High-Performance Embedded Architecture and Compilation Roadmap

Koen De Bosschere[1,2], Wayne Luk[1,3], Xavier Martorell[1,4], Nacho Navarro[1,4], Mike O'Boyle[1,5], Dionisios Pnevmatikatos[1,6], Alex Ramirez[1,4], Pascal Sainrat[1,7], André Seznec[1,8], Per Stenström[1,9], and Olivier Temam[1,10]

[1] HiPEAC Network of Excellence
`http://www.HiPEAC.net`
[2] Ghent University, Belgium
[3] Imperial College, UK
[4] UPC, Spain
[5] University of Edinburgh, UK
[6] ICS FORTH, Greece
[7] CNRS, France
[8] IRISA, France
[9] Chalmers, Sweden
[10] INRIA Futurs, France

Abstract. One of the key deliverables of the EU HiPEAC FP6 Network of Excellence is a roadmap on high-performance embedded architecture and compilation – the HiPEAC Roadmap for short. This paper is the result of the roadmapping process that took place within the HiPEAC community and beyond. It concisely describes the key research challenges ahead of us and it will be used to steer the HiPEAC research efforts.
The roadmap details several of the key challenges that need to be tackled in the coming decade, in order to achieve scalable performance in multi-core systems, and in order to make them a practical mainstream technology for high-performance embedded systems.
The HiPEAC roadmap is organized around 10 central themes: (i) single core architecture, (ii) multi-core architecture, (iii) interconnection networks, (iv) programming models and tools, (v) compilation, (vi) run-time systems, (vii) benchmarking, (viii) simulation and system modeling, (ix) reconfigurable computing, and (x) real-time systems. Per theme, a list of challenges is identified. In total 55 key challenges are listed in this roadmap. The list of challenges can serve as a valuable source of reference for researchers active in the field, it can help companies building their own R&D roadmap, and – although not intended as a tutorial document – it can even serve as an introduction to scientists and professionals interested in learning about high-performance embedded architecture and compilation.

Key words: HiPEAC, roadmap, single core architecture, multi-core architecture, interconnection networks, programming models and tools, compilation, run-time systems, benchmarking, simulation and system modelling, reconfigurable computing, real-time systems

Introduction

Modern embedded systems have computing resources that by far surpass the computing power of the mainframes of the sixties. This has been made possible thanks to technology scaling, architectural innovations, and advances in compilation. The driving force was the ability to speed-up existing binaries without much help of the compiler. However, since 2000 and despite new progress in integration technology, the efforts to design very aggressive and very complex wide issue superscalar processors have essentially come to a stop. The exponentially increasing number of transistors has since then been invested in ever larger on-chip caches, but even there we have reached the point of diminishing return.

Therefore, for the last five years, it has become more and more obvious that the quest for the ultimate performance on a single chip uniprocessor is becoming a dead-end. Although there are still significant amounts of unexploited instruction-level parallelism left, the complexities involved to extract it and the increased impact of wire-delay on the communication have left us with few ideas on how to further exploit it. Alternatively, further increasing the clock frequency is also getting more and more difficult because (i) of heat problems and (ii) of too high energy consumption. The latter is not only a technical problem for both server farms and mobile systems, but in the future, it is also going to become a marketing weapon targeted at the growing number of environmentally-aware consumers and companies in search of a greener computer.

For these and other reasons, there is currently a massive paradigm shift towards multi-core architectures. Instead of scaling performance by improving single core performance, performance is now scaled by putting multiple cores on a single chip, effectively integrating a complete multiprocessor on one chip. Since the total performance of a multi-core is improved without increasing the clock frequency, multi-cores offer a better performance/Watt ratio than a single core solution with similar performance. The interesting new opportunity is now that Moore's Law (which is still going to bring higher transistor density in the coming years) will make it possible to double the number of cores every 18 months. Hence, with 4 cores of the complexity of high-performance general-purpose processors already on a chip today, we can expect to fit as many as 256 such cores on a chip in ten years from now. The future scaling in the number of cores is called the multi-core roadmap hereafter.

This paradigm shift has a profound impact on all aspects of the design of future high-performance systems. In the multi-core roadmap, the processor becomes the functional unit, and just like floating-point units were added to single-core processors to accelerate scientific computations, special-purpose computing nodes will be added to accelerate particular application types (media processing, cryptographic algorithms, digital signal processing, . . .) leading to heterogeneous multi-cores. Heterogeneous multi-cores add a new design complexity issue, because special-purpose computing nodes can have a significant impact on the memory hierarchy of the system. This will require specially designed communication paths for which bus-based interconnects are no longer suited.

On the multi-core roadmap, cheaper and more reliable high-performance switched serial interconnects will be used. This trend is evident in all recent high-performance interconnects such as PCI Express, ASI, FSB, HyperTransport.

Programming these (heterogeneous) multi-core systems requires an advanced parallel programming environment enabling the user to manually express concurrency as well as to automatically discover thread-level parallelism (in contrast to instruction-level parallelism) in sequential code. Automatically extracting thread-level parallelism or auto-parallelization has been extensively studied for scientific programs since the 1970s. Despite impressive gains for certain applications it is highly sensitive to the programming idiom. Common programming languages featuring arbitrary pointer manipulations (like C or C++) make this auto-parallelization extremely difficult. Due to this difficulty in exploiting parallelism and the easier option of waiting for the next technology generation to provide greater performance, parallel computing has failed to deliver in the past. However, now it seems that thread-level parallelism is the only route to performance scalability together with customization. If we cannot extract sufficient thread-level parallelism from the user's code, it does not matter how many cores are available – there will be no performance gain. This situation has implications far beyond architecture and compilation as it will affect all consumers used to the steady improvement of application performance across computer generations. Such improvements will no longer occur unless the application is parallelized.

The increased computing power for a given power budget will pave the way for new high-performance embedded applications: more demanding multimedia applications, advanced online biomedical signal processing, software-defined radio, biometric data processing like voice processing and image recognition. Many of these applications have hard or soft real-time requirements. This is challenging in a multi-core system because all cores share common resources like the lower level caches and the off-chip communication bandwidth – making it more difficult to compute the worst case execution time. Due to the better performance/Watt metric for multi-cores, they will also be used as elementary computing nodes in supercomputers where they will be used to run traditional scientific workloads. Hence, multi-cores will span the complete computational spectrum.

It is clear that this paradigm shift is so profound that it is affecting almost all aspects of system design (from the components of a single core up to the complete system), and that a lot of research and tool development will be needed before it will be possible to bring many-core processors to the masses.

The remainder of this paper details several of the key challenges that need to be tackled in the coming decade, in order to achieve scalable performance in multi-core systems, and in order to make them a practical mainstream technology for embedded systems. It is in the first place a roadmap for research and is not meant to be a roadmap on industrial R&D. Furthermore it is a roadmap on high-performance embedded architecture and compilation, hence it is about future embedded hardware and tools to exploit that hardware in the broad sense. It is neither a technology roadmap, nor an embedded application roadmap as these aspects are already covered by other documents like the ITRS roadmap and the ISTAG documents.

The roadmap is structured around 10 themes: (i) single core architecture, (ii) multi-core architecture, (iii) interconnection networks, (iv) programming models and tools, (v) compilation, (vi) run-time systems, (vii) benchmarking, (viii) simulation and system modeling, (ix) reconfigurable computing, and (x) real-time systems. Per theme, a list of challenges is identified. More important challenges are put higher in the list.

The fact that we distinguish 10 themes does not mean that these themes are independent; it is just a way to structure this document. In fact, some of the challenges have moved from one theme to another several times during the roadmapping process. Other issues like power are popping up as a challenge in different themes.

The description of the individual challenges is kept concise, and we have tried to describe just the challenge, not the solutions as we did not want to impose our vision on the possible solutions. For the same reason, we decided not to include references per challenge.

1 Single Core Architecture

Many of the classical uniprocessor trade-offs of the last 20 years will have to be reconsidered when uniprocessors are used as building blocks in a multi-core system. Devoting precious silicon area to aggressive out-of-order execution hardware might no longer lead to an optimal solution, and using the area to implement two simpler cores can result in a better performance and/or lower power consumption (in a sense we might be witnessing the CISC-RISC transition again – this time at the core level). However, since many existing workloads are inherently sequential, and since even a parallelized application will contain significant amounts of sequential code, giving up on single core performance might cause serious problems for this class of applications. The research on processor micro-architecture must therefore continue to focus on the trade-off between performance and complexity of the micro-architecture. The following challenges are identified for future single core architectures.

Challenge 1.1: Complexity Reduction

The aggressive out-of-order execution mechanism is very complex, its verification is very time-consuming, its implementation is using up a lot of silicon area, and its operation is consuming a lot of power. In order to make it a suitable candidate as a basic building block in a multi-core system, its complexity has to be reduced, without compromising the single-core performance too much.

Challenge 1.2: Power Management

Besides the creation of specialized hardware modules, Dynamic-Voltage-Frequency-Scaling (DVFS) has been a prevailing power managing technique so far. It not only helps in reducing the dynamic power consumption, but it also helps fighting static (leakage) power consumption. Unfortunately, scaling down the voltage leads to an increase in the number of soft errors which creates a reliability problem in future systems. As a result, while DVFS has been an important technique so far, it will be less attractive as we move along. Hence, novel techniques will be needed to manage both dynamic and static power consumption in single cores. If not, it is expected that future architectural designs will be more and more constrained by leakage power consumption.

Challenge 1.3: Thermal Management

With the increasing integration density, power consumption is not the only concern. Power density has also risen to very high levels in several parts of the processor. Temperature hotspots are therefore becoming a major concern on processors, since they can result in transient or permanent failure. The temperature hotspots have also a major impact of the aging of the components. While systems are now designed with a predetermined power budget, they will also have to be designed with a fixed thermal envelope. In order to fix this issue, architects first have to build reliable models able to represent both dynamic power consumption and temperature behavior of modern cores. Then they have to propose hardware/software solutions to optimize performance while respecting the thermal envelope. Such proposals might include more uniform power density distribution through the chip, but also thermally-guided dynamic activity migration.

Challenge 1.4: Design Automation for Special-Purpose Cores

Future embedded systems will take advantage of special-purpose hardware accelerators to speed up execution, and to dramatically reduce power consumption. Such accelerators can be made available as independent IP blocks or can be custom designed. A major challenge in the custom design of special-purpose cores is the fully automatic generation of the hardware and the software tools from a single architecture description or an application.

Challenge 1.5: Transparent Micro-architecture

Modern execution environments such as just-in-time compilers, code morphers, and virtualization systems rely on run-time information about the code being executed. Most processors already provide a set of performance counters that are used to steer the optimization or translation process. Given the raising popularity of this type of applications, and in order to enable more advanced optimizations, there will be a growing demand to provide more information about the dynamic processor operation. An important issue is to come up with a standardized set of performance counters in order to make the optimizations that use them more portable.

Challenge 1.6: Software-Controlled Reconfiguration

Cores should provide a number of controls to the compiler to allow the latter to better control the detailed operation of the processor (e.g. the ability to power down particular components of the processor). The compiler has often a better view on the behavior of a program than the core executing it (e.g. it has information about the type of algorithm, the memory usage, the amount of thread-level parallelism). By allowing the compiler to adapt or reconfigure the core to the needs of the application, a better performance/Watt ratio can be obtained.

Challenge 1.7: Reliability and Fault Tolerance

Electronic circuit reliability is decreasing as CMOS technology scales to smaller feature sizes. Single event upsets will soon become a common phenomenon instead of being extremely rare. Furthermore, permanent faults can occur due to device fatigue and other reasons. Functionality must be added to the cores that allow them to operate in the presence of transient and permanent faults perhaps with degraded performance.

Challenge 1.8: Security

By putting multiple cores on one chip, security is getting increasingly important for single cores. Hardware protection mechanisms are needed to help the software staying secure and to prevent against on-chip attacks like denial-of-service attacks against cores, the exploitation of hidden channels leaking information between cores, etc.

Challenge 1.9: Virtualization

Virtualization is a technology that will gain importance. Hardware support is needed to keep the virtualization layer slim, fast and secure. For some types of applications, strong performance isolation guarantees will be required between multiple containers.

2 Multi-core Architecture

A multi-core architecture is a MIMD (multiple-instruction multiple-data) multiprocessor using the terminology that has been prevailing for many decades. In the last decade, chip multiprocessing (mostly heterogeneous, up to 6-8 cores) has been commonly used in embedded SOCs, thus anticipating some of the trends that have since then been adopted also by mainstream general-purpose processors. However, the ad-hoc programmability of such embedded system has been far from satisfactory, and we now have enough transistors to integrate even more complex cores on a single chip. Envisioning a multi-core microprocessor with 256 cores by 2015, several opportunities and system challenges arise at the architecture level. Multi-core challenges are identified at the hardware and the software level. Hardware challenges are discussed in this section, the software challenges in the sections on programming models and compilation.

Challenge 2.1: Hardware Support for Parallel Programming

When moving on the multi-core roadmap, at some point traditional software-based synchronization methods will no longer be feasible and new (hardware-based) methods will have to be introduced. Transactional memory is one candidate, but it is probably just the initial approach. In fact, the hardware/software interface, i.e., the instruction-set architecture has more or less stayed unaltered for several decades. An important challenge is to understand which hardware/software abstraction can enhance the productivity of parallel software development and then find suitable implementation approaches to realize it. In fact, the abundance of transistors available in the next decade can find good use in realizing enhanced abstractions for programmers.

Challenge 2.2: On-Chip Interconnects and Memory Subsystem

The critical infrastructure to host a large core count (say 100-1000 cores in ten years from now) consists of the on-chip memory subsystem and network-on-chip (NoC) technologies. Scaling these subsystems in a resource-efficient manner to accommodate the foreseen core count is a major challenge. According to ITRS, the off-chip bandwidth is expected to increase linearly rather than exponentially. As a result, a high on-chip cache performance is crucial to cut down on bandwidth. However, we have seen a diminishing return of investments in the real-estate devoted to caches, so clearly cache hierarchies are in need of innovation to make better use of the resources.

Challenge 2.3: Cache Coherence Schemes

At the scale of cores that is foreseeable within the next decade, it seems reasonable to support a shared memory model. On the other hand, a shared memory model requires efficient support for cache coherence. A great deal of attention was devoted to scalable cache coherence protocols in the late 80s and the beginning of the 90s and enabled industrial offerings of shared memory multiprocessors with a processor count of several hundred, e.g., SGI Origin 2000. More recently, the latency/bandwidth trade-off between broadcast-based (snooping) and point-to-point based (directory) cache coherency protocols has been studied in detail. However, now that we can soon host a system with hundreds of cores on a chip, technological parameters and constraints will be quite different. For example, cache-to-cache miss latencies are relatively shorter and the bandwidth on-chip is much larger than for the "off-chip" systems of the 90s. On the other hand, design decisions are severely constrained by power consumption. All these differences make it important to revisit the design of scalable cache coherence protocols for the multi-cores in this new context.

Challenge 2.4: Hardware Support for Heterogeneity

Multiple heterogeneous cores have their own design complexity issues, as special-purpose cores have significant impact on the memory hierarchy of the system, and require specially designed communication protocols for fast data exchange among them. A major challenge is the design of a suitable high-performance and flexible communication interface between less traditional computing cores (e.g. FPGAs) and the rest of the multi-core system.

Challenge 2.5: Hardware Support for Debugging

Debugging a multi-core multi-ISA application is a complex task. The debugger needs to be both powerful and must cause very low overhead to avoid timing violations and so-called Heisenbugs. This is currently a big problem for existing debuggers, since providing a global view of a multi-core machine is virtually impossible without specialized hardware support. Much more so than a classic single-core device, multi-core chips have to be designed to support debugging tools. The proper hardware support is needed to non-intrusively observe an execution, to produce synchronized traces of execution from multiple cores, to get debug data into and out of the chip.

3 Interconnection Networks

Bus-based interconnects, like PCI or AMBA have been the dominant method for interconnecting components. They feature low cost (simple wires), convenience in adding nodes, and some reliability advantages (no active elements other than end-nodes). However, they do not scale to high performance or to high number of nodes as they are limited by higher parasitic capacitance (multiple devices attached to the wires), arbitration delay (who will transmit next), turn-around overhead (idle time when the transmitting node changes, due to the bidirectionality of the medium), and lack of parallelism (only one transmission at a time).

Point-to-point links do not suffer from arbitration delays or turn-around overheads. For external connections, high-speed serial link technology has advanced and offers single link performance at the level of 6.25 GBaud in the immediate future, making it the preferred solution to support high throughput applications. Similarly, on-chip networks can use transmission lines and compensation or retiming techniques to minimize timing skew between long parallel wires, enabling the use of wide paths. To further increase aggregate system throughput, multiple links interconnected via switches offer parallelism, and hence even higher performance. The switch is becoming the basic building block for wired interconnections, much like the microprocessor is for processing and the RAM is for memory. Finally, embedded system components are slowly turning to packet-based transfers, aiding in this way the convergence to switched-based interconnects. Lately memory chips such as FBDIMM have appeared and directly support packet-style transfers.

Future on-chip interconnects will connect hundreds of nodes (processors, memories, etc) reaching the realm of today's off-chip networks. To build these networks we can use today's ideas and techniques, adapting them to the requirements and technologies of future embedded systems. The new applications, requirements, level of integration and implementation technologies will also open new possibilities for innovation and fresh ideas. In this context, the challenges for the near future are the following:

Challenge 3.1: Interconnect Performance and Interfaces

Increasing levels of functionality and integration are pushing the size and the performance requirements of future embedded systems to unprecedented levels. Networks-on-chips will need to provide the necessary throughput at the lowest possible latency. Besides implementation technology innovations, important research areas include the long-standing, deep and difficult problems of interconnection network architecture: lossless flow control and congestion management. Recent research results in these areas have shown good progress: regional explicit congestion notification (RECN), and hierarchical request-grant-credit flow control. New research directions are opening to fully exploit the implementation technologies, for example techniques to exploit the different characteristics of multiple metal layers to provide links with shorter latencies.

Network interface design and its related buffering issues are also important for the system-level performance and cost. The simple bus interfaces of the past are rapidly evolving to full-fledged, tightly coupled network interfaces. To improve end-to-end application throughput, we need both a new breed of simplified protocol stacks, and analogously architected network interfaces. The solutions may include key research findings from the parallel computing community: user-level protected communication protocols, network interface virtualization, and communication primitives for remote DMA and remote queues.

Challenge 3.2: Interconnect Power Consumption and Management

Meeting the required interconnect performance at the lowest power consumption is becoming increasingly important as the technology moves into the nanometer scale. Power consumption is affected by many interconnect design parameters such as implementation technology, link driver design, network topology, congestion and buffer management. For example, power consumption for off-chip networks is dominated by chip crossings, suggesting higher-radix switches for lower power consumption. In addition, power management functionality will extend from the processing nodes to the system level, creating a need for NoC power-management features and protocols. These features will enable performance for power dissipation trade-offs according to system-level processing requirements. Such features can also be used for thermal management that is also becoming important in sub-micron technologies.

Challenge 3.3: Quality of Service

Embedded systems are often real-time systems, and in these cases, the interconnection network has to provide guarantees in communication parameters such as bandwidth and latency. Predicable interconnect behaviour is cornerstone to providing these guarantees, and Quality of Service (QoS) differentiation can be the vehicle towards this goal. The requirements can vary greatly from best effort, soft- and hard-real time applications, and the entire range should be supported in the most uniform and transparent way to aid component reuse. Effectiveness in providing these guarantees (for example the size of buffers that hold low priority traffic) is also an important issue as it directly influences the interconnect cost. A similar need for QoS is created by resource virtualization, where a single system (even a single processor) is viewed as a set of virtual entities that operate independently and must have a fair access to the network resources. This can be achieved either through a physical partitioning of the network, or by virtualizing the network itself using the traditional QoS mechanisms.

Challenge 3.4: Reliability and Fault Tolerance

Single event upsets will introduce uncertainties even in fully controlled environments such as on-chip networks. While traditional networking solutions exist for dealing with transmission errors, they often come at a significant implementation cost. Efficient protocols that expose part of the necessary buffering to the application and possibly to the compiler in order to jointly manage the required space can offer an efficient solution to this problem. To deal with permanent faults techniques such as automatic path migration and network reconfiguration can be used. However, errors can affect not only the data links and switches, but also the network interface logic which needs to be designed to tolerate this type of upsets.

Challenge 3.5: Interconnect Design Space Exploration

To explore the large interconnect design space, and to create efficient interconnect topologies, while at the same time providing support and guarantees for Quality-of-Service requirements is a complex, time-consuming and error-prone process. Many individual aspects of this process can however be automated and provide feedback to the designer. However, there is a lack of integrated tool-chains that will operate from the requirement level allowing early design-space exploration, all the way to the implementation dealing with all the intermediate parameters. Such tool-chains will improve productivity and allow for faster and smaller designs and faster system verification.

Challenge 3.6: Protection and Security

Embedded systems traditionally have been “flat” systems with direct control of all resources aminimal – if any – protection domains. The increase in the number of nodes, the need for programmability and extensibility, and the ever-increasing complexity are creating the need for support of protected operation. This functionality can be implemented in the network interfaces but is an integral part of and has to be designed in coordination with the overall system-level protection architecture. The system also needs modularity to support virtualization features. At the next level is the interconnection of systems and devices, where there is a need for secure communications.

4 Programming Models and Tools

Exploiting the parallelism offered by a multi-core architecture requires powerful new programming models. The programming model has to allow the programmer to express parallelism, data access, and synchronization. Parallelism can be expressed as parallel constructs and/or tasks that should be executed on each of the processing elements. The data access and the synchronization models can be distributed – through message passing – or can be shared – using global memory.

As a result, the programming model has to deal with all those different features, allowing the programmer to use such a wide range of multiprocessors, and their functionality. At the same time, the programming model has to be simple for the programmers, because a large majority of them will now be confronted with parallel programming. Therefore, to a certain extent, the simplicity of parallel programming approaches is becoming as important as the performance they yield. For the programming of reconfigurable hardware, a combination of procedural (time) and structural (space) programming views should be considered. Debuggers, instrumentation, and performance analysis tools have to be updated accordingly to support the new features supported by the programming model. This is important to reduce the time to market of both run-time systems, and applications developed on multi-core processors.

Challenge 4.1: Passing More Semantics

A first challenge is how to get the correct combination of programming constructs for expressing parallelism. Most probably, they will be taken from different programming paradigms. OpenMP 3.0 will incorporate the task concept, and with it, it will be easy to program in a pthreads-like way without the burden of having to manually outline the code executed in parallel with other tasks. Incremental parallelization will be also possible, as OpenMP already allows it. Along with this, new approaches at the higher level will include techniques from the productivity programming models area: The definition of ”places” (X10), ”regions” (Fortress), ”locales” (Chapel) or addressable domains from the language perspective, allowing to distribute the computation across a set of machines in clustered environments; Futures (X10, Cilk), allowing the execution of function calls in parallel with the code calling the function.

Challenge 4.2: Transparent Data Access

A second challenge is to build a programming model that allows the programmer to transparently work with shared and distributed memory at the same time. Current attempts, like Co-Array Fortran, UPC, X10, Fortress, Chapel..., still reflect in the language the fact that there are such different and separate addressable domains. This interferes with data access and synchronization, because depending on where the computation is performed, different ways to access data and synchronization must be used. As hardware accelerators can also be seen as different execution domains with local memory, it is interesting to note that solving this challenge will also provide transparent support to run on accelerators (see also Challenge 5.4).

Challenge 4.3: Adaptive Data Structures

An observation is that at the low level all code is structured as procedures: (i) programmers break the different functionality they put in the application as subroutines or functions; (ii) parallelizing compilers outline as a subroutine the code to be executed in parallel; (iii) even accelerators can be used through a well-defined procedure interface, hiding the details of data transfer and synchronization; and (iv) most hardware vendors already provide libraries with optimized primitives in the form of procedures. But there is no such mechanism for data structures. A mechanism is needed to allow the compiler and the run-time system to tune – optimize – data structures, adapting them to the execution conditions. In such a way that a data structure can be automatically distributed in a cluster or accessed by a set of accelerators, while all data transfer involved is managed by the run-time system. Knowing the restrictions on the arguments of the procedures (atomicity, asynchrony ...) will also be needed to ensure correct data transfers and manipulation.

Challenge 4.4: Advanced Development Environments

An easy to program multi-core system requires sophisticated support for threading management, synchronization, memory allocation and access. When different threads run different ISA's, a single debugging session must show all types of machine instructions, and the information related to variables and functions, and must be able to automatically detect non-local bugs like race conditions, dangling pointers, memory leaks, etc. Debugging a multi-core system running hundreds of threads is a major unsolved challenge, which will require hardware support in order to be effectively solved.

Challenge 4.5: Instrumentation and Performance Analysis

Tools obtaining run-time information from the execution are essential for performance debugging, and to steer dynamic code translation (Just-in-Time compilation, code morphing,. . .). Hardware designs must take observability into consideration. The amount of information that can possibly be generated by a multi-core system is however overwhelming. The challenge is to find techniques to efficiently analyze the data (e.g. searching for periods or phases), to significantly reduce the amount of data, and to find effective ways to conveniently represent the data generated by hundreds of threads.

5 Compilation

Modern hardware needs a sophisticated compiler to generate highly optimized code. This increasing rate of architectural change has placed enormous stress on compiler technology such that current compiler technology can no longer keep up with architectural change. The key point is that traditional approaches to compiler optimizations are based on hardwired static analysis and transformation which can no longer be used in a computing environment that is continually changing. What is required is an approach which evolves and adapts to applications and architectural change along the multi-core roadmap and takes into account the various program specifications: from MATLAB programs to old already parallelized code. For future multi-core based high-performance embedded systems, the following challenges are identified.

Challenge 5.1: Automatic Parallelization

Automatic parallelization has been quite successful in the past for scientific programs, array-based programming languages (FORTRAN), and for homogeneous shared memory architectures. This work has to be extended to a much wider set of application types, to pointer-based programming languages, and to a wide variety of potentially heterogeneous multi-core processors with different memory models. This requires the development of new static analysis techniques to analyze pointer-based programs (maybe already parallelized) and manage the mapping of memory accesses to systems without explicit hardware-based shared memory. By providing hints or assertions, the programmer might be able to considerably improve the parallelization process.

It will incorporate speculative parallelization to extract greater levels of parallelism. Furthermore by adding certain architectural features, the compiler could communicate its assumptions at run-time and enable the violations to be detected by the hardware, causing more optimal overall program execution. Examples of these techniques include speculative load instructions, and speculative multithreading. These ideas enable the compiler to make better optimization choices without over-burdening the hardware with complexity. Finally, speculative parallelization can be combined with dynamic optimization such that as the program evolves in time, the (just-in-time) compiler can learn about the relative success of speculation and dynamically recompile the code accordingly.

Challenge 5.2: Automatic Compiler Adaptation

Tuning the optimization heuristics for new processor architectures is a time-consuming process, which can be automated by machine learning. The machine learning based optimizer will try many different transformations and optimizations on a set of benchmarks recording their performance and behavior. From this data set it will build an optimizing model based on the real processor performance.

This approach can also be used for long running iterative optimization where we want to tune important libraries and embedded applications for a particular configuration. Alternatively, it can be used by dynamic just-in-time compilers to modify their compilation policy based on feedback information. In addition, if we wish to optimize for space, power and size simultaneously we just alter the objective function of the model to be learned and this happens automatically.

Challenge 5.3: Architecture/Compilation Cooperation

The role compilation will have in optimization will be defined by the architectural parameters available to it. Longer term work will require strong compiler/architecture co-design opening up the architecture infrastructure to compiler manipulation or conversely passing run-time information to the architecture to allow it to best use resources.

This is part of a larger trend where the distinction between decisions currently made separately in the compiler and in the hardware is blurred. If the compiler has exact knowledge of behavior within an up-coming phase of a program (a so-called scenario), then the hardware should be appropriately directed. If, however, analysis fails, then the hardware should employ an appropriate general mechanism possibly aided by hardware-based prediction. In between these two extremes, the compiler can provide hints to the hardware and can modify its behavior based on run-time analysis.

Challenge 5.4: Mapping Computations on Accelerators

Some approaches already offer access to accelerators through well defined interfaces, thus summarizing computation, data access and synchronization on a single procedure call. This challenge seeks to enable the compiler to automatically detect and map parts of the application code to such "accelerated" routines. This may be easy for well-known procedures from the HPC environment, like FFT or DGEMM, but there is no general solution yet for general application code. The problem is especially challenging for less conventional computing nodes such as FPGAs.

Challenge 5.5: Power-Aware Compilation

As the demand for power efficient devices grows, compilers will have to consider energy consumption in the same way space and time are considered now. The key challenge is to exploit compile-time knowledge about the program to use only those resources necessary for the program to execute efficiently. The compiler is then responsible for generating code where special instructions direct the architecture to work in the desired way. The primary area of interest of such compiler analysis is in gating off unused parts or in dynamically resizing expensive resources with the help of the run-time system. Another compiler technique for reducing power is the generation of compressed or compacted binaries which will stay important in the embedded domain.

Challenge 5.6: Just-in-Time Compilation

Given the popularity of programming languages that use just-in-time compilation (Java and C# being very popular programming languages of the moment), more research is needed in Just-in-Time compilation for heterogeneous multi-core platforms. Since the compilation time is part of the execution time, the optimization algorithms have to be both accurate and efficient. The challenge in Just-in-Time compilation for heterogeneous multi-cores is not only to predict (i) when to optimize, (ii) what to optimize and (iii) how to optimize, but also (i) when to parallelize, (ii) what to parallelize, and (iii) how to parallelize. Appropriate hardware performance counters or possibly additional run-time information can help making these decisions.

Challenge 5.7: Full System Optimization

Given the component-based nature of modern software, the massive use of libraries, and the widespread use of separate compilation, run-time optimization, and on-line profiling, no single tool has an overview of the complete application, and many optimization opportunities are left unexploited (addresses, function parameters, final layout,...). One of the challenges in full system optimization is to bring the information provided at all these levels together in one tool, and then to use this information to optimize the application. This will enable cross-boundary optimization: between ISAs in a heterogeneous multi-core, between a processor and reconfigurable hardware, between the application and the kernel. An underlying technical challenge is often the design of scalable algorithms to analyze and optimize a full system.

6 Run-Time Systems

The run-time system aims at controlling the environment of the embedded system during system operation. It is concerned mainly with issues related to dynamic behavior that cannot be determined through static analysis (by the compiler). The run-time system consists of a collection of facilities, such as dynamic memory allocation, thread management and synchronization, middleware, virtual machines, garbage collection, dynamic optimization, just-in-time compilation and execution resources management.

The current trend in embedded systems is to customize automatically or by hand the operating systems developed for general-purpose platforms. There is a large opportunity for improving operating system and run-time performance via hardware support for certain features and functions (e.g., fault management and resource monitoring). Operating system and runtime research should be more closely coupled to hardware research in the future in order to integrate multi-core heterogeneous systems (medium term challenges) and to seamlessly support dynamic reconfiguration and interoperability (long term challenges).

Challenge 6.1: Execution Environments for Heterogeneous Systems

Runtimes and operating systems have to be aware that the architecture is a heterogeneous multi-core. Currently, most specific accelerators are considered as devices or slave coprocessors. They need to be treated as first class objects at all levels of management and scheduling. The scheduler will map fine grain tasks to the appropriate computing element, being processors with different ISA or even specific logic engines. Memory will be distributed across the chip, so the runtime needs to graciously handle new local storages and sparse physical address spaces. Support for code morphing should be integrated. Hardware and software will be co-designed, and compilers should generate multiple binary versions for the software to run on and control the multiple cores.

Challenge 6.2: Power Aware Run-Time Systems

Allocation of resources should take energy efficiency into account, for example the fair allocation of battery resources rather than just CPU time. Operating system functionalities and policies should consider: disk scheduling (spin down policies), security (adaptive cryptographic policy), CPU scheduling (voltage scaling, idle power modes, moving less critical tasks to less power-hungry cores), application/OS interaction for power management, memory allocation (placement, switch energy modes), resource protection/allocation (fair distribution, critical resources) and communication (adaptive network polling, routing, and servers).

Challenge 6.3: Adaptable Run-Time Systems

The run-time systems should (semi-)automatically adapt to different heterogeneous multi-cores based on the requirements of the applications, available resources, and scheduling management policies. However, static application-specific tailoring of operating systems is not sufficient. An adaptable operating system is still tailored down for specific requirements, but can be reconfigured dynamically if the set of required features changes. We will need new run-time systems that leverage the OS modularity and configurability to improve efficiency and scalability, and provide support to new programming models or compilers that exploit phase and versioning systems. Reliability, availability and serviceability (RAS) management systems have to work cooperatively with the OS/Runtime to identify and resolve these issues. Advanced monitoring and adaptation can improve application performance and predictability.

Challenge 6.4: Run-Time Support for Reconfiguration

Run-time support for reconfigurable (RC) applications is hampered by the fact that the execution model of dynamically reconfigurable devices is a paradigm currently not supported by existing run-time systems. Research challenges include reconfigurable resource management that deals with problems such as reconfiguration time scheduling, reconfigurable area management and so on. The need for real-time, light-weight operating systems support on RC platforms is also emerging. The problem of multi-tasking in dynamically reconfigurable RC context is largely unsolved. Transparent hardware/software boundaries are envisioned: a task can be dynamically scheduled in hardware or in software by the run-time system, while the rest of the system does not need to be aware of such activities. This challenge is about target-architecture and technology dependent automated run-time system customization needed to allow fine-tuning of RC-aware run-time systems.

Challenge 6.5: Fault Tolerance

Providing new levels of fault tolerance and security in an increasingly networked embedded world by taking advantage of large degrees of replication and by designing fence mechanisms in the system itself (as opposed at the application level). Moreover, besides providing novel techniques that will increase system reliability and security, research should consider methods for allowing users/programmers to specify their intentions in this domain. Self-diagnosis mechanisms should detect and repair defective components while maintaining the system operation.

Challenge 6.6: Seamless Interoperability

Due to the very nature of embedded applications (communication with the environment is a key part in embedded computation) many existent embedded devices are somehow (wired or wireless) connected. Mobility suggests support for wireless connectivity, and in the future the majority of the embedded systems platform will have a connection either with other similar devices through small area networks or with bigger infrastructures via the Internet or other private networks. Future embedded systems devices will gain enhanced new capabilities using the networks: they will be able to upgrade themselves (automatically or semi-automatically), maintain consistency with other devices (PCs, laptops), and perform backups and other useful administrative tasks. At least a dozen wireless networks along with their protocols and interfaces are available and cover the wide range from personal- and small-area to medium and long distance networks, offering different trade-offs in features, throughput, latency and power consumption. The challenge towards this goal is the seamless functional integration of vastly different communication media and methods, and the automatic adaptation to network performance both at the level of interconnect management as well as the application behaviour.

7 Benchmarking

The design of high-performance processor architecture is a matter of trade-offs, where a sea of design options is available, all of them having some performance/power/area impact. In order to reliably compare two design points, we must use realistic applications. Such is the role of a benchmark suite, which is a set of representative applications.

First of all, it is getting increasingly difficult to find useful benchmarks: realistic applications are generally carefully protected by companies. Even inside companies, strict Intellectual Property (IP) barriers isolate the application development groups from the hardware development groups, who only get a vague description of the real application needs. And then again, another set of IP barriers isolate the hardware development groups from the research groups, who only get access to distorted versions of the applications, or synthetic kernels. In the absence of real applications, academia generally relies on micro-benchmarks (kernels), synthetic applications or the widely spread SPEC benchmark suite. However, they only model particular aspects of an application.

Secondly, different application domains have different computing requirements, and hence need their own benchmarks. There are already a number of benchmark suites representing different application domains: SPEC for CPU performance, Mediabench for multimedia, TPC-H, TPC-C for databases, etc. The EEMBC embedded benchmark is already composed of different suites. As new application domains emerge, new benchmarks will have to be added. Besides the application domain, the software implementation technology is also changing rapidly. Virtualization is gaining popularity, and an increasing number of applications are being deployed for managed execution environments like the Java virtual machine or .NET. Currently, there are very few good benchmarks in this area.

Finally, the hardware is also evolving. Old benchmarks are not suited anymore to evaluate hardware extensions like MMX, SSE, Altivec, or 3DNow and hardware accelerators in general (including FPGAs). Since these hardware accelerators are getting increasingly common in heterogeneous multi-cores, there is a huge need for realistic benchmarks that can exploit these (parallel) hardware resources.

In the benchmarking domain, there are three major challenges:

Challenge 7.1: Multi-core Benchmarks

Multi-cores (and especially heterogeneous multi-cores) are less standardized than general-purpose processors. They can contain several special-purpose hardware accelerators, and even reconfigurable components. Benchmarking such architectures against one another is very difficult because there are almost no benchmarks available that fully exploit the resources of such a platform (computing cores, but also the interconnections and memory hierarchy). This challenge aims at creating a set of representative benchmark applications for this type of processing architectures.

Challenge 7.2: Synthetic Benchmarks

Creating a good benchmark is a tough job. First one has to get full access to a realistic, IP-free, relevant and representative application, and then this application has to be adapted in order to make it platform independent. This challenge aims at transforming an existing application into an IP-free and platform independent synthetic benchmark with a given execution time. Such a methodology would significantly facilitate the exchange of code between (departments of) companies, or between companies and academia. The fact that the execution time of the synthetic benchmark can be chosen, will have a dramatic effect on simulation time. One particularly challenging task is to come up with a synthetic benchmark that models the behavior of an application that runs in a virtual machine.

Challenge 7.3: Benchmark Characterization

Traditionally, benchmarks are divided in application domains: floating-point (scientific) applications, CPU-intensive applications, media processing, digital signal processing kernels. Recently, some application domains have been added: bio-informatics, gaming, software-defined radio,... There is no solid methodology to characterize all these different types of benchmarks in such a way that the characterization could be used to steer the selection of benchmarks or eventually prune the system design space. More research is needed to come up with (mathematical) models describing benchmark behavior – including benchmarks that run in a virtual machine.

8 Simulation and System Modelling

Simulation technology and methodology is at the crux of computer architecture research and development. Given the steadily increasing complexity of modern computer systems, simulation has become the de facto standard for making accurate design trade-offs efficiently.

Simulation technology and methodology need to meet high standards. Computer architects want simulation software to be fast, accurate, modular, extensible, etc. We expect these requirements to simulation software to expand drastically in the multi-core era. We observe at least five major challenges for the near future.

Challenge 8.1: Simulator Building

Being able to quickly build simulators by reusing previously built modules is a first key challenge for future simulation technology. Too often until now, computer architects have been building monolithic simulators which are hard to reuse across projects. Modular simulators on the other hand, which provide an intuitive mapping of hardware blocks to software modules, enable the easy exchange of architecture components.

Challenge 8.2: Simulation Modeling Capabilities

A related issue is that simulation software should be extensible in the sense that novel capabilities of interest that crosscut the entire simulator should be easy to add in a plug-and-play fashion. Example capabilities are architectural power/energy modeling, temperature modeling, reliability modeling, etc. In addition, the modeling itself needs further research.

Challenge 8.3: Simulation Speed

Simulation speed is a critical concern to simulation technology. Simulating one second of real hardware execution takes around one day of simulation time, even on today's fastest simulators and machines. And this is to simulate a single core. With the advent of multi-core systems, the simulation time is expected to increase more than proportional with the number of cores in a multi-core system. Exploring large design spaces with slow simulation technology obviously is infeasible. As such, we need to develop simulation technology that is capable of coping with the increased complexity of future computer systems.

There are a number of potential avenues that could be walked for improving simulation speed. One avenue is to consider modeling techniques, such as analytical modeling and transaction-level modeling, which operate at a higher level of abstraction than the cycle-by-cycle models in use today. A second avenue is to study techniques that strive at shortening the number of instructions that need to be simulated. Example techniques are sampled simulation and statistical simulation; these techniques are well understood in the uniprocessor domain but need non-trivial extensions to be useful in the multiprocessor domain. A third avenue is to parallelize the simulation engine. This could be achieved by parallelizing the simulation software to run on parallel (multi-core) hardware, or by embracing hardware acceleration approaches using for example FPGAs for offloading (parts of) the simulation. A fourth avenue consists in building slightly less accurate but very fast architecture models using machine-learning techniques such as neural networks.

Challenge 8.4: Design Space Exploration

The previous challenge concerned the simulation speed in a single design point. However, computer architects, both researchers and developers, need to cull large design spaces in order to identify a region of interesting design points. This requires running multiple simulations. Although this process is embarrassingly parallel, the multi-billion design points in any realistic design space obviously make exhaustive searching infeasible. As such, efficient design space exploration techniques are required. The expertise from other domains such as operational research, data mining and machine learning could be promising avenues in the search for efficient but accurate design space exploration.

Challenge 8.5: Simulator Validation

As alluded to before, designing a computer system involves modeling the system-under-design at different modeling abstractions. The idea is to make high-level design decisions using high-level abstraction models that are subsequently refined using low-level abstraction models. The key point however is to validate the various abstraction models throughout the design cycle to make sure high-level design decisions are valid decisions as more details become available throughout the design process. Since this already is an issue for a uniprocessor design flow, it is likely to increase for multi-core designs where high-level abstraction models will be used to cope with the simulation speed problem.

9 Reconfigurable Computing

Reconfigurable computing (RC) is becoming an exciting approach for embedded systems in general, and for application-specific designs in particular. The main advantage of such systems is that they can adapt to static and dynamic application requirements better than those with fixed hardware. Furthermore, the power/performance ratio for reconfigurable hardware is at least an order of magnitude better than that of general-purpose processors. However, for RC technology to be deployed on a large scale, a number of (but not limited to) important gaps in theory and in practice have to be bridged.

Challenge 9.1: Application Domain Extension

Currently, RC technology has much success for selected applications in networking, communications, and in defense where cost is not a limiting factor. One future challenge here is to combine high-performance with high reliability, durability and robustness, by exploiting the properties inherent to RC devices. Future research on RC is expected to enable new application domains: medical and automotive (reliability and safety), office applications (scientific, engineering and media processing), and high-end consumer electronics (trade-offs in speed/size/power/cost/development time). A set of relevant (realistic) benchmarks is needed for analysis and comparison investigations.

Challenge 9.2: Improved Run-Time Reconfiguration

Techniques for supporting fast and reliable reconfiguration, including those for dynamic and partial run-time reconfiguration are rapidly gaining importance. Novel configuration memories that overcome soft errors are desirable; they can be used to support technologies for rapid reconfiguration, such as multi-context organization. These would be useful for designs that make use of reconfiguration to adapt to environmental changes, and for evolutionary computing applications.

Challenge 9.3: Interfacing

Techniques to enable components on an RC device to be efficiently interfaced to each other and to external devices are required. Within the RC device, improved communication bandwidth between processors, memories and peripherals will tend to reduce power and energy consumption. Dynamically adaptable network-on-chip technologies and main memory interfaces, such as packet-based routing, flux networks and memory controllers on demand will become relevant.

Challenge 9.4: Granularity

Efficient support for special-purpose units is important. Commercial FPGAs are becoming more heterogeneous, since common functional blocks such as arithmetic units or a complete DSP are hardwired to reduce area, speed and power consumption overheads associated with reconfigurability; it is useful to develop theory and practice to enable optimal inclusion of such special-purpose hardwired units in a reconfigurable fabric for given applications, to get the best trade-offs in adaptability and performance.

Understanding the pros and cons of the granularity of the reconfigurable fabric is also of particular interest. Fine-grained architectures are more flexible, but less efficient than coarse-grained architectures which have fewer reconfigurable interconnects. The challenge is to determine, for a given application, the optimal RC fabric granularity or the optimal combination of RC fabrics with different granularities. This is related to RC technologies for special-purpose units described earlier, since an array of special-purpose units with limited programmability can be seen as a coarse-grained RC array.

Challenge 9.5: Efficient Softcores

Support for efficient hardwired and softcore instruction processors, and determining, for a given application, the optimal number for each type and how they can be combined optimally will become important. One direction is to investigate facilities for instruction extension, exploiting customizability of RC technology to produce Application Specific Instruction Processors (ASIP). Multiple instructions in an inner loop can, for instance, be replaced by fewer custom instructions to reduce overhead in fetching and decoding. Another direction is to explore how processor architectures can be customized at design time and at run time, so that unused resources can be reconfigured, for instance, to speed up given tasks. Relevant topics include caching of multiple configurations, dedicated configuration memory controllers, and dedicated memory organizations.

Challenge 9.6: RC Design Methods and Tools

RC design methods and tools, including run-time support, are key to improving designer productivity and design quality; they are responsible for mapping applications, usually captured in high-level descriptions, into efficient implementations involving RC technology and architectures.

Design methods and tools can be classified as either synthesis or analysis tools. The synthesis tools are required to map high-level descriptions into implementations with the best trade-offs in speed, size, power consumption, programmability, upgradeability, reliability, cost, and development time. Their satisfactory operation relies on extending current synthesis algorithms to take, for instance, dynamic reconfigurability into account. They need to integrate with analysis tools, which characterize a design at multiple levels of abstraction, and to support verification of functionality and other properties such as timing and testability.

Even for a given application targeting specific RC systems, there is currently no coherent tool- chain that covers all the main synthesis and analysis steps, including domain-specific design capture, performance profiling, design space exploration, multi-chip partitioning, hardware/software partitioning, multi-level algorithmic, architectural and data representation optimization, static and dynamic reconfiguration mapping, optimal instruction set generation, technology mapping, floor planning, placement and routing, functional simulation, timing and power consumption analysis, and hardware prototyping.

10 Real-Time Systems

The computing requirements of real-time systems are increasing rapidly: video-coding and decoding in tv-sets, set-top boxes, DVD recorders, compute-intensive energy saving and safety algorithms in automotive, railway, and avionic applications. Processors in most current real-time systems are characterized by a simple architecture, encompassing short pipelines and in-order execution. These types of pipelines ease the computation of the worst-case execution time (WCET). Due to the use of hard-to-predict components (caches, branch predictors,...) or due to resource sharing in multi-core processors, the worst-case execution time is either (i) hard to compute, or (ii) if computed, it is way beyond the real execution time (RET). Hence, on the one hand, multi-core processors will be needed to meet the computing demand of future real-time systems, but on the other hand they also pose serious challenges for real-time applications.

Challenge 10.1: Timing-Analyzable High-Performance Hardware and Software

In soft real-time systems, the time constraints of applications are relaxed. Instead of ensuring that every instance of a task meets its deadline, a guaranteed mean performance is required. This allows more freedom in the processor design. A matter of great concern is however the parallel execution of the tasks on general-purpose hardware while ensuring the access to shared resources. Often, it is realized by specific hardware but, more and more, we would like to rely on "general-purpose" hardware to reduce costs. Thus, the aim of the research is to define how, on a multithreaded processor, the hardware can observe the behavior of the tasks in order to dynamically decide which resources (or which percentage of resources) should be devoted to each task.

On the one hand, a high resource sharing (SMT) implies that chips are smaller and that the performance per resource is higher. But also, a high resource sharing causes a high interference between threads, which causes more variable execution times. On the other hand, a reduced resource sharing (like CMPs) causes much smaller execution time variability, but implies that many hardware resources are duplicated, increasing area and cost of the chip. Hence there is a large space of architectural solutions to explore like creating private processors for time-critical processing, or providing hardware components that warn when deadlines are not met or are getting too close.

Challenge 10.2: WCET Computation for Shared Resources

Research on hard real-time architectures is also needed because some applications (mainly in automotive, aeronautics and space) render current embedded hardware solutions obsolete. Indeed, these applications will require much more performance than today while preserving the need for a static WCET analysis to determine and guarantee the maximum execution time of a task. Furthermore, this maximum execution time should not be (too much) overestimated to be useful. Thus, in addition to enhancing static analysis, there is a need for new architectural features, some of them being managed by software, which increase performance while favoring static analysis.

Challenge 10.3: Alternative Real-Time Modeling Methodologies

In absence of analyzable hardware, more sophisticated (probabilistic) models need to be developed that can model the probability of not meeting the deadline. This relaxed real-time demand only guarantees that no more than x% of all real execution times will exceed a boundary called LET (longest execution time). The LET (which can be determined through extensive simulation) should be much closer to the RET than the WCET.

Conclusion

The paradigm shift to multi-core architectures is having a profound effect on research challenges for the next decade. The exponential improvement of application performance across computer generations will come to an end if we do not succeed in adapting our applications to the parallel resources of the future computer systems. This requires a massive effort to improve all aspects of computing, as detailed in the 55 challenges in this HiPEAC Roadmap.

Acknowledgements. This work was funded by the 6th European Framework Programme (FP6), under contract no. IST-004408 High-Performance Embedded Architecture and Compilation (HiPEAC) Network of Excellence.

Compiling a roadmap and making it into a coherent document is not an easy task and builds on the expertise of many. Therefore, the authors of this research roadmap would like to thank the following people for their contributions to the roadmap effort: (in alphabetical order) Angelos Bilas (FORTH, Greece), Doug Burger (University of Texas at Austin, USA), Francisco Cazorla (UPC, Spain), Albert Cohen (INRIA, France), Sorin Cotofana (TU Delft, The Netherlands), Bruno De Bus (Ghent University, Belgium), José Duato (University of Valencia, Spain), Lieven Eeckhout (Ghent University, Belgium), Jakob Engblom (Virtutech), Paolo Faraboschi (HP), Piero Foglia (University of Pisa, Italy), Georgi Gaydadjiev (Tu Delft, The Netherlands), Marisa Gil (UPC, Spain), Christoforos Kachris (TU Delft, The Netherlands), Manolis Katevenis (FORTH, Greece), Stefanos Kaxiras (University of Patras, Greece), Kathryn McKinley (University of Texas at Austin, USA), Osman Unsal (BSC, Spain), Barbara Ryder (Rutgers University, USA), Guri Sohi (University of Wisconsin-Madison, USA), Theo Ungerer (University of Augsburg, Germany), Mateo Valero (UPC, Spain), Hans Vandierendonck (Ghent University, Belgium), David Whalley (Florida State University, USA), Sami Yehia (ARM), Ayal Zaks (IBM).