Runtime-aware Architectures: A Second Approach

Mateo Valero, Lluc Alvarez, Emilio Castillo, Dimitrios Chasapis, Timothy Hayes, Luc Jaulmes, Oscar Palomar, Marc Casas, Miquel Moreto, Eduard Ayguade, Jesus Labarta

Barcelona Supercomputing Center (BSC)
Universitat Politècnica de Catalunya (UPC)

Abstract. In the last few years, the traditional ways to keep the increase of hardware performance to the rate predicted by the 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. 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 face. The runtime of the parallel application 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 the paper, we introduce a first approach towards a Runtime-Aware Architecture (RAA), a massively parallel architecture designed from the runtime's perspective.

1 Introduction and Motivation

When uniprocessors were prominent, Instruction Level Parallelism (ILP) and Data Level Parallelism (DLP) were used to maximize the number of instructions per cycle. The most important designs devoted to exploit ILP were superscalar and Very Long Instruction Word (VLIW) processors. VLIW requires to statically figure out dependencies between instructions and to schedule them accordingly. However, since compilers do not do good a job obtaining optimal schedulings, VLIW is not successful in achieving the maximal ILP workloads have. Superscalar designs try to overcome the increasing memory latencies, the so called Memory Wall [24], by using Out of Order (OoO) and speculative executions [11]. Also, improvements like prefetching, to fetch data from main memory in advance, memory hierarchies, to exploit temporal and spatial locality, and reorder buffers, to expose more instructions to the hardware, have been proposed. 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 typically in charge of decoupling the high level application source code and the underlying hardware. In this context, new architecture ideas were applied at the pipeline level without changing the ISA.

Besides the problems associated with the memory wall, traditional useful ways to increase hardware performance at the Moore's Law rate vanished. For instance, the processor clock frequency stagnated because, when it reached a threshold, the power per unit of area (power density) could not be dissipated. That problem was called the Power Wall. Indeed, a study made by the International Technology Roadmap for Semiconductors expects the frequency to increase by 5% every year for the next 15 years [12]. Therefore, further performance increases are expected to come from larger concurrency levels rather than higher frequencies.

To overcome the stagnation of the processor clock frequency, vendors started to release multi-core devices over a decade ago. By exploiting Task Level Parallelism (TLP) multi-core devices may achieve significant performance gains. 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. Energy consumption is also a major problem since the current increase makes computing challenges such as building an exascale machine completely infeasible. This set of challenges related to power consumption issues constitutes a new power wall.

Also, there is a trend towards more heterogeneous multi-core systems, which might have processors with different ISA's connected through deep and complex memory hierarchies. To manage data motion among these memory hierarchies 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. The Programmability Wall [7] concept is commonly use to categorize the above mentioned data management and programmability issues.

Multi-core architectures provide significant performance levels under low voltages. 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 total number of hardware components in future designs is expected to increase by several orders of magnitude, which only makes the fault prevalence problem more dramatic. Therefore, in addition to the current challenges in parallelism, memory and power management, we are moving towards a Reliability Wall.

Since the irruption of multi-cores and parallel applications it is not possible anymore to write high-level code in a completely hardware oblivious way. An option is to transfer the role of decoupling applications from the hardware to the runtime system , that is, to let the runtime layer to be 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 conceived as simple symmetric multiprocessors (SMP) on a chip. However, that is not enough to overcome all the problems that multi-cores already have to face. To properly take advantage of their potential, tight hardware-software collaboration is required. The runtime has to drive the design of hardware components to overcome the challenges of the above mentioned walls. We envision a Runtime-Aware Architecture (RAA) [22], 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 Task Dependency Graph (TDG) can be built at runtime or statically. In this context, the runtime drives the design of new architecture components to support activities like the construction of the TDG [9], among other things.

In the next sections, we describe some illustrative examples of techniques that allow alleviating the challenges arisen from the Memory, Power, Resilience and Programmability Walls. These examples show that an adequate hardware-software co-designed system can significantly improve the final performance and energy consumption of our envisioned RAA's. Section 2 presents a hybrid memory approach that combines scratchpads and caches to deal with the Memory Wall. Section 3 shows how task criticality and hardware reconfiguration can reduce energy consumption. We also highlight the important of vector processors in that same section. Next, Section 4 describes how the asynchrony provided by the OmpSs programming model combined with fine grain error detection techniques can be efficiently combined to mitigate the Resilience Wall. Section 5 provides some examples to illustrate how to deal with the Programmability Wall. Finally, Section 6 presents the related work and Section 7 summarizes the main findings of this work.

2 Memory Wall

The increasing number of cores in shared memory manycore architectures causes important power and scalability problems in the memory hierarchy. One solution is to introduce ScratchPad Memories (SPM) alongside the caches, forming a hybrid memory hierarchy. SPMs are more power-efficient than caches and they do not generate coherence traffic, but they suffer from poor programmability. A good way to hide the programmability difficulties to the programmer is to give the compiler the responsibility of generating code to manage the scratchpad memories but, unfortunately, compilers do not succeed in generating this code in the presence of random memory accesses with unknown aliasing hazards.

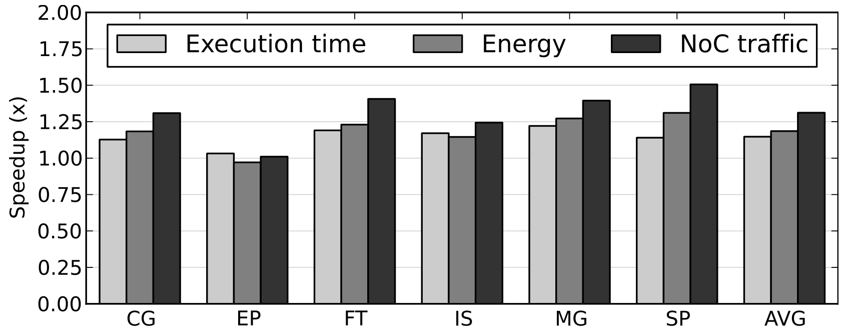

Fig. 1: Performance, energy and NoC traffic speedup of the hybrid memory hierarchy on a 64-core processor with respect to a cache-only system.

We propose a hardware/software co-designed coherence protocol that allows the compiler to always generate code to manage the SPMs of hybrid memory hierarchies, even if it encounters memory aliasing hazards between strided and random memory references [1]. On the software side, the proposed solution consists on simple modifications to the compiler analyses so that it can classify memory references in three categories: strided memory references, random memory references that do not alias with strided ones, and random memory references with unknown aliases. The compiler then transforms the code for the strided memory references to map them to the SPMs using tiling software caches, while for the random memory references that do not alias with strided ones it generates memory instructions that are served by the cache hierarchy. For the random memory references with unknown aliasing hazards the compiler generates a special form of memory instruction that gives the hardware the responsibility to decide what memory is used to serve them. On the hardware side, a coherence protocol is proposed so that the architecture can serve the memory accesses with unknown aliasing hazards with the memory that keeps the valid copy of the data. For this purpose the hybrid memory hierarchy is extended with a set of directories and filters that track what part of the data set is mapped and not mapped to the SPMs. These new elements are consulted at the execution of memory accesses with unknown aliases, so all memory accesses can be correctly and efficiently served by the appropriate memory.

As shown in Figure 1, the proposed system achieves significant speedups in terms of performance, energy and NoC traffic for several NAS benchmarks. Average improvements reach 14.7%, 18.5% and 31.2%, respectively. Reduced execution time combined with more energy-efficient accesses to the hybrid memory hierarchy lead to the highlighted reduction in energy. Even for benchmarks with minimal accesses to the SPM (as in the case of `EP`), performance, energy consumption and NoC traffic are not degraded

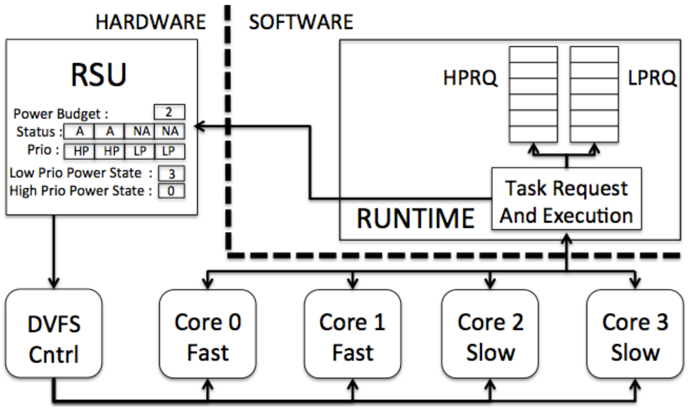

Fig. 2: Runtime Support Unit (RSU) to accelerate critical tasks in the application.

3 Power Wall

3.1 Exploiting Task Criticality

Task-based data-flow programming models intrinsic information and execution mechanisms can be exploited to open new performance gains or power savings opportunities. Such programming models overcome the performance of widely used threading approaches when running on heterogeneous many-cores. Furthermore, task criticality information can be exploited to optimize execution time or Energy-Delay Product (EDP). A task is considered critical if it belongs to the critical path of the Task Dependency Graph. Consequently, critical tasks can be run in faster or accelerated cores while non critical tasks can be scheduled to slow cores without affecting the final performance and reducing overall energy consumption. Moreover, task criticality can be simply annotated by the programmer and exploited to reconfigure the hardware by using DVFS, achieving improvements over static scheduling approaches that reach 6.6% and 20.0% in terms of performance and EDP on a simulated 32-core processor, respectively.

The cost of reconfiguring the hardware with a software-only solution rises with the number of cores due to locks contention and reconfiguration overhead. Therefore, novel architectural support is proposed to reduce these overheads on future many-core systems. Figure 2 illustrates such hardware support to build a runtime-aware architecture. The runtime system is in charge of informing the Runtime Support Unit (RSU) of the criticality of each running task. Based on this information and the available power budget, the RSU decides the frequency of each core, which can be seen as a criticality-aware turbo boost mechanism. Consequently, this hardware support minimally extends hardware structures already present in current processors, which allows further improvements in performance with negligible hardware overhead. The integrated solution proposed, which goes from the source code to the hardware level passing through the run-

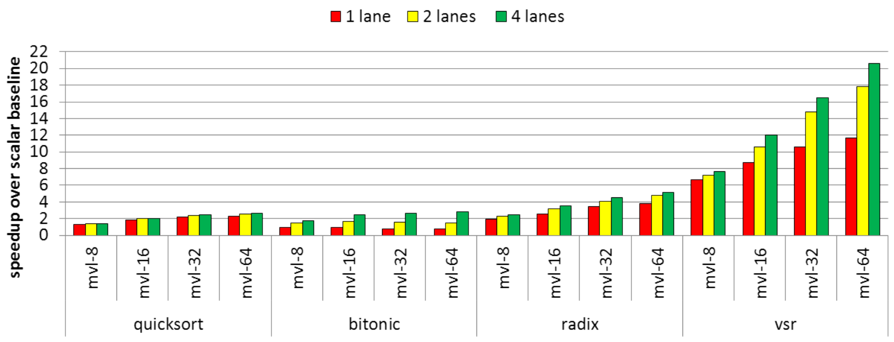

Fig. 3: Speedup over a scalar baseline for different vectorized sorting algorithms. Different maximum vector lengths (MVL) and lanes are considered.

time and the operating system, shows the need for a multi-layer approach to optimally exploit the heterogeneity of future many-core systems.

3.2 Vector Processors

Due to their energy efficiency, SIMD extensions have become ubiquitous in modern microprocessors and are expected to grow in width and functionality in future generations. After extensive analysis on three diverse sorting algorithms in the context of future SIMD support, we learned that all of the algorithms suffer from bottlenecks and scalability problems due to the irregularity of the DLP and the limitations of a standard SIMD instruction set. Based on these findings we proposed VSR sort [10], a novel way to efficiently vectorize the radix sort algorithm. To enable this algorithm in a SIMD architecture we defined two new instructions: `vector prior instances` (VPI) and `vector last unique` (VLU). VPI uses a single vector register as input, processes it serially and outputs another vector register as a result. Each element of the output asserts exactly how many instances of a value in the corresponding element of the input register have been seen before. VLU also uses a single vector register as input but produces a vector mask as a result that marks the last instance of any particular value found. We provided a suitable hardware proposal that includes both serial and parallel variants, demonstrating that the algorithm scales well when increasing the maximum vector length, and works well both with and without parallel lockstepped lanes. VSR sort is a clear example of the benefits that a hardware/software co-designed system can offer.

As illustrated in Figure 3, VSR sort shows maximum speedups over a scalar baseline between 7.9x and 11.7x when a simple single-lane pipelined vector approach is used, and maximum speedups between 14.9x and 20.6x when as few as four parallel lanes are used. Next, we compare VSR sort with three very different vectorized sorting algorithms: quicksort, bitonic mergesort and a previously proposed implementation of radix sort. VSR sort outperforms all of the aforementioned algorithms and achieves a comparatively low Cycles Per Tuple (CPT) without strictly requiring parallel lanes. It has a complexity of $O(k \cdot n)$ meaning that this CPT will remain constant as the input size increases, a highly-desirable property of a sorting algorithm. The k factor is significantly improved over the original vectorized radix sort as well as the constant performance factor. Its dominant memory access pattern is unit-stride which helps maximise the utilisation of the available memory bandwidth. Unlike the previous vectorized radix sort, VSR sort does not replicate its internal bookkeeping structures which consequently allows them to be larger and reduces the number of necessary passes of the algorithm. On average VSR sort performs 3.4x better than the next-best vectorized sorting algorithm when run on the same hardware configuration.

4 Resilience Wall

Relying on error detection techniques already available in commodity hardware, we develop algorithmic-level error correction techniques for Detected and Uncorrected Errors (DUE) in iterative solvers. When a data loss or corruption is detected, we use simple algorithmic redundancies that are not available under coarser grained error models, such as node failure.

Using straightforward relations existing in the solver, we interpolate the lost data and manage to recover it exactly. Our forward recovery scheme allows better performance than backwards recoveries such as checkpointing and rollback. Other previous recoveries trade in convergence rate for recovery by restarting, and the better of these methods gain an immediate reduction in the solver's residual norm. We are able to avoid sacrificing convergence rate altogether thanks to the exactitude of the recovered data, allowing the solver to continue, which is better in the long run.

Furthermore, we can lever the asynchrony of task-based programming models to perform our recoveries' interpolations simultaneously with the normal workload of the solver. This allows to reduce the overheads of our recovery technique, and is done with virtually no burden on the programmer thanks to the programming model, by scheduling the recoveries in tasks that are placed out of the critical path of the solver.

Figure 4 illustrates these behaviours, for a single error scenario where the Conjugate Gradient method for the matrix thermal2 is disturbed by a DUE around 30s. The lightblue checkpointing scheme incurs a significant overhead when rolling back, and the restart method, in green, has a slower convergence afterwards, when compared to the ideal baseline, in red, which has no fault injected nor resilience mechanism. Our recovery technique, in purple, shows a convergence time close to the ideal baseline, and its asynchronous counterpart, in blue, displays an even smaller overhead.

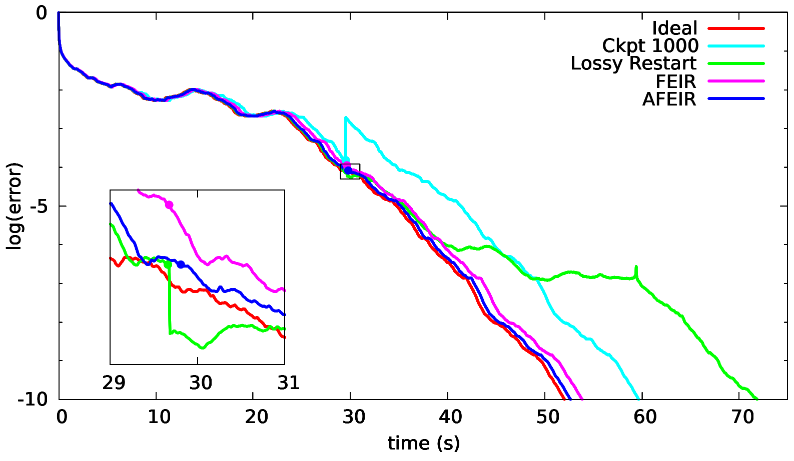

Fig. 4: CG execution example with a single error occurring at the same time for all implemented mechanisms.

5 Programmability Wall

Task parallel models are being widely used to program parallel shared memory machines, offering an alternative to the effective, but difficult to use, Pthreads model. They offer simpler syntax than Pthreads, which allows the programmer to easily describe parallel work as asynchronous tasks. Task-based models are coupled with a runtime system, which at its simplest form, takes the burden of thread management from the programmer. Such runtime systems can offer additional functionality, such as load balancing or tracking data dependencies between different tasks, ensuring their correct order of execution. In order for the runtime to track data dependencies, task parallel models often offer syntactic tools to the programmer for expressing data-flow relations between tasks [2, 8, 13, 21, 23]. The OpenMP standard has recently adopted tasks and dataflow extensions to its syntax [17], allowing dynamic tracking of dependencies during execution.

However, it is important to evaluate how effective this emerging programming model is in terms of usability and performance. This has been extensively studied, but only in the scope of HPC kernel applications [3, 19, 21]. Parallelism today is employed by all kinds of applications, from economic calculations to search engines and multimedia. We believe that it is important to understand how task parallelism can be effectively adopted in different application domains and where its limitations lie. To answer this question we have ported a large subset (10 out of 13) of applications from the PARSEC benchmark suite [5]. These applications are representative of the different domains that are currently adopting parallelism to improve their performance. We have used the OmpSs programming model, which is a forerunner of OpenMP 4.0, to port the bench-

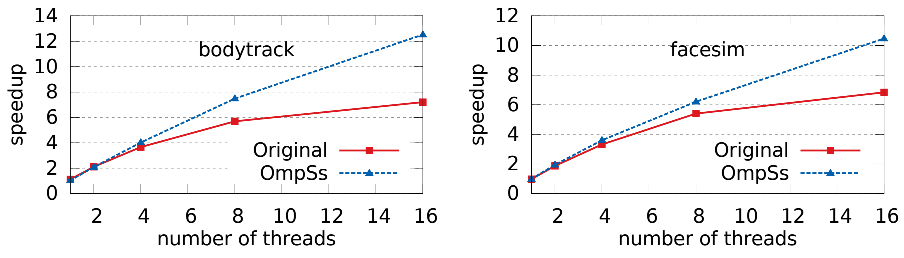

Fig. 5: Scalability comparion between OmpSs and Pthreads.

marks. We evaluate our task-based implementations in terms of usability and performance against the native implementations of the PARSEC suite (which is always in Pthreads, except in the case of `freqmine`, which uses OpenMP's parallel loops).

To evaluate the usability of task-parallel models we need to see how maintainable and compact the code is, compared to Pthreads/OpenMP and also how expressive the model is. By measuring the lines of code, we have observed that OmpSs' syntax is less verbose than Pthreads, for most benchmarks. Compared to OpenMP loops, we did not observe any benefit from using tasks. Moreover, we were able to express the same level of parallelism as the Pthreads/OpenMP versions using tasks and dataflow relations. In some cases, where the applications used pipeline parallelism, we could express additional parallelism executing asynchronously I/O intensive sequential stages and overlapping them with computation intensive parallel regions. In these cases we could also improve the scalability of the applications. Figure 5 shows the scalability comparison between OmpSs and Pthreads versions for `bodytrack` and `facesim` on a 16-core machine. Both applications improve significantly their scalability over the original code, reaching a scaling factor of 12 and 10, respectively, when running with 16 cores. Overall, our evaluation shows that data-parallel applications, limited to simple do-all loops, can be implemented using tasks, but cannot benefit from them in neither programmability nor performance. Benchmarks that have pipeline parallelism can greatly reduce the lines of code of the application, since simple data-flow relations can replace user implemented queuing and thread management systems. Performance can also improve if the pipeline can be extended to asynchronously execute sequential I/O intensive regions. Applications that work with irregular or dynamic data structures cannot benefit from dynamic dependence analysis, since the standard syntax can only be used to express data footprints of continguous data structures.

6 Related Work

Previous work has been devoted to many-core architectures with a single global address space where parallel work is expressed in terms of a taskcentric bulksynchronous model using hardware support [14]. The execution paradigm of this approach is based on tasks, like the one presented in this paper. However, this previous approach assummes the mapping of the tasks to the functional units of the processor to be specified in the application binary, significantly reducing its flexibility.

Some approaches propose architectures composed of multiple processor types and a set of user-managed direct memory access (DMA) engines that let the runtime scheduler overlap data transfer and computation [20]. The runtime system automatically allocates tasks on the heterogeneous cores and schedules the data transfers through the DMA engines. The programming model suggested by this approach supports various highly parallel applications, with matching support from specialized accelerator processors. The Runtime-aware architecture presented in this paper includes these ideas and incorporates new ones (resilience, hardware support for frequency reconfiguration, etc.) to achieve a more general and robust design.

Other many-core proposals with separate execution units for runtime and application code, application-managed on-chip memory and direct physical addressing, a hierarchical on-chip network and a codelet-based execution model [6] have been suggested to reduce energy consumption and increase performance. The main drawback of such designs is programmability, as they require the memory and data transfer management to be done at the source code level.

Hardware techniques to accelerate dynamic task scheduling on scalable CMPs have also been suggested [15]. They consist in relatively simple hardware that can be placed far from the cores. While our proposal also aims to support task scheduling, it incorporates many more innovations like runtime-based hybrid memory designs or hardware support for reconfiguration, to mention just two.

Some previous approaches aim to exploit the runtime system information to either reduce cache coherence traffic [16] or enable software prefetching mechanisms [4, 18]. The Runtime-aware architecture presented in this paper gathers all these previous experiences and provides an holistic view that integrates not only the memory system but also all the hardware components to ride again on the Moore's Law.

7 Conclusions

Our approach towards parallel architectures offers a single solution that could 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.

References

1. Alvarez, L., Vilanova, L., Moreto, M., Casas, M., Gonzàlez, M., Martorell, X., Navarro, N., Ayguadé, E., Valero, M.: Coherence protocol for transparent management of scratchpad memories in shared memory manycore architectures. In: International Symposium on Computer Architecture (ISCA). pp. 720–732 (2015)
2. Ayguadé, E., Copty, N., Duran, A., Hoeflinger, J., Lin, Y., Massaioli, F., Teruel, X., Unnikrishnan, P., Zhang, G.: The design of OpenMP tasks. IEEE Trans. Parallel Distrib. Syst. 20(3), 404–418 (Mar 2009)
3. Ayguadé, E., Duran, A., Hoeflinger, J., Massaioli, F., Teruel, X.: An experimental evaluation of the new OpenMP tasking model. In: LCPC. pp. 63–77 (2007)
4. Bellens, P., Perez, J.M., Badia, R.M., Labarta, J.: CellSs: a programming model for the cell B.E. architecture. In: Supercomputing (SC) (2006)
5. Bienia, C.: Benchmarking Modern Multiprocessors. Ph.D. thesis, Princeton University (January 2011)
6. Carter, N.P., Agrawal, A., Borkar, S., Cledat, R., David, H., Dunning, D., Fryman, J.B., Ganev, I., Golliver, R.A., Knauerhase, R.C., Lethin, R., Meister, B., Mishra, A.K., Pinfold, W.R., Teller, J., Torrellas, J., Vasilache, N., Venkatesh, G., Xu, J.: Runnemede: An architecture for ubiquitous high-performance computing. In: International Symposium on High Performance Computer Architecture (HPCA). pp. 198–209 (2013)
7. Chapman, B.: The multicore programming challenge. In: International Conference on Advanced Parallel Processing Technologies (APPT). pp. 3–3 (2007)
8. Duran, A., Ayguadé, E., Badia, R.M., Labarta, J., Martinell, L., Martorell, X., Planas, J.: OmpSs: a Proposal for Programming Heterogeneous Multi-Core architectures. Parall. Proc. Lett. 21(2), 173–193 (2011)
9. Etsion, Y., Cabarcas, F., Rico, A., Ramírez, A., Badia, R.M., Ayguadé, E., Labarta, J., Valero, M.: Task superscalar: An out-of-order task pipeline. In: MICRO. pp. 89–100 (2010)
10. Hayes, T., Palomar, O., Unsal, O.S., Cristal, A., Valero, M.: VSR sort: A novel vectorised sorting algorithm & architecture extensions for future microprocessors. In: International Symposium on High Performance Computer Architecture (HPCA). pp. 26–38 (2015)
11. Hennessy, John L.; Patterson, D.A.: Computer architecture - A quantitative approach (5. ed.). Morgan Kaufmann (2012)
12. International technology roadmap for semiconductors (ITRS) (2011)
13. Jenista, J.C., Eom, Y.h., Demsky, B.C.: OoOJava: Software out-of-order execution. SIGPLAN Not. 46(8), 57–68 (Feb 2011)
14. Kelm, J.H., Johnson, D.R., Johnson, M.R., Crago, N.C., Tuohy, W., Mahesri, A., Lumetta, S.S., Frank, M.I., Patel, S.J.: Rigel: An architecture and scalable programming interface for a 1000-core accelerator. In: In ISCA 09. pp. 140–151

15. Kumar, S., Hughes, C.J., Nguyen, A.: Carbon: Architectural support for fine-grained parallelism on chip multiprocessors. In: International Symposium on Computer Architecture (ISCA). pp. 162–173 (2007)
16. Manivannan, M., Stenstrom, P.: Runtime-guided cache coherence optimizations in multi-core architectures. In: International Parallel and Distributed Processing Symposium (IPDPS). pp. 625–636 (2014)
17. OpenMP Architecture Review Board: OpenMP application program interface version 4.0 (Jul 2013), `http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf`
18. Papaefstathiou, V., Katevenis, M.G., Nikolopoulos, D.S., Pnevmatikatos, D.: Prefetching and cache management using task lifetimes. In: International Conference on Supercomputing (ICS). pp. 325–334 (2013)
19. Podobas, A., Brorsson, M.: A comparison of some recent task-based parallel programming models. In: Multiprog (2010)
20. Ramírez, A., Cabarcas, F., Juurlink, B.H.H., Alvarez, M., Sánchez, F., Azevedo, A., Meenderinck, C., Ciobanu, C.B., Isaza, S., Gaydadjiev, G.: The SARC architecture. IEEE Micro 30(5), 16–29 (2010)
21. Tzenakis, G., Papatriantafyllou, A., Kesapides, J., Pratikakis, P., Vandierendonck, H., Nikolopoulos, D.S.: BDDT: Block-level dynamic dependence analysis for deterministic task-based parallelism. SIGPLAN Not. 47(8), 301–302 (Feb 2012)
22. Valero, M., Moreto, M., Casas, M., Ayguade, E., Labarta, J.: Runtime-aware architectures: A first approach. International J. Supercomputing Frontiers and Innovations 1(1), 29–44 (2014)
23. Vandierendonck, H., Tzenakis, G., Nikolopoulos, D.: A unified scheduler for recursive and task dataflow parallelism. In: International Conference on Parallel Architectures and Compilation Techniques (PACT). pp. 1–11 (Oct 2011)
24. Wulf, W.A., McKee, S.A.: Hitting the memory wall: Implications of the obvious. SIGARCH Comput. Archit. News 23(1), 20–24 (Mar 1995)