DOI: 10.14529/jsfi140102

Runtime-Aware Architectures: A First Approach

Mateo Valero [1,2]***, Miquel Moreto*** [1]***, Marc Casas*** [1]***, Eduard Ayguade*** [1,2]***, Jesus Labarta*** [1,2]

In the last few years, the traditional ways to keep the increase of hardware performance at the rate predicted by Moore's Law have vanished. When uni-cores were the norm, hardware design was decoupled from the software stack thanks to a well defined Instruction Set Architecture (ISA). This simple interface allowed developing applications without worrying too much about the underlying hardware, while hardware designers were able to aggressively exploit instruction-level parallelism (ILP) in superscalar processors. With the irruption of multi-cores and parallel applications, this simple interface started to leak. As a consequence, the role of decoupling again applications from the hardware was moved to the runtime system. Efficiently using the underlying hardware from this runtime without exposing its complexities to the application has been the target of very active and prolific research in the last years.

Current multi-cores are designed as simple symmetric multiprocessors (SMP) on a chip. However, we believe that this is not enough to overcome all the problems that multi-cores already have to face. It is our position that the runtime has to drive the design of future multi-cores to overcome the restrictions in terms of power, memory, programmability and resilience that multi-cores have. In this paper, we introduce a first approach towards a Runtime-Aware Architecture (RAA), a massively parallel architecture designed from the runtime's perspective.

Keywords: Parallel architectures, runtime system, hardware-software co-design.

Introduction

When uniprocessors were the norm, Instruction Level Parallelism (ILP) and Data Level Parallelism (DLP) were widely exploited to increase the number of instructions executed per cycle. The main hardware designs that were used to exploit ILP were superscalar and Very Long Instruction Word (VLIW) processors. The VLIW approach requires to statically determine dependencies between instructions and schedule them. However, since it is not possible in general to obtain optimal schedulings at compile time, VLIW does not fully exploit the potential ILP that many workloads have. Superscalar designs try to overcome the increasing memory latencies, the so called *Memory Wall* [42], by using Out of Order (OoO) and speculative executions [18]. Additionally, techniques such as prefetching, to start fetching data from the memory ahead of time, deep memory hierarchies, to exploit the locality that many programs have, and large reorder buffers, to increase the number of speculative instructions exposed to the hardware, have been also used to enhance superscalar processors performance. DLP is typically expressed explicitly at the software layer and it consisted in a parallel operation on multiple data performed by multiple independent instructions, or by multiple independent threads. In uniprocessors, the Instruction Set Architecture (ISA) was in charge of decoupling the application, written in a high-level programming language, and the hardware, as we can see in the left hand side of Figure 1. In this context, the architecture improvements were applied at the pipeline level without changing the ISA.

Some years ago, the traditional ways to keep increasing hardware performance at the rate predicted by Moore's Law vanished, additionally to the memory wall. The processor clock frequency stagnated because, when it increased beyond a threshold, the power per unit of area (power density) could not be dissipated. That problem was called the *Power Wall* [27]. A study

[1]Barcelona Supercomputing Center, Carrer Jordi Girona 29, 08034 Barcelona, Spain
[2]Universitat Politecnica de Catalunya, Carrer Jordi Girona 1-3, 08034 Barcelona, Spain

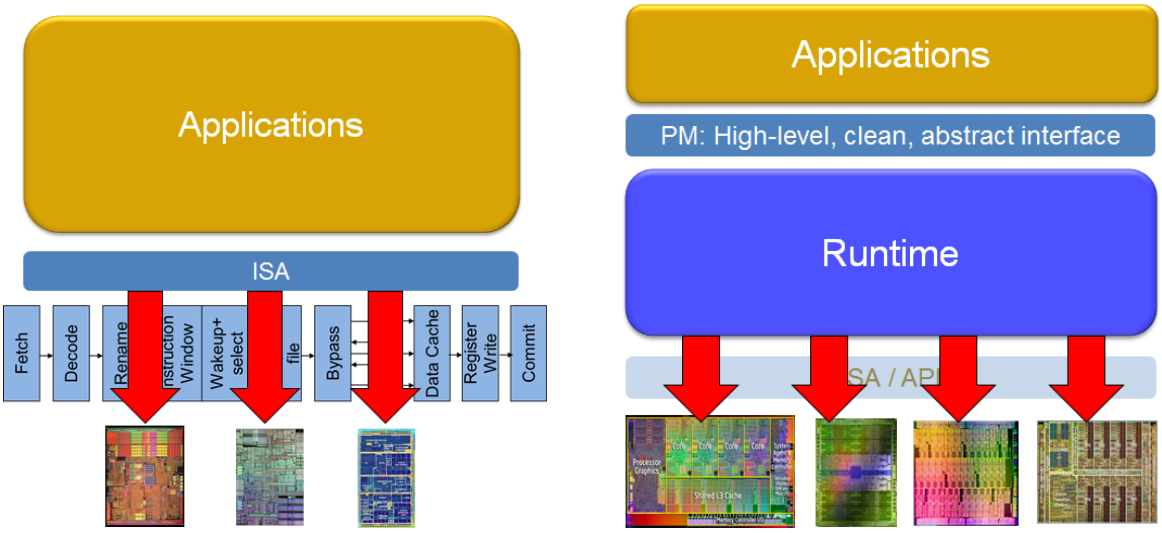

Figure 1. Left: Decoupling the hardware and the software layers in uniprocessors. Right: The runtime drives the hardware design in multiprocessors. We call this approach a Runtime-Aware Architecture (RAA) design.

made by the International Technology Roadmap for Semiconductors predicts an annual frequency increase of 5% for the next 15 years [19]. That means that we are left with parallelism alone in order to further increase performance.

To overcome the stagnation of the processor clock frequency, vendors started to release multi-core devices over a decade ago. They can potentially provide the desired performance gains by exploiting Task Level Parallelism (TLP). However, multi-core designs, rather than fixing the problems associated with the memory and power walls, exacerbate them. The ratio cache storage / operation stagnates or decreases in multi-core designs as well as the memory bandwidth per operation does, making it very hard to fully exploit the throughput that multi-core designs have. Another major concern is energy consumption, since if it keeps growing with the same rate as today, some major technological challenges like designing exascale supercomputers or developing petaflop mobiles will become chimeras. This set of challenges related to power consumption issues constitutes a new power wall.

Additionally, multi-core systems might have a heterogeneous set of processors with a different ISA, connected through several layers of shared resources with variable access latencies and distributed memory regions. To manage data motion among this deep and heterogeneous memory hierarchy while properly handling Non-Uniform Memory Access (NUMA) effects and respecting stringent power budget in data movements is going to be a major challenge in future multi-core machines. All these problems regarding programmability and data management across the memory hierarchy are commonly referred as the *Programmability Wall* [9].

Multi-core architectures can theoretically achieve significant performance with low voltages and frequencies. However, as the voltage supply scales relative to the transistor threshold voltage, the sensitivity of circuit delays to transistor parameter variations increases remarkably, which implies that processor faults will become more frequent in future designs. Additionally, the fact that the total number of cores in future designs will increase in several orders of magnitude only makes the fault prevalence problem more dramatic. In addition to the current challenges in parallelism, memory and power management, we are moving towards a *Reliability Wall* [43].

With the irruption of multi-cores and parallel applications, the simple interface between the hardware and the application started to leak. As a consequence, the role of decoupling again applications from the hardware was moved to the runtime system. This runtime layer is also in charge of efficiently using the underlying hardware without exposing its complexities to the application. In fact, the collaboration between the heterogeneous parallel hardware and the runtime layer becomes the only way to keep the programmability hardship that we are anticipating within acceptable levels while dealing with the memory, power and resilience walls.

Current multi-cores are designed as simple symmetric multiprocessors (SMP) on a chip. However, we believe that this is not enough to overcome all the problems that multi-cores already have to face. To properly take advantage of their potential, an enhanced hardware-software collaboration is required. It is our position that the runtime has to drive the design of future multi-cores to overcome the challenges of the above mentioned *walls*. We envision a Runtime-Aware Architecture (RAA), a holistic approach where the parallel architecture is partially implemented as a software runtime management layer, and the remainder in hardware. In this architecture, TLP and DLP are managed by the runtime and are transparent to the programmer. The idea is to have a task-based representation of parallel programs and handle the tasks in the same way as superscalar processors manage ILP, since tasks have data dependencies between them and a data dependency graph can be built at runtime or statically. As such, the runtime would drive the design of new architecture components to support its activity. In the right hand side of Figure 1 we can see a representation of this idea, where the application is implemented by using a high-level programming model that decouples it from the runtime and the hardware. The runtime not only uses the hardware efficiently, but also drives its design. As such, specific hardware components that support the runtime activities are a key point of the RAA approach.

Under the experience of the current ending age and equipped with a mature vision of what a productive future can be, many good ideas that disappeared during the RISC clock frequency boom of the 80's and 90's can be reshaped and applied with unforeseen scales or scope, resulting in an innovative vision of how to address the current embroilment where hardware technology has taken us.

Our approach towards parallel architectures offers a single solution that could alleviate most of the problems we encounter in the current approaches: handling parallelism, the memory wall, the power wall, the programmability wall, and the upcoming reliability wall in a wide range of application domains from mobile up to supercomputers. Altogether, this novel approach towards future parallel architectures is the way to ensure continued performance improvements, getting us out of the technological hardship that computers have turned into, once more riding on Moore's Law.

In Section 1 we describe more in detail how a task-based runtime manages the workload and how some ideas that are exploited by superscalar processors may be exploited by the runtime. In Section 2 we explain how the superscalar runtime has been used to efficiently exploit multi-cores. In Section 3 the concept of runtime-aware architectures is explained in detail. Section 4 talks about some related work. Finally, in Section 5 we comment the conclusions of this work.

1. Bringing the Superscalar Vision to the Runtime Level

We plan to use a runtime system that uses a task-based abstraction in which the programmer specifies which are the input and output arguments of the different tasks, which are going to

Table 1. Comparing superscalar and runtime visions

Superscalar	Task-based Runtime
Instructions	Tasks
Functional Units	Cores
Fetch and Decode Units	Cores
Registers (name space)	Main Memory
Registers (storage)	Local Memory
Out-of-Order Execution	
Pipelined Execution	
Speculative Execution	

be managed in the same way as superscalar processors manage instructions. This dataflow information allows the runtime to dynamically build and maintain a Task-Dependency Graph (TDG), which constitutes the foundation of a tight collaboration with the hardware to drive scheduling decisions and to discover opportunities to manage data movement in the architecture.

More precisely, as we can see in Table 1, the runtime layer conceives the different tasks of a parallel application as if they were instructions in a superscalar processor. Similarly, the fetch, decode and functional units in a pipeline can be seen as the cores of the heterogeneous many-core hardware, while registers can be foreseen as the local and main memories of parallel architectures. As such, concepts like Out-of-Order, pipelined, and speculative execution appear naturally at the runtime level in terms of task-based parallel applications. The task-dependency graph allows the runtime to execute independent tasks Out-of-Order, as instructions in traditional superscalar architectures. Deep pipelines, which are typically a component of superscalar processors, also appear at the many-core level in terms of sequences of tasks that can be overlapped to achieve more performance. In future many-core systems with hundreds of cores, idle cores can be used to speculatively execute tasks to accelerate application progress and prefetch data into the chip.

Since our approach consists in processing tasks in the same way as superscalar architectures handle dynamic instructions, many approaches that have been typically applied at the instruction pipeline level can inspire runtime optimizations and new hardware designs. Even more, as the task-dependency graph is much more complex and has much more parallelism to exploit than instruction dependencies, significant optimizations in terms of performance, power or resilience can be achieved by exposing the task-dependency graph to the available hardware and then balance the workload accordingly. Also, new architectural components to support the runtime activity are expected to play a key role.

In particular, we plan to implement our approach in the top of the OpenMP Superscalar (OmpSs) [11] runtime layer, which represents the state of the art in task-based runtime layers. OmpSs is an embedding of StarSs [29] in OpenMP.

2. Efficiently Exploiting Parallel Architectures from the Runtime

The task abstraction and the management of parallelism from the runtime system represented a major breakthrough in parallel programming. Exposing the dataflow across tasks, enables the runtime system to efficiently operate the parallel hardware in the same way the

superscalar processors manage the functional units. This opens the door to a vast amount of complex optimizations that the runtime system and the underlying architecture can perform in a transparent way, providing sustained performance improvements across new hardware generations. For the rest of the section, we comment several techniques that the runtime can apply to overcome each one of the above mentioned walls.

Memory Wall The runtime scheduler can detect and exploit temporal **data reuse** by reordering tasks that reuse the same input data, or that use the outputs of the previous tasks. Also, it can make decisions about data distribution, allocating data close to where the task will be executed, prefetching data ahead of time, and creating explicit copies for increased locality if the same data is required in multiple locations at the same time. Bellens et al. [3] show the potential utility of these techniques in the Cell Broadband Engine microarchitecture, which has eight accelerators, each with a 256KB local memory, and a PowerPC processor. The runtime takes care of the task scheduling and data handling between the different processors of this heterogeneous architecture by using a locality-aware mechanism to reduce the overhead of data transfers from the PowerPC to the accelerators.

Minimizing **data movement** in the memory hierarchy is indeed a key technique to deal with the memory wall, as it reduces the number of accesses to the main memory and exploits synergies between the different components of the memory system. Some initial results have been already obtained in CPU+GPU systems [7]. The runtime system moves the data as needed between the different nodes and GPUs minimizing the impact of communication by using affinity scheduling, caching, and by overlapping communication with the computational task. When a GPU kernel is launched, the GPU threads request a new task to the scheduler. Then, the transfer of any data that might be needed by the prefetched task is initiated. In this way, by the time this task can be executed the data will already be available. This prefetch is more effective when combined with overlapped computation and data transfers.

To allow processors exploiting more ILP, register **renaming** has been exploited in superscalar architectures. Such approaches allow having more physical registers than logical registers, avoiding serialization penalties due to registers reuse. They require keeping track of dependencies between instructions' operands to determine if a new renaming register can be assigned to an architecture register. Renaming can help removing anti-dependencies between instructions, which can also be applied to tasks [4]. Renaming can be applied at task instantiation time, or delayed until just before task execution, similarly to virtual registers [15], which can delay the allocation of physical registers until a late stage in the pipeline, instead of doing it in the decode stage. Since the task-dependency graph can be generated ahead of time, it is possible to delay memory allocation to tasks until they start executing. This **virtual resource allocation** allows other tasks in the critical path to take advantage of this extra memory.

Power Wall The potential of exploiting the **critical path** has been already evaluated in MPI programs [6]. By using the knowledge of the critical path and combining it with straightforward profile-based techniques, a set of compact performance indicators that describe important performance-related questions, such as load imbalance, resource consumption or dynamic workload, an efficient scheduling can be derived. These performance indicators can be used by the runtime system to efficiently manage the load, trade performance for power or vice versa, and overall, optimize and adapt runtime decisions to the users' needs. Improvements up to 18% in

execution time and 21% in energy efficiency have been achieved by using dynamic performance metrics to adjust runtime decisions [37].

Exploiting **task specialization** can give significant improvements since different tasks can be scheduled and mapped to different hardware components to deliver the maximum performance while spending low power. For example, by using memory-specific performance information (i. e. cache capacity and memory bandwidth required), the different tasks can be mapped to hardware components with the required memory resources or, alternatively, if a task was already running, it would be possible to switch-off non-required pieces of the memory. That would provide very important improvements in terms of power consumption.

Also, the programmer or the compiler can provide, for the same task, different versions of code targeting several accelerators and, according to the hardware state, the runtime can choose which version of the code must be used [30]. As such, if the machine has some cores with support for vector instructions, the runtime can reduce power consumption be scheduling the appropriate tasks to them to exploit their reduced fetch and decode power consumption.

Reliability Wall A wide range of techniques can be applied at the runtime level to increase applications resilience and deal with the reliability wall. If we assume that the tasks are idempotent, we can take check-points of the tasks' inputs and re-execute them if some fault takes place [38]. To reduce the memory overhead, we can apply a smart copy mechanism that takes just one checkpoint per input, even if it is used by multiple tasks.

Alternatively, it is also possible to extend the programming model to enable the user to specify pieces of code that are particularly sensitive from the resilience perspective by using special pragma annotations. That would give to the runtime the information on what tasks should be mapped into more resilient hardware components.

Another possible improvement that task-based runtimes allow is to tolerate the latency that recomputation techniques induce by overlapping computation with recovery techniques and let the execution progress if the faulty task is not in the critical path. That is an excellent framework to deploy algorithmic recoveries as its cost can be tolerated by overlapping them with standard runtime tasks.

Programmability Wall The runtime can handle **data dependencies** allowing the programmer to deliver straightforward code where just the input and output parameters of the different tasks are specified, which notably simplifies the work of efficiently programming heterogeneous many-core architectures. As such, the task-dependency graph is used by the runtime to expose the parallel workload to the available hardware in a transparent way from the programmer point of view, in the sense that the application source code does not contain information on how to handle the workload besides specifying the input/output parameters.

As such, the programming hardships observed in heterogeneous many-core architectures such as the Cell processor [41] can be avoided by considering the runtime management as a part of the hardware that efficiently manages the load without the need of explicitly exposing the problem to the programmers. As a consequence, the runtime takes care of balancing the load among different cores and is able to assign more tasks to faster cores. The same runtime layer can easily adapt to programs with multiple implementations of the same task type (i.e. tuned for different accelerators or cores), and decide in which execution unit the task has to be run.

A task-based approach can potentially reduce the **synchronization costs** that shared memory approaches typically have and thus achieve significant performance improvements when

they manage highly parallel workloads on many-cores chips. As they allow the programmer to specify program parts called tasks, which can be executed concurrently, the mapping of tasks to threads is done dynamically by a runtime environment without any specific programmer responsibility in the way that synchronization costs are reduced.

3. Runtime-Aware Architectures

Our envisioned Runtime-Aware Architecture (RAA) constitutes a new paradigm of parallel computing systems in which the runtime management layer drives the hardware design, and they both collaborate to leverage an unprecedented degree of parallelism on-chip and exploit information on data dependencies. In this section, we describe in detail the different aspects of such hardware-software collaboration and we discuss some new opportunities brought by it. We expose the several aspects of the RAA approach by describing several techniques to deal with the memory, power, resilience and programmability walls and explaining how the hardware-software collaboration can exploit them.

Memory Wall Increasing **data reuse** in superscalar architectures was crucial to reduce the impact of the memory wall. In RAA's, there is a great margin for performance and energy efficiency improvements in the development of scheduling strategies that exploit this knowledge about the future, encoded in the task-dependency graph. RAA's offer the possibility to extend these ideas with specific hardware support that automatically handle these data transfers with the help of the runtime system by relying on software coherency or on specific architectural support to propagate updates across the multiple copies of data.

Prefetching had an extremely important role in fighting the memory wall in superscalar architectures. In RAA's, we can combine the capabilities of the runtime scheduler to exploit information about future data transfers, with the right kind of hardware support to optimize such transfers. The required hardware support is asynchronous data movement operations, that enable the scheduler to overlap data transfers required for future tasks with the execution of the current task, which reduces the bandwidth requirements and tolerates memory latencies as the transfer is out of the critical path [25]. The design requirements for this new memory system differ significantly from today's designs where memory transfers are always in the critical path.

Since the runtime system can decide in advance which tasks will execute in a given core after the current executing task finalizes, it can also determine the data required by the upcoming task and prefetch this data to the desired level of the cache hierarchy using locking and flushing mechanisms developed with this objective [14, 28]. This technique has a lot of potential, but if applied too aggressively or too early, can evict active data of the current executing task, negatively impacting the performance of the application. As an alternative, special purpose buffers can be added to the architecture to store the inputs of future tasks without affecting the contents of the cache hierarchy. Also, producer-consumer data can be efficiently forwarded with the adequate hardware support [23].

Minimizing on-chip and off-chip **data movements** is very important in terms of final performance and power consumption. Exploiting locality via reuse or prefetching is not going to be enough in future many-core systems. Simplified coherence protocols guided by the runtime system can be used to reduce coherence traffic [24]. An interesting complement consists in upgraded movement primitives that perform user defined transformations on the data as it is being transferred. In-memory functional units for integer, floating point or vector operations have been

proposed in the past, but these transformations can also be done in the network routers. Also, more advanced Network-on-Chip (NoC) topologies that can dynamically adapt data movements depending on the network contention can help in reducing NoC's contention and communication latencies. In general, criticality-aware communications will be essential in our envisioned RAA.

We can also envision strategies that assign more or faster resources to tasks that are in the **critical path** (or likely to be) of the parallel application. In a many-core system, scheduling critical tasks to faster cores while minimizing resource contention and data motion is of paramount importance and requires a tight collaboration between the architecture and the runtime system. Since hybrid NoC's and memories will be a reality in such systems, developing the adequate prioritization mechanisms for communications and memory accesses of critical tasks is a key point to reduce overall execution time of the parallel application.

Power Wall The critical path is one of the most important dynamic properties of a parallel program. As it is the global execution path of tasks that forces wait operations on other tasks without itself being stalled, the events that are not included on it are, up to certain extend, **latency tolerant**, as they can take more time to complete without hurting the performance of the whole application. That opens a wide range of potential improvements in terms of power consumption or performance: If a given memory access is not in the critical path, it can be performed in a long-latency and thus low power region of the memory. Similarly, tasks that are not in the critical path can be mapped to slower and thus low power hardware components of the many-core system. Alternatively, if a particular task is in the critical path, it can be mapped to faster hardware components as its early completion may significantly reduce the waiting time of other tasks and thus reduce overall energy consumption.

In superscalar processors, specialized functional units can improve performance and achieve low power consumption rates for different phases of programs' execution. **Heterogeneous designs** can accelerate many applications that combine compute-intensive and control-intensive phases of computation, which should ideally be handled by different processing elements. These codes include large-scale scientific computations, complex simulations of physical phenomena, complex visualizations or financial markets' predictions. While systems with heterogeneous functional units can significantly improve performance and power consumption, they require to properly partition the application's code across all the available functional units or to efficiently manage data motion between functional units. Thus, the runtime system can become an excellent management layer to efficiently use hardware with heterogeneous cores and accelerators without making the programmability harder. As the task-dependency graph can be generated ahead of tasks' execution, the management of heterogeneous functional units can be properly planned. Additionally, some historical information on the tasks that have been already executed in terms of performance or resource usage can be kept by the runtime. By combining this historical data with the task-dependency graph, efficient scheduling decisions can be made by the runtime.

Reliability Wall Reducing **error propagation** is crucial to increase applications' resilience and deal with the reliability wall. Since soft faults that take place during a task execution will impact its outputs and all the tasks that use these as input parameters, RAA's offer important opportunities as the information contained in the task-dependency graph can be used as a proxy for tasks' sensitivity in terms of error propagation. As such, we can determine the most sensitive parts of the task-dependency graph and perform resilience enhancements on them. These en-

hancements can vary from straightforward approaches such as task replication to sophisticated algorithmic checkers. In any case, the resilience techniques should be able to check for errors and correct as many as possible. The hardware can support these resilience enhancements either by enabling fast and efficient recomputations, by using the data that is already in the cache, by supporting fast data movements, or by having a few dedicated cores exclusively focused on running algorithmic checkers to detect and correct data corruptions.

Error Correcting Codes (ECC) [35] are a well known technique based on encoding some data in a redundant way specially conceived to detect any corruption in the original data. They are typically implemented in memory systems to detect and correct data corruptions. ECC can even be used to measure the fault rate that a particular hardware component experiments, which allows to deliver machines with a certain resilience warranty. The RAA approach can also benefit from these approaches by performing ECC checks over data while they are being transferred. For example, if a given packet is waiting in the queue of a switch, some ECC-based checks can be performed on it to detect corruptions and restore data integrity if possible. Such ECC checkers would not impact on performance, as they are performed during transmisions. As such, memory, on- and off-chip networks should have support for ECC.

Programmability Wall For current runtime systems, the granularity of tasks has to be coarse enough to neglect the runtime overhead. As a consequence, the minimum duration of the tasks has to be in the order of tens-hundreds of milliseconds. For some algorithms, finer granularities of tasks are required to better express the inherent parallelism of the algorithm. For that reason, architectural support for task-based execution paradigms has been proposed to accelerate the runtime system and tolerate much finer task granularities.

Managing thousands of tasks at runtime requires hardware mechanisms to accelerate the construction of the TDG [12, 44], and scheduling decisions [22, 33]. In order to balance the load of the application, task stealing techniques can be implemented in hardware [22]. In heterogeneous systems, load balancing and scheduling tasks is very complex from the programmer's perspective. Being able to manage a massive amount of fine-grain tasks together with hardware support for load balancing and scheduling will significantly facilitate writing parallel applications for many-cores.

Our Envisioned RAA Architecture Figure 2 and 3 depict the different parts of our envisioned runtime-aware architecture. We are designing a massively parallel system with multiple nodes where each node has multiple sockets, and each socket multiple clusters of cores. It is our position that future exascale systems will be heterogeneous at multiple levels, with different types of execution units (big and little cores) and accelerators (GPUs, vector processors, network, etc.), interconnection networks (electric, optic, and wireless), and memories (DRAM, non-volatile memory, etc.). Also, managing deep memory hierarchies and multi-level interconnection networks will be critical to obtain peak performance.

Considering hardware heterogeneity, data locality and reuse, power consumption, and reliability at runtime is crucial to overcome the walls that threaten Moore's Law. Enriched information at hardware level is required to allow the runtime system to optimize these objectives. In order to minimize the overhead of this runtime system, specific hardware support is mandatory [12, 22, 33]. With the help of these structures, key aspects of RAA's such as building the TDG, task scheduling, load balancing, and data placement can be made without affecting the performance of the application.

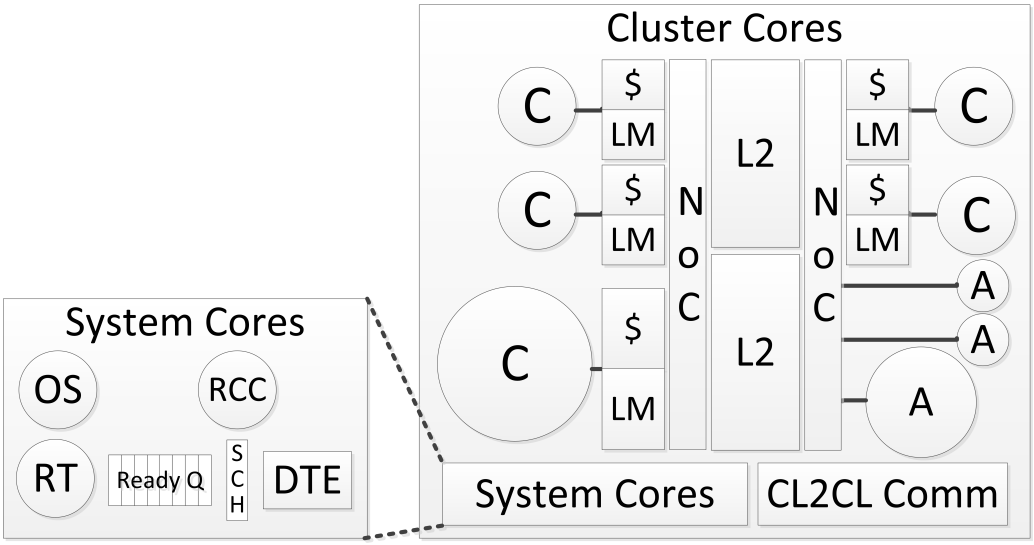

Figure 2. Runtime-Aware Architecture (RAA) system cores and cluster.

These structures would be the main part of the system cores block diagram in Figure 2. Similarly to the IBM Blue Gene/Q processor [17], a specific core for operating system activities (OS in Figure 2) is included in each cluster of cores, while the runtime front-end activities happen in a specific hardware (RT). It is still an open question if these two cores can be combined in a single structure. Runnable tasks are inserted into a queuing system similar to Carbon [22], which supports task stealing, and scheduled to the available cores. In order to characterize the hardware resources that each task type requires, a specific structure is conceived, denoted Resource Contention Core (RCC). Finally, a Data Transfer Engine (DTE) is in charge of data movements in the cores cluster. The DTE exploits data reuse, locality, and prefetches tasks inputs ahead of time.

The generic cluster cores in Figure 2 includes the execution cores, accelerators, on-chip network, and cache hierarchy. Effectively, processor cores thus serve as functional units. In RAA's, we propose to have a **hybrid memory hierarchy** with an L1 data cache and a local memory (or scratchpad) per core, as shown in Figure 2. Managing such a memory hierarchy is very difficult, but if done adequately, we can reduce significantly the coherence traffic and obtain a more energy efficient system. For example, we have proved that with the appropriate compiler support, such a hybrid hierarchy can be exploited for OpenMP codes [1]. In that approach, strided accesses are served by the local memory, while irregular accesses are served by the L1 data cache. A minimal hardware support is required for coherence and consistency purposes, but significant energy savings are obtained as a result. In the context of RAA's, task-based programming models such as OmpSs can be very helpful. For instance, task inputs and outputs can be automatically mapped to the local memories, while other accesses would be served by the L1 cache. The DTE in the system cores can manage DMA transfers to prefetch data in time. Finally, with this approach data movements due to coherence protocols would be significantly reduced.

The current **Network-on-Chip** (NoC) designs such as the ring- and bus-based topologies [39] provide an energy and throughput efficient solution for communication within small multi-cores. Since these traditional approaches for NoC's do not scale for many-cores, it is required to explore novel and scalable NoC approaches such as wireless interconnects, which can be used in conjunction with traditional wired and optical interconnects in novel RAA NoC topolo-

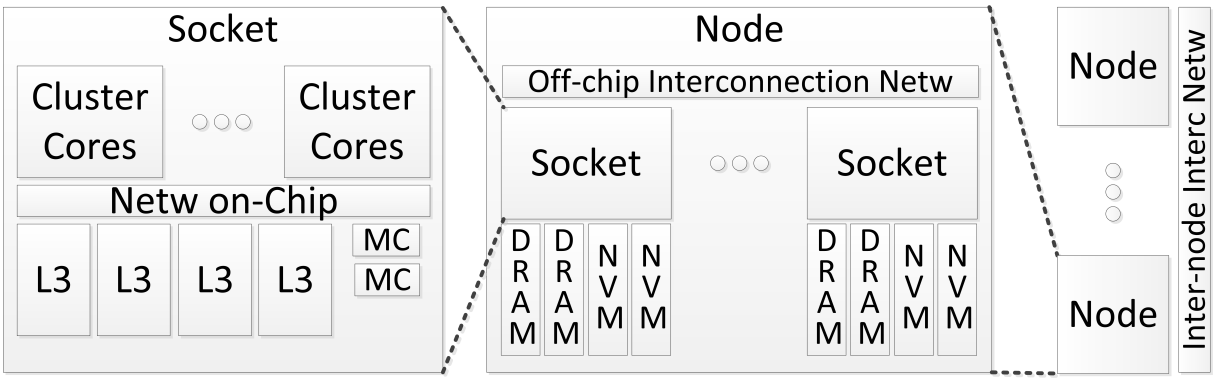

Figure 3. Runtime-Aware Architecture (RAA) socket, node and full system architecture.

gies for both 2D and 3D integrated circuits [45]. Finally, heterogeneous networks on chip are devised to serve the different characteristics of these structures in the hybrid memory hierarchy.

Cluster cores share a local L2 cache, while a large last-level on-chip cache is shared among different clusters in the socket. Local memories in different clusters will be able to communicate via a specific cluster to cluster communication engine (denoted *CL2CL Comm* in Figure 2). This engine could deal also with communications between different sockets in a node.

Different sockets in a node are connected with a high speed interconnection network, while different kinds of memories are available to the socket. Deciding whether to allocate memory in a regular DRAM or in a non-volatile memory is going to be a key aspect that is going to be easier to decide thanks to the enriched information provided by the dataflow representation of the application. For example, we can imagine that final outputs of an application should be mapped to persistent memories.

Several research groups have proposed system architectures with similarities to our envisioned RAA. For example, the SARC architecture [31] proposes having separate clusters of master and worker cores with local memories and coherent L2 and L3 caches. The Rigel architecture [21] proposes having clusters of cores with L1 instruction caches and incoherent L2 caches (per cluster), together with a global shared L3 cache. Finally, the Runnemede architecture [8] also relies on a dataflow execution model to execute in a near-threshold computing environment, with multiple clusters of homogeneous cores and a hierarchy of local memories. In this architecture, coherence between clusters is fully managed in software.

4. Related Work

4.1. Shared and Distributed Memory Programming Models

The most wide spread distributed memory API is MPI [16], which basically consists on a set of macros to explicitly indicate data exchanges between different tasks. Communications can be point to point or collective and use synchronous or asynchronous protocols, the first being more robust in the sense that data cannot be lost but paying the typical synchronization burden. OpenMP is an implementation of the classical shared memory multithreaded fork-join programming model [2]. By default, each thread runs its own parallelized section of code independently. Task- and data-level parallelism can be achieved trough work-sharing constructs that are used to divide the computational load among the threads. Threads are allocated to processors based on environment variables or in code using functions.

In the context of high performance computing, OpenMP is typically used to handle on-chip parallelism, since sharing data threads can use the memory resources more efficiently. However, OpenMP does not scale up to several tens of threads due to synchronization costs. Consequently, MPI must be used to achieved acceptable scalability in moderate and large scale runs. Typically, MPI is used to manage off-chip communications. It is interesting to state that hybrid MPI+OpenMP implementations somehow reflect the way the memory is organized in parallel clusters. As such, hybrid MPI+OpenMP have become the norm in the field of scientific computing.

4.2. Task-based Programming Models

Several task-based programming models have been developed in the past years: Cilk [5] is a runtime system for multithreaded codes. To fully use Cilk potential, the codes must be structured to expose parallelism and exploit locality, leaving Cilk's runtime with the responsibility of scheduling the computation to optimally run on a given platform. As such, the Cilk runtime system manages things like load balancing, synchronization, and communication protocols. Intel Threading Building Blocks (TBB) [32] is a C++ template library for task-based parallelism. It is supposed to simplify the parallel programming burden by asking the programmers to specify logical parallelism instead of threads and by letting the runtime library map logical parallelism onto threads that efficiently use the available hardware. CUDA (Compute Unified Device Architecture) is a parallel computing hardware and programming model developed by NVIDIA specifically designed for graphics processing units (GPUs). It allows the developers to access the virtual instruction set and memory of the parallel GPUs. OpenCL is a standard for cross-platform parallel programming. It allows programmers to implement and run parallel codes in heterogeneous platforms that may include CPU, GPU's, Digital Signal Processors (DSP) or other kinds of processors.

4.3. Dataflow Programming Models

Despite the fact that many of the programming models and computing paradigms mentioned above have been successful on achieving significant throughputs from current high performance computing infrastructures, the exhaustion of traditional performance enhancements techniques, like ILP or OoO, implies that more asynchronous and flexible programming models and runtime systems are going to be used to expose huge amounts of parallelism to the hardware. To reduce synchronization costs, optimize data motion across deep memory hierarchies, and handle critical path based optimizations, we need dataflow execution scenarios, which are much more flexible and have far more potential than traditional fork-join approaches.

Some dataflow programming models are being designed to overcome such issues. The Habanero [34] project is aimed to develop a programming model, a compiler and a runtime system to handle task-based asynchronous parallelism while taking advantage of locality among tasks and data distributed across the cores.

Charm++ [20] is a C++ based asynchronous message driven programming model. Its fundamental working units are message driven objects called *chares*. When the program triggers a message, an object is created and its associated work is carried out. Once the work is finished, the chare is destroyed and its output is sent to the next burst of chares that use these data as input. The chares are dynamically mapped to physical processors by the runtime system. Such

mapping is transparent to the program and, therefore, it is done with the aim to increase load balancing and fault tolerance.

Interestingly, domain specific programming languages, like Sequoia [13], have been developed to specifically express hierarchical memory by using some programming model primitives and thus allowing the programmer to describe data motion vertically through the machine and to localize computation to particular memory locations within it.

4.4. Data-Graph Execution ISA's

Some ISA's have specifically designed to execute the instructions by using a data-graph approach. The LAU multiprocessor [10] was proposed aiming to take advantage of parallelism over three levels: Between jobs, between tasks within a job and between instructions within a task. To enable scalable and distributed processor cores, tiled architectures have been proposed [26, 36, 40]. They consist of multiple simple processing elements connected by an on-chip interconnect. Scheduling instructions in tiled architectures is crucial to obtain good performance [26]. Explicit Data Graph Execution (EDGE) architectures are examples of tiled architectures [36]. Unlike traditional processor architectures that operate at the granularity of a single instruction, EDGE ISA's support large graphs of computation mapped to a flexible hardware substrate, with instructions in each graph communicating directly with other instructions, rather than going through a shared register file. This capability not only reduces design complexity, but amortizes execution overheads over a large graph of instructions.

5. Conclusions

In this paper, we introduce a first approach towards a *Runtime-Aware Architecture* (RAA), a massively parallel architecture designed from the runtime's perspective. This approach offers a unified and general solution that can potentially solve most of the problems we encounter in the current approaches: handling parallelism, the memory wall, the power wall, the programmability wall, and the upcoming reliability wall in a wide range of application domains from mobile up to supercomputers. Altogether, this novel approach toward future parallel architectures is the way to ensure continued performance improvements, getting us out of the technological hardship that computers have turned into, once more riding on Moore's Law.

Acknowledgments. *This work has been partially supported by the Spanish Ministry of Science and Innovation under grant TIN2012-34557, the HiPEAC Network of Excellence, and by the European Research Council under the European Union's 7th FP, ERC Grant Agreement number 321253. We would like to thank Alex Ramirez, Osman Unsal, Adrian Cristal, Mario Nemirovsky, Ramon Beivide, Alejandro Rico and all the RoMoL team for the prolific discussions and all the feedback that we received while preparing this manuscript.*

References

1. L. Alvarez, L. Vilanova, M. González, X. Martorell, N. Navarro, and E. Ayguadé. Hardware-software coherence protocol for the coexistence of caches and local memories. In *SC*, 2012.

2. E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang. The design of OpenMP tasks. *IEEE Trans. Parallel Distrib. Syst.*,

20(3):404–418, Mar. 2009.

3. P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta. CellSs: a programming model for the Cell BE architecture. In *SC*, 2006.

4. P. Bellens, J. M. Pérez, F. Cabarcas, A. Ramírez, R. M. Badia, and J. Labarta. CellSs: Scheduling techniques to better exploit memory hierarchy. *Scientific Programming*, 17(1-2):77–95, 2009.

5. R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *PPoPP*, pages 207–216, 1995.

6. D. Bohme, F. Wolf, B. R. de Supinski, M. Schulz, and M. Geimer. Scalable critical-path based performance analysis. In *IPDPS*, pages 1330–1340, 2012.

7. J. Bueno, X. Martorell, R. M. Badia, E. Ayguadé, and J. Labarta. Implementing OmpSs support for regions of data in architectures with multiple address spaces. In *ICS*, pages 359–368, 2013.

8. N. P. Carter et al. Runnemede: An architecture for ubiquitous high-performance computing. In *HPCA*, pages 198–209, 2013.

9. B. Chapman. *The Multicore Programming Challenge*, volume 4847 of *Lecture Notes in Computer Science*, pages 3–3. Springer Berlin Heidelberg, 2007.

10. D. Comte, N. Hifdi, and J.-C. Syre. The data driven lau multiprocessor system: Results and perspectives. In *IFIP*, pages 175–180, 1980.

11. A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas. OmpSs: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(2):173–193, 2011.

12. Y. Etsion, F. Cabarcas, A. Rico, A. Ramírez, R. M. Badia, E. Ayguadé, J. Labarta, and M. Valero. Task superscalar: An out-of-order task pipeline. In *MICRO*, pages 89–100, 2010.

13. K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Sequoia: programming the memory hierarchy. In *SC*, 2006.

14. V. Garcia, A. Rico, C. Villavieja, P. Carpenter, N. Navarro, and A. Ramirez. Adaptive runtime-assisted block prefetching on chip-multiprocessors. Technical Report UPC-DAC-RR-2014-8, Department of Computer Architecture, UPC, May 2014.

15. A. González, J. González, and M. Valero. Virtual-physical registers. In *HPCA*, pages 175–184, 1998.

16. W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface,* 2nd edition. MIT Press, Cambridge, MA, 1999.

17. R. Haring et al. The IBM Blue Gene/Q compute chip. *IEEE Micro*, 32(2):48–60, Mar. 2012.

18. J. L. Hennessy and D. A. Patterson. *Computer Architecture - A Quantitative Approach (5. ed.)*. Morgan Kaufmann, 2012.

19. International technology roadmap for semiconductors (ITRS), system drivers. In *ITRS*, 2011.

20. L. V. Kalé and S. Krishnan. Charm++: A portable concurrent object oriented system based on c++. In *OOPSLA*, pages 91–108, 1993.

21. J. H. Kelm, D. R. Johnson, M. R. Johnson, N. C. Crago, W. Tuohy, A. Mahesri, S. S. Lumetta, M. I. Frank, and S. J. Patel. Rigel: An architecture and scalable programming interface for a 1000-core accelerator. In *ISCA*, pages 140–151, 2009.

22. S. Kumar, C. J. Hughes, and A. D. Nguyen. Carbon: architectural support for fine-grained parallelism on chip multiprocessors. In *ISCA*, pages 162–173, 2007.

23. M. Manivannan, A. Negi, and P. Stenström. Efficient forwarding of producer-consumer data in task-based programs. In *ICPP*, pages 517–522, 2013.

24. M. Manivannan and P. Stenström. Runtime-guided cache coherence optimizations in multi-core architectures. In *IPDPS*, 2014.

25. V. Marjanović, J. Labarta, E. Ayguadé, and M. Valero. Overlapping communication and computation by using a hybrid MPI/SMPSs approach. In *ICS*, pages 5–16, 2010.

26. M. Mercaldi, S. Swanson, A. Petersen, A. Putnam, A. Schwerin, M. Oskin, and S. J. Eggers. Instruction scheduling for a tiled dataflow architecture. In *ASPLOS*, pages 141–150, 2006.

27. T. Mudge. Power: A first-class architectural design constraint. *Computer*, 34(4):52–58, Apr. 2001.

28. V. Papaefstathiou, M. Katevenis, D. S. Nikolopoulos, and D. N. Pnevmatikatos. Prefetching and cache management using task lifetimes. In *ICS*, pages 325–334, 2013.

29. J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta. Hierarchical task-based programming with starss. *Int. J. High Perform. Comput. Appl.*, 23(3):284–299, Aug. 2009.

30. J. Planas, R. M. Badia, E. Ayguade, and J. Labarta. Self-adaptive OmpSs tasks in heterogeneous environments. In *IPDPS*, pages 138–149, 2013.

31. A. Ramirez et al. The SARC architecture. *Micro, IEEE*, 30(5):16–29, Sept 2010.

32. J. Reinders. *Intel threading building blocks - outfitting C++ for multi-core processor parallelism*. O'Reilly, 2007.

33. D. Sanchez, R. M. Yoo, and C. Kozyrakis. Flexible architectural support for fine-grain scheduling. In *ASPLOS*, pages 311–322, 2010.

34. J. Shirako, J. M. Zhao, V. K. Nandivada, and V. N. Sarkar. Chunking parallel loops in the presence of synchronization. In *ICS*, pages 181–192, 2009.

35. J. Sim, G. H. Loh, V. Sridharan, and M. O'Connor. Resilient die-stacked DRAM caches. In *ISCA*, pages 416–427, 2013.

36. A. Smith, R. Nagarajan, K. Sankaralingam, R. McDonald, D. Burger, S. W. Keckler, and K. S. McKinley. Dataflow predication. In *MICRO*, pages 89–102, 2006.

37. S. Sridharan, G. Gupta, and G. S. Sohi. Holistic run-time parallelism management for time and energy efficiency. In *ICS*, pages 337–348, 2013.

38. O. Subasi, F. J. Arias, O. Unsal, J. Labarta, and A. Cristal. Leveraging a task-based asynchronous dataflow substrate for efficient and scalable resiliency. Technical Report UPC-DAC-RR-CAP-2013-12, Department of Computer Architecture, UPC, May 2013.

39. B. Vermeulen, J. Dielissen, K. Goossens, and C. Ciordas. Bringing communication networks on a chip: test and verification implications. *IEEE Communications Magazine*, 41(9):74–81, 2003.

40. E. Waingold, M. B. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. P. Amarasinghe, and A. Agarwal. Baring it all to software: Raw machines. *IEEE Computer*, 30(9):86–93, 1997.

41. S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick. The potential of the Cell processor for scientific computing. In *CF*, pages 9–20, 2006.

42. W. A. Wulf and S. A. McKee. Hitting the memory wall: implications of the obvious. *ACM SIGARCH computer architecture news*, 23(1):20–24, 1995.

43. X. Yang, Z. Wang, J. Xue, and Y. Zhou. The reliability wall for exascale supercomputing. *IEEE Trans. Comput.*, 61(6):767–779, June 2012.

44. F. Yazdanpanah, D. Jiménez-González, C. Alvarez-Martinez, Y. Etsion, and R. M. Badia. Analysis of the task superscalar architecture hardware design. In *ICCS*, pages 339–348, 2013.

45. D. Zhao and Y. Wang. SD-MAC: Design and synthesis of a hardware-efficient collision-free QoS-aware MAC protocol for wireless network-on-chip. *IEEE Trans. Comput.*, 57(9):1230–1245, 2008.

Received June 8, 2014.