime system level. Furthermore, state-of-the-art dynamic algorithms [34–38] are assumed to be running on a management processor (i.e., a control processor); hence they can be compiled and simulated using the control processor's architecture. Our algorithm receives real-time, hardware feedback from the on-chip network and the processors/clusters and is designed on custom hardware, controlled

exclusively by the feedback received from the hardware and by minimal O/S interface. The proposed algorithms take into consideration both network congestion as well as processor workload characteristics, in contrast to existing NoC-based dynamic methods which only monitor the network status (but not the PEs' status). This enables indirect integration of all delay mechanisms observed on a chip, such as memory traffic stemming from cache misses as well as other delays encountered due to unpredictable nature of most general-purpose applications. Consequently, the hardware-based cycle-accurate RLT-level simulation framework utilized is the most appropriate for studying the impact of RTL-level modifications done in this work.

The remainder of this section details the experimental methodology, focusing on each of the issues discussed above.

6.1. Experimental Platform Details. The experimental platform was designed based on the generic architectural considerations described in Section 4.

For the homogeneous CMP case, the size of the network and the overall number of cores and clusters were parameterized. The number of cores per cluster was limited; otherwise the bus-based communication will become impractical. Thus, clusters of four and eight cores were considered. We used modified 32-bit MIPS RISC R2000 cores [49], with forwarding hardware but without branch prediction or any other instruction level parallelism optimizations. Each processor consists of a 5-stage pipeline and 1 KB of L1 cache. The processor contains a 16-entry instruction queue, where assigned instructions are stored and executed in FIFO order from the queue. Cached instructions are fetched directly from the L1 and stored into the instruction queue as well. All cores in each cluster share a 16 KB L2 Memory. Cache units contain snooping hardware for coherency as mentioned in Section 3. L1 cache misses are handled internally within the cluster, and L2 misses externally within the network; in each case, the requesting core stalls until the requested data arrive. Two primary NoC topologies, mesh and torus, were used. The network size was varied for each topology, implementing several manycore configurations.

For the heterogeneous experimental platform, we selected generic MPSoC components obtainable from OpenCores [50, 51] and integrated them in the NoC-based SystemC framework. We implemented four different MPSoC architectures, which utilize a variety of on-chip PEs. The first architecture (IP MPSoC) targets image processing applications. The second architecture (DSP MPSoC) targets general DSP applications, and the third architecture (PAR MPSoC) was designed to simultaneously decode multiple video streams and perform neural network-based detection of circular objects. In this case, there are four and eight I/O controllers (we implemented both versions), so each AE receives its input data from four or eight different sources, using round robin time-division multiplexing, with the objective of decoding all streams in real time. Lastly, a custom MPEG2 decoding MPSoC was designed consisting of the typical blocks of the algorithm (Huffman decoding, IDCT, motion compensation and inverse quantization), memory, and embedded microprocessors for control purposes. The PAR MPSoC and MPEG2 architectures are given in Figure 4.

The network parameters were the same in both heterogeneous and homogeneous platforms. Communication was implemented based on dimension-order XY routing, using wormhole switching [4]. The bidding packets were routed in reverse XY priority, as explained in Section 5.2. Packets consist of a 32-bit header flit and four 32-bit payload flits, each containing one instruction, making the packet size 160 bits total, with each packet able to transmit 4 instructions to the destination cluster (or core, for the cases of MPSoCs). Each cluster/core contains a Network Interface (NI) hardware, which decodes the received packets. In the clustered CMP case, the NI also decides how to dispatch the various process threads to the cores inside the cluster. In our current implementation, we used a static round robin allocation within each cluster, primarily due to the small number of cores within each cluster.

6.2. Simulation Methodology. A system simulator was developed, based on an existing cycle-accurate NoC simulator in SystemC [48]. The NoC simulator is able to simulate cycle-accurate variations of the considered mesh and torus topologies and the XY-dimension order routing algorithm at the RTL level. The simulator models the behavior of links (pipelined versus nonpipelined), routers (consisting of a routing decision unit, a crossbar arbitration unit, virtual channel (VC) and associated VC selection unit, and the crossbar itself). Moreover the simulator offers parameterizeable network interface models (NI), which connect to the network infrastructure and allow the user to create user-defined Processing Elements (PEs), using the NI protocols defined in the simulator. This allowed us to create architectures for the simulation of both the CMP as well as the MPSoC cases, by connecting microprocessor cores, clusters, I/O controllers, or special-purpose PEs on the existing NoC simulator. As such, our expanded simulator consists of the on-chip network infrastructure, the NIs and the processor cores and I/O controllers. All were modeled in Synthesizable SystemC. The allocation engines were placed as parts of the I/O controllers. The simulator receives the binary inputs at the I/O controllers, where the AE proceeds to direct them to their bound PEs. Communication between each PE and the NI was supported by hardware handshake signals.

For the CMP case, we constructed a SystemC cluster model, composed of existing SystemC models of the modified 32-bit MIPS R2000 RISC core (without the FP coprocessor and the multiply/divide unit) [49], interconnected using a bus model, also developed in SystemC. We additionally created SystemC models of our proposed AEs. All clusters, as well as the AE, were then plugged in the NoC simulator as NoC processing nodes, with the AE acting as the NoC I/O node, forming a system-level simulation framework that has the ability to operate under a plug-and-play approach, while giving RTL-based simulation. The CMP simulator accepts 32-bit MIPS assembly instructions as inputs, converts them to binary data, packetizes them based on the NoC simulation protocol, and transmits them

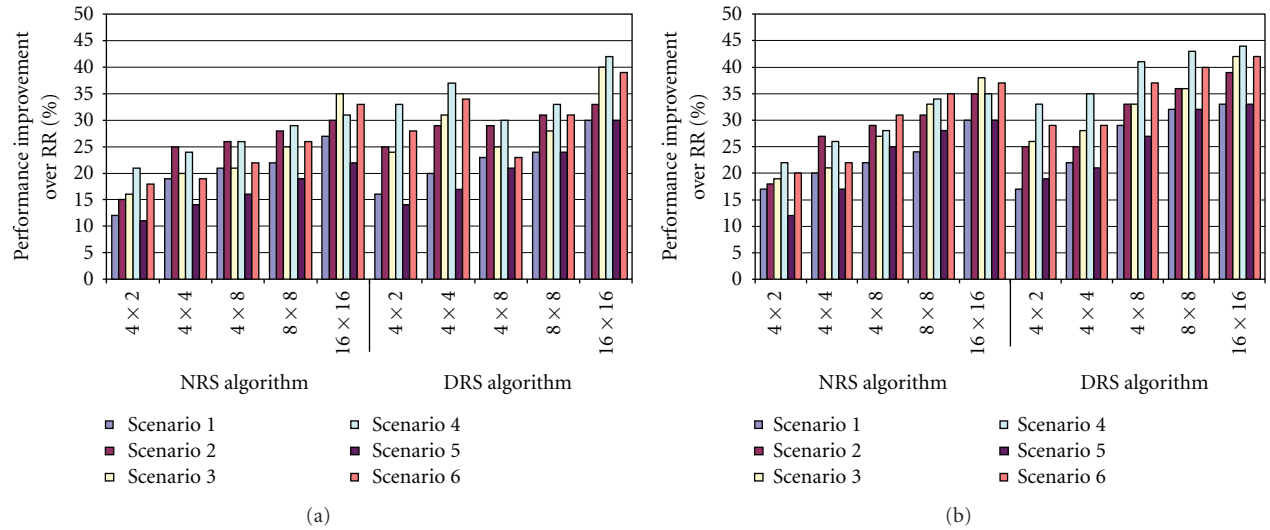

Figure 5: Necessary Re-Sorting (NRS) and Dynamic Re-Sorting (DRS) percentage improvement over Round Robin (RR), for clustered configurations of different network sizes for mesh (a) and torus (b) topologies.

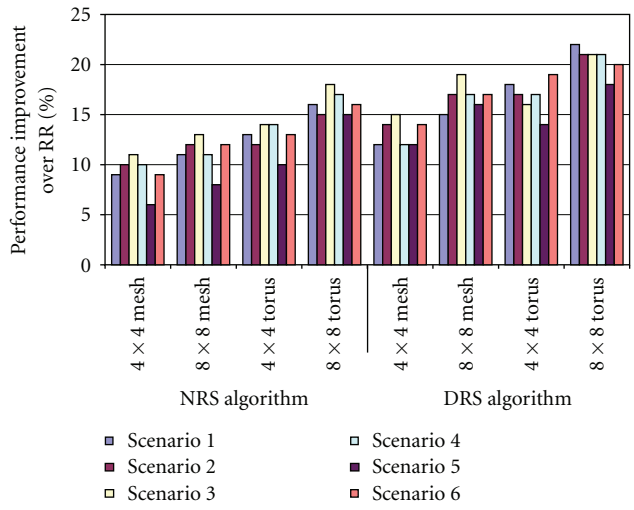

Figure 6: Necessary Re-Sorting (NRS) and Dynamic Re-Sorting (DRS) percentage improvement over Round Robin (RR) for non-clustered configurations of different network sizes and topologies.

through the network to the specific cluster chosen by the AE. The simulator implements data transfer of packets through the network, communication between the cores and the L2 cache, and instruction execution and L1 activity at each MIPS core, giving cycle-accurate behavior. Furthermore, it provides performance metrics such as the total number of cycles taken for a set of instructions to execute and individual core utilization (number of busy cycles over total number of cycles). The simulator can be functionally modeled using Modelsim [52] where debugging and validation of the results and signals were done. Moreover, assertion-based verification using the Open Verification Library was also integrated inside the simulation framework for validation and verification purposes.

A similar approach was followed for the MPSoC case, for each of the four MPSoC architectures described earlier. All implemented MPSoCs were interconnected via a 2D torus NoC, with a number of I/O blocks (controllers) where tasks and associated data enter the chip. Tasks were then directed through the corresponding AE (of the appropriate type), which in turn created the dynamic binds between the tasks and PEs.

6.3. Synthetic and Application-Based Benchmarks. The objective was to create benchmarks suitable for both the CMP and the MPSoC cases, given the differences between the predictability and types of applications. We therefore utilized a set of both synthetic as well as real application benchmarks, for which we were able to control the level of parallelism we induced in our experiments.

In the CMP case, emphasis was placed in creating random workloads consisting of independent processes of random sizes, yielding also random sized threads. Given the impracticality of creating a cross-compiler, we decided to create low-level, MIPS assembly benchmarks from programs that could easily yield the desired parallel processes, while maintaining precedence between processes. We used MIPS assembly instructions to develop four different types of processes. The nature of the chosen processes allows us to vary the number of threads per process type, creating several processes for each considered type, with similar characteristics but different workloads, ranging from tens to tens of thousands of instructions per process. The first part of Table 1 gives a short description of each process type. We implemented a random generator script to generate sets of processes, randomly choosing (i) process type, (ii) number of threads per process, and (iii) number of instructions per thread, to construct a list of processes ranging from 2–256 processes per list. Each list represents the processes scheduled for execution during one time interval. This method gives

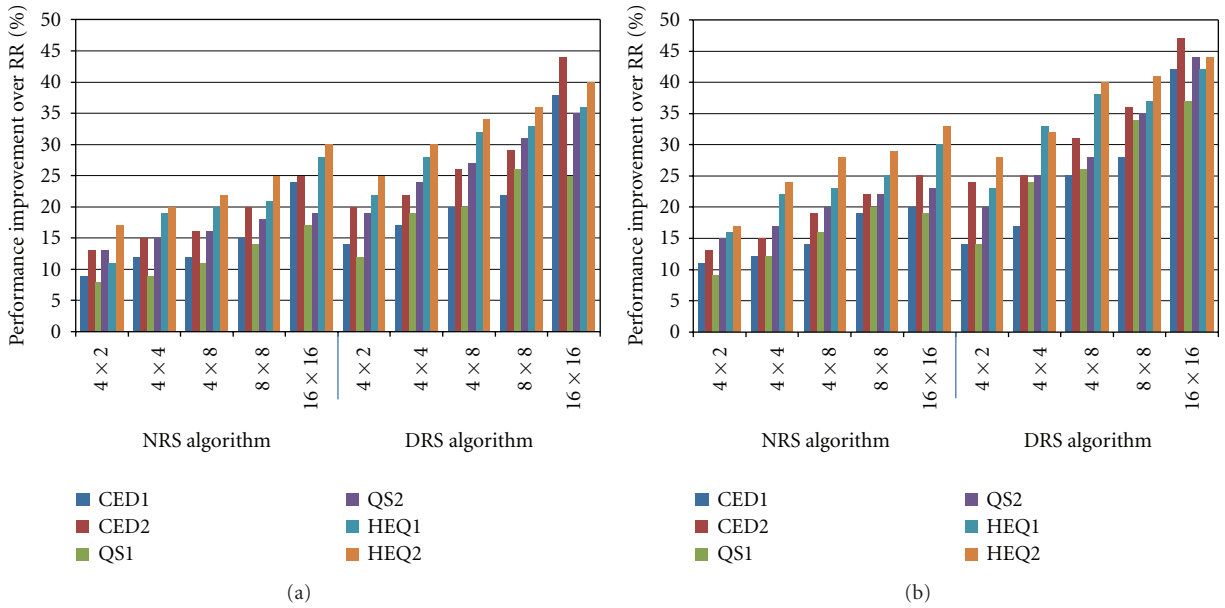

Figure 7: Necessary Re-Sorting (NRS) and Dynamic Re-Sorting (DRS) percentage improvement over Round Robin (RR) for the applications of Table 1, using different mesh network sizes (a) and different torus network sizes (b).

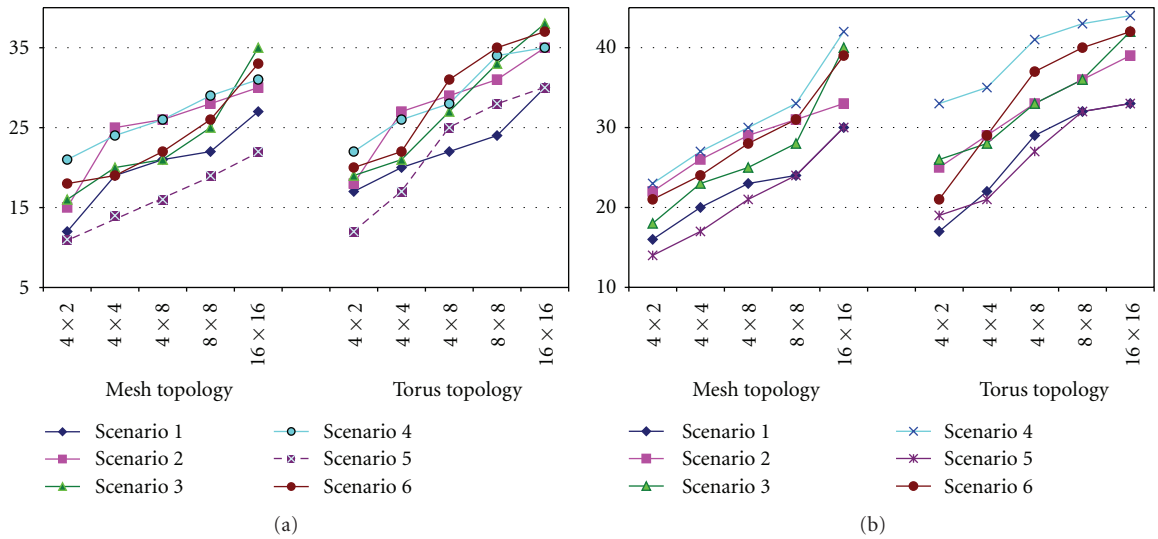

Figure 8: Trends in NRS (a) and DRS (b) improvement over Round Robin (RR) as the network size grows, for mesh and torus topologies.

us a wide range of workloads, consisting of long and short processes, in random order and with random distribution in terms of length. Using these processes as input, we created six different scenarios of execution workloads, which we labeled as Scenario1–Scenario 6. These workloads consist of a variable number of processes for each interval, and each scenario consists of 200 intervals. Details of each scenario are shown in Table 2.

In addition, we chose three real-world applications that could yield parallel processes without the need of an optimized cross-compiler: a canny edge detection algorithm, an implementation of the *quicksort* algorithm, and an image enhancement algorithm based on histogram equalization. These applications, in addition to their inherent parallelism, are capable of presenting us with a varied number of processes and threads per process, due to varied intensity levels on the images, and by varying the input data to the *quicksort* algorithm. Using two data sets for the *quicksort* algorithm and four different image sizes, we created another set of benchmarks shown in the second part of Table 1. In summary, the utilized workloads for the CMPs platform feature different number of allocation intervals, processes per interval, threads per process, and instructions per process.

For the MPSoC case, we selected a variety of MPSoC applications [50, 51, 53] that targeted the four implemented platforms. These applications are summarized in Table 3

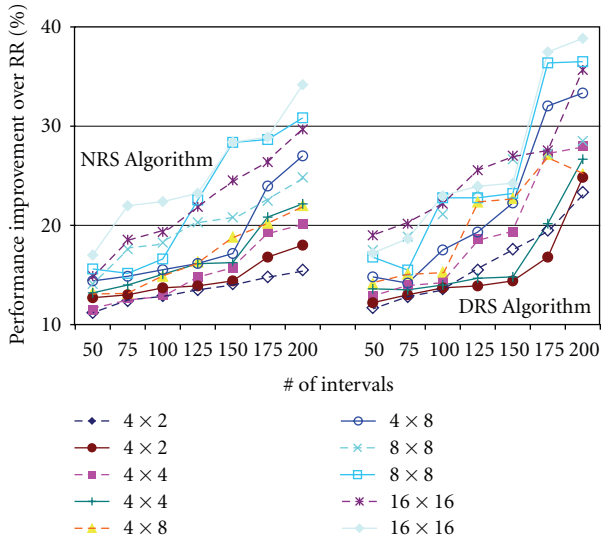

Figure 9: Trends in NRS and DRS improvement over Round Robin (RR) as the workload increases from 50 to 200 intervals.

and feature audio and video (image-based) applications. The image-based applications target images of two sizes; this provides additional workload in each application, giving observations regarding the performance of the algorithm as the number of tasks increases.

Prior to simulating each of the evaluation scenarios, the simulator was initialized for 100,000 cycles, sending random instructions involving random computations and data to each cluster/core/PE, in order to bring the network and the caches into a random system operating condition. At that time, simulation pursues using the benchmark scenarios. For each scenario, we recorded the total number of cycles taken for the system to execute all the workloads and measured the improvement of each of the bidding algorithms over the RR algorithm. We also computed the utilization of each cluster/core/PE, measured by the number of busy cycles over the number of total cycles.

6.4. Allocation Interval. An important parameter in the performance of the bidding algorithm is the allocation interval, traditionally determined by the scheduler, based on the average length of the software processes (i.e., the instruction stream) [7, 8]. In the absence of this information, in our simulations we have set the time allocation interval as follows: (i) for the MPSoC case, the time interval is set appropriately depending on the bound execution time for the smallest task in each application; doing so, ensures that even the smallest tasks will be given fair allocation, a conservative approach but necessary in a time-constrained environment, (ii) for the CMP case, the average estimated execution time of all processes scheduled for a particular interval was used.

In practice, the system scheduler can vary the allocation interval depending on the scheduling policy (i.e., earliest deadline first, rate-monotonic, etc.). In our case, the allocation interval was implemented as a counter; the value can be loaded and adjusted at runtime, through a simple instruction issued by the O/S. The implications of this interval however are important in deciding whether to implement the algorithm in hardware, or explore alternative options such as embedding the allocation algorithm as part of the O/S. Frequent allocation intervals benefit from hardware implementation as feedback is received in real time, and adjustment policies can be determined in real time. In time-constrained environments, especially in applications with hard deadlines, this is very beneficial. If however the allocation happens in sparse, large intervals, then a hardware implementation might not be necessary as the whole operation can be integrated in the software since feedback and adjustment will happen sparsely. In this paper we experiment with the former case, having frequent allocations and small-scale processes with small intervals, as our aim was to investigate the potential benefits of simple but real-time system feedback.

7. Results and Discussion

7.1. Homogeneous Manycore Systems (CMP). We first examine the overall performance impact of each allocation mechanism, by measuring the overall system performance of each algorithm in total clock cycles. Figure 5 shows the obtained results, comparing the two bidding-based algorithms (DRS and NRS) for different network sizes (up to 16 × 16) and topologies, over the RR algorithm. Results are normalized to the RR allocation performance (base of 100%), indicating improvements in the order of 10–38% for NRS and 15–44% for DRS. The DRS algorithm returns the highest improvement in all cases, however the NRS algorithm consumes less power and cycles when running, since it only sorts the bids when there are more processes than bidders (as already discussed in Section 5.5). The extra cycles however are not significant, as over a large period of time the delay is amortized.

A similar trend, although at a lower scale of improvement, is also observed for non-clustered CMPs. Figure 6 shows the six scenarios and the performance improvement for both bidding algorithms, through 4 × 4 and 8 × 8 non-clustered mesh and torus configurations. Figure 7 shows the performance improvement gained for the applications listed in the bottom part of Table 1 for both mesh and torus topologies. Again, we observe a considerable performance improvement for both mechanisms, with DRS giving consistently the best performance.

Another interesting observation obtained from the simulation results concerns the network size; in both bidding algorithms, an increase to the size of the network results in additional performance improvement. This is better illustrated in Figure 8, which shows this increasing trend in all experimental scenarios. This is mainly attributed to two reasons: (i) a larger network offers more routing paths, absorbing input congestion formed during each allocation interval quicker, and (ii) as the network delay increases for cores which are further away from the AE, the algorithm acts as a load balancing mechanism taking this delay into consideration.

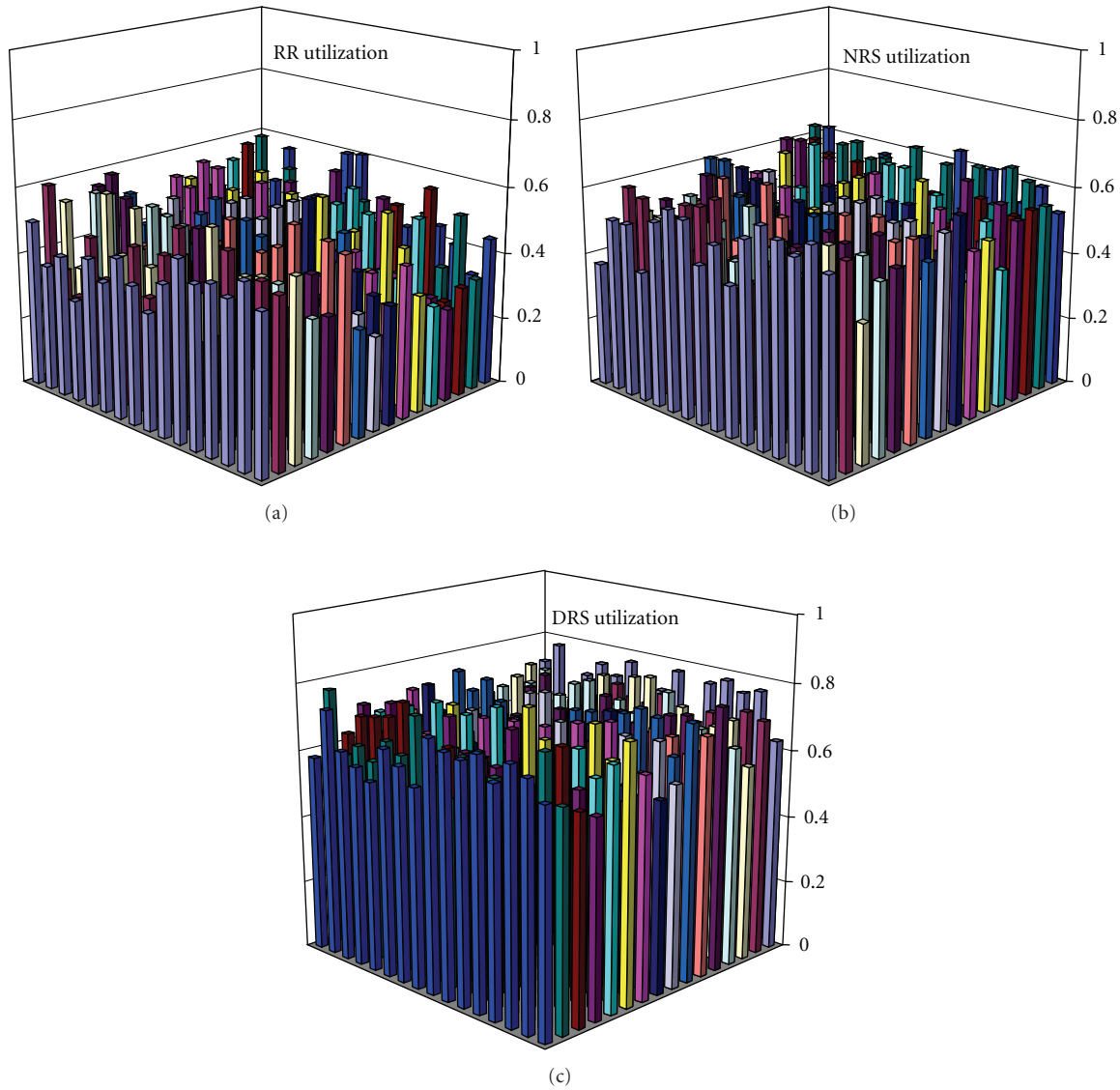

Figure 10: Average cluster utilization when running all benchmark scenarios with RR (a), NRS (b), and DRS (c). The DRS utilization results in a much more balanced distribution of the workload, with a variance between 55%–78%.

Additionally, both DRS and NRS algorithms exhibit increased performance improvement over RR as the number of allocation intervals increases. In Figure 9, we compare various scenarios in terms of increasing number of timing intervals (i.e., increasing overall workload), starting from 50 allocation intervals, and observing the improvement as the amount of allocation intervals increases to 200. As more processes are dispatched to the system, both DRS and NRS algorithms show an increase in the % of improvement over RR. This is encouraging since it shows that as the workload increases, the performance gap between the proposed algorithms and RR algorithm increases significantly.

Load balancing is another important observation. Figure 10 shows the average utilization obtained from all synthetic scenarios (average cluster utilization of Scenario 1–Scenario 6) for the case of a 16×16 clustered network (utilization shown in a 16×16 grid, each grid data point represents the average utilization of each cluster during all six scenarios). Both NRS and DRS give better workload balancing to that of RR, with the DRS algorithm returning the most balanced scenario. This is also observed in other network sizes and configurations. The variance in utilization between clusters decreases; utilization rates between ~27% to ~61% are observed in RR, whereas in the DRS, utilization is distributed more evenly between ~55% and ~78%. This holds true for all network sizes and topologies simulated. Balanced workload distribution across the on-chip grid, offers significant benefits in terms of chip temperature and allocation fairness. Moreover, it improves system performance, as more processor cores are being utilized in parallel with other cores and executing more tasks in less amount of time.

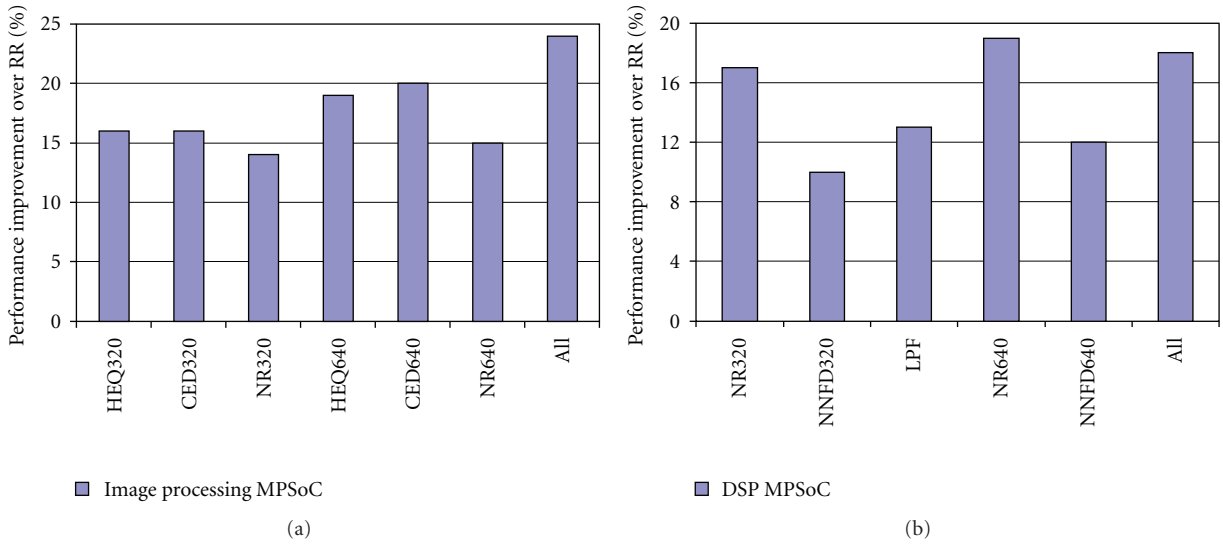

Figure 11: Performance improvement of DRS over RR for two different MPSoC platforms (image processing and DSP platforms).

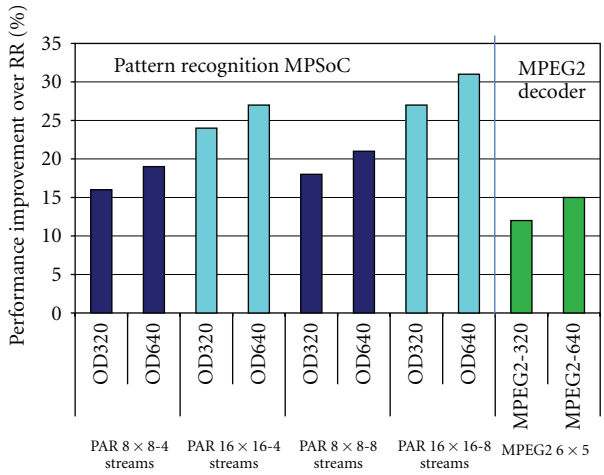

Figure 12: Performance improvement of DRS over RR for the applications running on the pattern recognition MPSoC platform and the MPEG2 Decoding Algorithm.

7.2. Heterogeneous Manycore Systems (MPSoC). As in the CMP case, the DRS algorithm has consistently outperformed the NRS algorithm for MPSoC platforms and, therefore, we only report results for the DRS mechanism in this section.

Figure 11 shows the performance improvement of the DRS allocation engine over the RR engine. In all benchmark cases, the DRS algorithm outperforms RR by a range of 10%–24%. This is especially evident when all applications are run in parallel (labeled as ALL in Figure 11). Additionally, as the number of tasks increases (as shown in the cases where the image size increases yielding more tasks, that is going from NR320 to NR640, from NNFD 320 to NNFD640, and from OD320 to OD640), the improvement of the DRS algorithm increases, indicating better performance as the number of tasks increases (this trend was also observed in the CMP platforms).

Figure 12 shows the results for the pattern recognition MPSoC and the MPEG2 Decoder MPSoC (Figure 4). In the first MPSoC, the improvement of the DRS performance increases as the network size increases, which demonstrates the impact of the encapsulated network delays in the bid. Moreover, the performance increases when handling an increased number of I/O streams, as the number of I/O controllers increases from 4 to 8. For the 16×16 network, the bidding algorithm's improvement increases to 25%–27%, an indication that as the network size increases, the effects of the optimization are clearly more visible, an improvement which also stands when we introduce more I/O streams on the chip. The MPEG2 decoder also observes significant improvements in its performance when using the DRS algorithm, albeit lower improvement when compared to the other applications. This is primarily allocated to the streaming nature of the MPEG2 algorithm. However, as the workload increases, the improvement observed also increases, something that holds true so far throughout our experimental results.

In general, the DRS algorithm outperforms the NRS algorithm; however the NRS algorithm does consume less power and hardware (for details on power estimates and hardware overhead see Section 5.5) and is suitable in cases where the overall number of processes per assignment interval can be known ahead of time to be roughly the same as the overall number of cores.

8. Conclusions and Future Work

As manycore architectures evolve as a computing paradigm, hardware-enabled system-level dynamic algorithms surface as a possible practical approach in addressing system-wide manycore problems such as resource allocation. In this paper, we presented on-chip, bidding-based resource allocation algorithms which improve performance and system utilization (with minimal overhead), demonstrating the benefits

from embedding intelligence on the chip, by utilizing real-time system feedback. Overall the results encourage us to further investigate the impact of system level optimization algorithms.

Future work will investigate the expansion of the proposed hierarchical framework with intra-cluster allocation (thread allocation), incorporating other important parameters in on-chip large-scale multiprocessors, such as energy-aware allocation and interfacing the system to more I/O ports, using an AE per port, in order to increase bandwidth.

References

[1] K. Asanovic et al., "The landscape of parallel computing research: a view Ffrom berkeley," Tech. Rep., University of California at Berkeley, Berkeley, Calif, USA, December 2006.

[2] S. Borkar, "Thousand core chips—a technology perspective," in *Proceedings of the 44th ACM/IEEE Design Automation Conference (DAC '07)*, pp. 746–749, June 2007.

[3] G. De Micheli and L. Benini, *Networks on Chips: Technology and Tools (Systems on Silicon Series)*, Morgan Kauffmann, Boston, Mass, USA, 2006.

[4] W. J. Dally and B. Towles, "Route packets, not wires: on-chip interconnection networks," in *Proceedings of the 38th Design Automation Conference (DAC '01)*, pp. 684–689, June 2001.

[5] R. I. Bahar, D. Hammerstrom, J. Harlow et al., "Architectures for silicon nanoelectronics and beyond," *IEEE Computer*, vol. 40, no. 1, pp. 25–33, 2007.

[6] S. V. Gheorghita, M. Palkovic, J. Hamers et al., "System-scenario-based design of dynamic embedded systems," *ACM Transactions on Design Automation of Electronic Systems*, vol. 14, no. 1, article 3, 2009.

[7] C. C. Price, "Task allocation in distributed systems: a survey of practical strategies," in *Proceedings of the ACM Annual Conference/Annual Meeting*, pp. 176–181, New York, NY, USA, 1982.

[8] V. M. Lo, "Heuristic algorithms for task assignment in distributed systems," *IEEE Transactions on Computers*, vol. 37, no. 11, pp. 1384–1397, 1988.

[9] A. Das and D. Grosu, "Combinatorial auction-based protocols for resource allocation in grids," in *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS '05)*, April 2005.

[10] D. P. Bertsekas, "Auction algorithms for network flow problems: a tutorial introduction," *Computational Optimization and Applications*, vol. 1, no. 1, pp. 7–66, 1992.

[11] R. Jain and P. Varaiya, "Efficient market mechanisms for network resource allocation," in *Proceedings of the 44th IEEE Conference on Decision and Control, and the European Control Conference (CDC-ECC '05)*, pp. 1056–1061, December 2005.

[12] P. Maillé and B. Tuffin, "Multi-bid auctions for bandwidth allocation in communication networks," in *Proceedings of the 23rd Annual Joint Conference of the IEEE Computer and Communications Societies (IEEE INFOCOM '04)*, pp. 56–71, March 2004.

[13] T. Theocharides, M. K. Michael, M. Polycarpou, and A. Dingankar, "A novel system-level on-chip resource allocation method for manycore architectures," in *Proceedings of the IEEE Computer Society Annual Symposium on VLSI Design (ISVLSI '08)*, pp. 99–104, Montpellier, France, 2008.

[14] T. Theocharides, M. K. Michael, M. Polycarpou, and A. Dingankar, "Towards embedded runtime system level optimization for MPSoCs: on-chip task allocation," in *Proceedings of the 19th ACM Great Lakes Symposium on VLSI (GLSVLSI '09)*, pp. 121–124, Boston, Mass, USA, May 2009.

[15] E. W. Brião, D. Barcelos, and F. R. Wagner, "Dynamic task allocation strategies in MPSoC for soft real-time applications," in *Proceedings of the Design, Automation and Test in Europe (DATE '08)*, pp. 1386–1389, March 2008.

[16] X. Liao, W. Jigang, and T. Srikanthan, "Brief announcement: a temperature-aware virtual submesh allocation scheme for NoC-based manycore chips," in *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '08)*, pp. 182–184, Munich, Germany, June 2008.

[17] P. Liu, Y. Kiyoki, and T. Maruda, "Efficient algorithms for resorce allocation In distributed and paralel query processing environments," in *Proceedings of the 9th International Conference on Distributed Computing Systems*, pp. 316–323, June 1989.

[18] A. K. Coskun, T. S. Rosing, and K. Whisnant, "Temperature aware task scheduling in MPSoCs," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE '07)*, pp. 1659–1664, March 2007.

[19] W.-L. Hung, Y. Xie, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin, "Thermal-aware task allocation and scheduling for embedded systems," in *Proceedings of the Design, Automation and Test in Europe (DATE '05)*, pp. 898–899, March 2005.

[20] W.-L. Hung, Y. Xie, N. Vijaykrishnan, C. Addo-Quaye, T. Theocharides, and M. J. Irwin, "Thermal-aware floorplanning using genetic algorithms," in *Proceedings of the Sixth International Symposium on Quality of Electronic Design (ISQED '05)*, pp. 634–639, March 2005.

[21] S. Murali, A. Mutapcic, D. Atienza, R. Gupta, S. Boyd, and G. De Micheli, "Temperature-aware processor frequency assignment for MPSoCs using convex optimization," in *Proceedings of the 5th International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '07)*, pp. 111–116, Salzburg, Austria, October 2007.

[22] P. K. F. Hölzenspies, J. L. Hurink, J. Kuper, and G. J. M. Smit, "Run-time spatial mapping of streaming applications to a heterogeneous multi-processor System-on-Chip (MPSoC)," in *Proceedings of the Design, Automation and Test in Europe (DATE '08)*, pp. 212–217, March 2008.

[23] V. Suhendra, C. Raghavan, and T. Mitra, "Integrated scratch-pad memory optimization and task scheduling for MPSoC architectures," in *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES '06)*, pp. 401–410, October 2006.

[24] M. A. A. Faruque, R. Krist, and J. Henkel, "ADAM: run-time agent-based distributed application mapping for on-chip communication," in *Proceedings of the 45th Design Automation Conference (DAC '08)*, pp. 760–765, June 2008.

[25] A. Kumar, B. Mesman, B. Theelen, H. Corporaal, and H. Yajun, "Resource manager for non-preemptive heterogeneous multiprocessor system-on-chip," in *Proceedings of the IEEE/ACM/IFIP Workshop on Embedded Systems for Real Time Multimedia (ESTIMEDIA '06)*, pp. 33–38, October 2006.

[26] V. Nollet, T. Marescaux, P. Avasare, D. Verkest, and J.-Y. Mignolet, "Centralized run-time resource management in a network-on-chip containing reconfigurable hardware tiles," in *Proceedings of the Design, Automation and Test in Europe (DATE '05)*, pp. 234–239, March 2005.

[27] S. Murali and G. De Micheli, "Bandwidth-constrained mapping of cores onto NoC architectures," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE '04)*, pp. 896–901, February 2004.

[28] D. Pani, G. Passino, and L. Raffo, "Run-time adaptive resources allocation and balancing on nanoprocessor arrays," in *Proceedings of the 8th IEEE Euromicro (CDSD '05)*, pp. 492–499, Porto, Portugal, September 2005.

[29] K. Shaw and W. Dally, "Migration in single chip multiprocessors," *IEEE Computer Architecture Letters*, vol. 1, no. 3, pp. 2–5, 2002.

[30] M. Becchi and P. Crowley, "Dynamic thread assignment on heterogeneous multiprocessor architectures," in *Proceedings of the 3rd Conference on Computing Frontiers (CF '06)*, pp. 29–39, May 2006.

[31] A. C. Nácul, F. Regazzoni, and M. Lajolo, "Hardware scheduling support in SMP architectures," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE '07)*, pp. 642–647, March 2007.

[32] R. Watanabe, M. Kondo, M. Imai, H. Nakamura, and T. Nanya, "Task scheduling under performance constraints for reducing the energy consumption of the GALS multi-processor SoC," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE '07)*, pp. 797–802, March 2007.

[33] C.-L. Chou and R. Marculescu, "User-aware dynamic task allocation in networks-on-chip," in *Proceedings of the Design, Automation and Test in Europe (DATE '08)*, pp. 1232–1237, ACM, New York, NY, USA, March 2008.

[34] L. T. Smit, G. J. M. Smit, J. L. Hurink, H. Broersma, D. Paulusma, and P. T. Wolkotte, "Run-time mapping of applications to a heterogeneous reconfigurable tiled system on chip architecture," in *Proceedings of the IEEE International Conference on Field-Programmable Technology (FPT '04)*, pp. 421–424, December 2004.

[35] M. A. A. Faruque, R. Krist, and J. Henkel, "ADAM: run-time agent-based distributed application mapping for on-chip communication," in *Proceedings of the 45th Design Automation Conference (DAC '08)*, pp. 760–765, ACM, New York, NY, USA, June 2008.

[36] A. Mehran, A. Khademzadeh, and S. Saeidi, "DSM: a heuristic dynamic spiral mapping algorithm for network on chip," *IEICE Electronics Express*, vol. 5, no. 13, pp. 464–471, 2008.

[37] E. Carvalho and F. Moraes, "Congestion-aware task mapping in heterogeneous MPSoCs," in *Proceedings of the International Symposium on System-on-Chip Proceedings (SOC '08)*, pp. 1–4, 2008.

[38] A. K. Singh, W. Jigang, A. Prakash, and T. Srikanthan, "Efficient heuristics for minimizing communication overhead in noc-based heterogeneousmpsoc platforms," in *Proceedings of the IEEE/IFIP International Symposium on Rapid System Prototyping*, pp. 55–60, IEEE Computer Society, 2009, Washington, DC, USA.

[39] A. K. Singh, W. Jigang, A. Kumar, and T. Srikanthan, "Run-time mapping of multiple communicating tasks on MPSoC platforms," in *Proceedings of the International Conference on Computational Science (ICCS '10)*, Elsevier, 2010.

[40] H. Lin, Y. Feng, and X. Qiang, "Lifetime reliability-aware task allocation and scheduling for MPSoC platforms," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE '09)*, pp. 51–56, Nice, France, April 2009.

[41] M. Cintra, J. F. Martinez, and J. Torrellas, "Architectural support for scalable speculative parallelization in shared-memory multiprocessors," in *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA '00)*, pp. 13–24, June 2000.

[42] J. F. Martínez and J. Torrellas, "Speculative synchronization: applying thread-level speculation to explicitly parallel applications," in *Proceedings of the10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '02)*, pp. 18–29, San Jose, Calif, USA, October 2002.

[43] K. Asanovic, R. Bodik, J. Demmel et al., "A view of the parallel computing landscape," *Communications of the ACM*, vol. 52, no. 10, pp. 56–67, 2009.

[44] M. K. Prabhu and K. Olukotun, "Using thread-level speculation to simplify manual parallelization," in *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP '03)*, pp. 1–12, San Diego, Calif, USA, 2003.

[45] J. Leverich, H. Arakida, A. Solomatnikov, A. Firoozshahian, M. Horowitz, and C. Kozyrakis, "Comparative evaluation of memory models for chip multiprocessors," *Transactions on Architecture and Code Optimization*, vol. 5, no. 3, article 12, 2008.

[46] L. Chisvin and R. J. Duckworth, "Content-addressable and associative memory: alternatives to the ubiquitous RAM," *IEEE Computer*, vol. 22, no. 7, pp. 51–64, 1989.

[47] [33] The Wisconsin Multifacet Gems Simulator, June 2009, http://www.cs.wisc.edu/gems/.

[48] The *NIRGAM NoC Simulator*, http://www.nirgam.ecs.soton.ac.uk.

[49] A. Vad Lorentzen and N. A. Jørgensen, "SystemC MIPS R2000 Core," http://www2.imm.dtu.dk/SoC-Mobinet/elements/mips_core.htm.

[50] Open Cores, September 2008, http://www.opencores.org.

[51] N. Nicolici et al., Verilog MPEG2 Decoder, August 2010, http://www.ece.mcmaster.ca/~nicola/mpeg.html.

[52] Modelsim HDL Simulator, Mentor Graphics, June 2009, http://www.model.com.

[53] Intel Integrated Performance Primitives, September 2008, http://www.intel.com.