Virtualization for Morphable Multi-Cores

Panayiotis Petrides, Department of Computer Science, University of Cyprus, Nicosia, Cyprus
Frederico Pratas, INESC-ID/IST, Technical University of Lisbon, Lisbon, Portugal
Leonel Sousa, INESC-ID/IST, Technical University of Lisbon, Lisbon, Portugal
Pedro Trancoso, Department of Computer Science, University of Cyprus, Nicosia, Cyprus

Abstract

In recent years we have observed a shift in processor architectures towards chips with multiple cores, thus avoiding the power and complexity walls. The increasing number of cores is leading to major challenges such as processor configuration and management of such complex hardware. In order to achieve a better match between the hardware and the demands of different applications and their phases of execution, future processors will have to offer cores with different specifications which could even change dynamically at run-time. To address these issues we envision that for future processors the hardware will be packaged along with a virtualization layer that hides the hardware complexity and at the same time monitors the application behavior as to transparently improve its performance at run-time. The virtualization layer must be built in a modular way. Extra features, such as mechanisms to improve the execution of an application (e.g. data prefetching), and support for heterogeneous architectures (e.g. system with accelerators) are implemented into the virtualization layer as separate plug-in modules. As a case-study we present the benefits of implementing a bandwidth-aware scheduler as a mechanism of the virtualization layer.

1 Introduction

The increasing number of cores on a chip along with the fact that diverse types of cores will be available in the early future, requires a high management effort. The effort involved in porting and tuning applications for different currently available multi-core architectures is high and will increase dramatically for larger scale multi-core chips that include cores of different characteristics, which in some cases could even change their configuration dynamically at run-time. We propose that future multi-core architectures will contain processing cores and memory hierarchies that are able to change their configurations at run-time. We call these Morphable multi-cores. Manually tuning applications to such an architecture would be a time consuming effort. As such, we believe that the best solution will be for manufacturers to provide not only the hardware platform but also a virtualization layer or hypervisor which will hide the complexity and diversity of the hardware. This virtualization layer operates as the manager of the underlying Morphable multi-core, releasing the programmer from this demanding task, hiding complexity from the Operating System (OS), and also allowing that products from different manufacturers may, transparently, use different implementations. In other words, by offering this virtualization layer along with the hardware it is possible to offer a standard set of basic mechanisms transparent to the upper layers, such as resource detection and managing, thread monitoring and migration, and fault-tolerance. Moreover, these basic mechanisms can be extended to provide additional features at the virtualization level. Examples of these extended mechanisms are thread scheduling, memory prefetching, and hardware reconfiguration. For example, the scheduling mechanism could assign tasks to the available cores. This method could be as simple as just randomly distributing the threads among the different cores or as complex as making architecture-aware decisions that, based on online monitoring information of the application behavior, are able to select the best matching cores available.

2 Morphable Multi-Core Architecture

In this work, we focus on a coarse-grain reconfigurable approach as a solution to dynamically adapt the architecture according to the application requirements. The architecture may include a discrete set of configurations. In a limited reconfiguration space, a discrete set of different configurations of the cores can be supported, thus reducing the design complexity and overheads. The denomination “morphable” designates those architectures which are not fully reconfigurable but are still able to adapt at run-time.

We present a morphable architecture in Figure 1(a). The cores are grouped into three different types according to the roles they assume. The first two groups include general purpose cores for (a) parallel processing, and (b) se-
Figure 1: Overview of the proposed system: (a) Morphable Architecture and its components (MC - Memory Controller); (b) Hypervisor Architecture with the relations between the basic and extended mechanisms.

Figure 1: Overview of the proposed system: (a) Morphable Architecture and its components (MC - Memory Controller); (b) Hypervisor Architecture with the relations between the basic and extended mechanisms.

quent processing. The third group (c) is used to support the Mechanisms provided by the hypervisor [1] (see Section 3), and can be implemented either by directly using specialized hardware or on top of the available processing cores.

The amount of parallelism contained in the different applications is the main factor that lead us to distinguish between parallel and sequential processing cores. Sequential cores are designed to improve sequential execution, e.g., implementing techniques such as aggressive ILP and branch prediction, while parallel processing cores are optimized for multithreading and SIMD. The latter cores are simpler and therefore it is possible to have more of them in the same circuit area, increasing the degree of parallelism. Both types of cores have access to two different types of memories: private and shared memory, in order to reduce the overall memory latency, while making use of the memory bandwidth more efficiently. Moreover, groups of cores can be served by different memory controllers in order to further increase the effective memory bandwidth.

The proposed virtualization layer offers specialized mechanisms that support the reconfiguration of the hardware. Since the proposed virtualization layer is implemented in a modular way, in addition to the examples presented below other mechanisms can be integrated without the need of a complete redesign.

2.1 Computational Element Reconfiguration

Different configurations of the architecture are obtained by performing reconfiguration at different levels: at the lower-level it can use hardware components that are reconfigurable, such as computational elements that may be merged [2]; and at a higher-level the voltage-frequency of the cores may be changed, or some cores may be switched off [3]. For example, if a massively parallel application is detected by the system, using the low-level reconfiguration allows the system to configure the hardware to offer as much parallelism as possible by having simpler processing cores. If there is the need for accelerating a specific computationally intensive thread, the cores can be “merged” to combine their processing power. Adapting the frequency and the number of active cores at run-time according to the processing workload allows the processor to balance power consumption, performance, and thermal efficiency.

2.2 On-chip Memory Hierarchy Reconfiguration

The design of the memory hierarchy takes into account different factors. A configuration may include large memory modules shared among a large number of cores. This helps in the sharing but may lead for contention overheads. Distributing this memory modules may solve the contention, but may lead to increased coherence traffic. As these trade-offs depend from application to application, the architecture should be reconfigured dynamically as to provide a better match to the application demands. Memory elements may change size and organization [4], and the system may identify and distinguish private from shared data. This allows for private data not to generate useless coherence traffic and shared data to be placed in modules that are smaller thus also resulting in better performance.

2.3 Memory Controller Reconfiguration

The increasing number of cores on a chip leads to memory bandwidth limitations. We propose to reconfigure, at run-time, the assignment of memory controllers to the different cores according to the application’s demands. For example if an application running on one core requires high memory bandwidth a controller can be assigned to serve it, while the other cores are served by the remaining.

3 Virtualization Layer

In this work we focus on Bare-metal Virtualization [5] and how its mechanisms and features can be used in order to
provide portability among different hardware configurations, while at the same time with low overhead for applications’ execution. More specifically, we propose a hypervisor that abstracts the underlying hardware of the system from the OS and application layer.

The functionality offered by the hypervisor is transparent to the user and OS. The hypervisor has built-in mechanisms to monitor the behavior of the applications and thus acts according to this information. In addition, it provides a higher-level interface that allows the user and/or OS to control the hypervisor’s mechanisms. For example, the user can fine-tune applications performance by exploiting system’s hardware characteristics at a more fine-grain level by triggering reconfiguration or assigning the threads to specific resources. Also, the OS can use the offered features in order to improve scheduling decisions.

3.1 Hypervisor Mechanisms

The functionalities of the hypervisor are implemented as a set of mechanisms, as shown in Figure 1(b). By mechanisms we define the basic functionalities provided by the hypervisor in order to abstract the architecture to the upper layers.

The Resource Detection Mechanism (RDM) identifies the underlying hardware in terms of computational and memory capabilities. Through this mechanism, the hypervisor is aware of the current configuration of the underlying hardware and its performance. Typical information collected by this mechanism includes the operating frequency, temperature, memory cache configurations, peak performance, idle time, type and number of issued instructions, number of misses and other characteristics for each core. This information can be gathered through performance counters [6] of the system. Moreover, it is initially collected at boot time and updated at regular intervals. It is stored into the corresponding Resource DataBase (RDB), were it remains available to the other mechanisms. The Core Managing Mechanism (CMM) supports the high level reconfiguration of the system. It is used to change parameters such as the operating frequency and voltage, and to switch on or off cores. These changes are also recorded in the RDB.

The Threads Monitoring Mechanism (TMoM) is responsible for monitoring the performance requirements of the different threads, by gathering statistics, such as memory bandwidth utilization. Online monitoring techniques [7, 8] focus on monitoring the execution of an application through the hardware counters, depending on the hardware these are usually individual counters per active thread. The hypervisor organizes the information for every interval of a given number of instructions in a thread, also named a phase, and stores the main statistics obtained (e.g. CPI, cache misses, and bandwidth) for the N last execution phases in the Thread Database (TDB). Notice that in addition to the local cache information, the TMoM needs also to monitor the cache coherence traffic and memory access patterns to identify possible access sharing and conflict patterns [9]. The Threads Migration Mechanism (TMiM) is essential for adapting the workloads to the underlying architecture, and is used for example by the scheduler. The TMiM mechanism implements efficient schemes for migrating threads between cores. Thread migration from one core to another should occur with the minimal performance overhead (which maybe occur due to data transfer from the already executing core to the new assigned core, or by copying the memory pages between memory banks assigned to different controllers) and be transparent to the other threads, in order not to influence their execution.

3.2 Hypervisor Extension Mechanisms

In addition to the four basic mechanisms described above, the virtualization layer can implement additional mechanisms as to provide new features. We call them Extension Mechanisms as they use the features of the basic mechanisms.

The Power Manager Mechanism (xPMM) is responsible for maintaining the system power consumption at a certain level by considering different modes, such as: low power, balanced execution, or best performance. This mechanism uses the global information available in the RDB to adjust the power consumption. It performs decisions like changing the frequency of certain cores or turning them on/off, similarly to the POWER6 [3].

Many applications have irregular memory access patterns that can not be captured neither by the compiler nor by prefetching engines. This fact justifies the implementation of a Prefetching Mechanism (xPRE). A careful monitoring of the data accesses may allow to identify non trivial memory access patterns. These access patterns are stored as streams of data. Based on these streams, an intelligent prefetching identifies irregular memory access patterns and triggers the prefetching service in order to increase applications performance. Thus we implement a form of Correlation Prefetching [10]. This prefetching is done transparent to the OS and applications. Information available in the TDB, such as the memory access patterns, triggers the prefetching mechanism.

As previously mentioned, one limitation for the scalability of multi-core processors is the increasing traffic to maintain data coherence in the multiple memory modules within the processor. One architectural optimization is to provide separated memory modules for shared and private data. For such an architecture, the extension Data Split Mechanism (xDAT) gathers information from the TDB, such as the memory sharing patterns, and then dynamically assigns the data to private or shared memory modules. This is also transparent to the application and OS and with the minimum possible overhead. Another alternative to reduce the coherence overheads is to have an architecture that provides only private memory modules. In this case, cache-coherence is provided to the upper levels (OS and application) by implementing the Coherence Mechanism (xCOH) on the virtualization layer. This is efficient enough to also issue coherence traffic for data that is in-
3.3 Bandwidth-Aware Scheduling Mechanism

As the memory bandwidth has become a limitation with the increasing number of cores per chip, in this section we present in more detail the potential benefit for implementing an extension mechanism, the Bandwidth-Aware Scheduling Mechanism (xSCH), as a case study. More details on this work can be found in [12]. For practical purposes we simulated the effects of the utilization of the scheduler in an Intel SCC like architecture that is a clustered multi-core architecture [13], i.e., several groups of cores (clusters) are served by different memory controllers. We have also considered that the size of the clusters may vary, resulting in clusters with different memory bandwidth capabilities. This reflects some of the characteristics of the proposed architecture and allow us to faithfully analyze the potential impact of the xSCH proposed mechanism.

The results presented herein were obtained as a base simulations performed with a modified version of the PIN Binary Instrumentation Tool. The workloads used are presented in Table 1 along with a brief description and the inputs used in the several cases. They were carefully selected from a larger group of different applications, both from real world applications (e.g., TPC-H, NAMD) and also well known benchmarks (e.g., PARSEC, Biobench). The selection was performed based on the average memory bandwidth requirements over time of the different applications, namely we have selected them according to three classes: high, medium, and low. Moreover, we have considered different mixes of the selected workloads. In Figure 2 we show how the bandwidth increases with the number of cores in a single cluster (i.e. all cores are served by the same memory controller), where the workload X:Y:Z represents X% of the applications of high bandwidth, Y% of medium and Z% of low.

It is possible to observe that for the more demanding applications the bandwidth surpasses the 12.8Gb/s, which are supported by the memory controller (DDR3-1600). As such, in order to proceed with the execution, the applications are forced to stall, waiting for the controller to be able to serve them. In a multi-cluster architecture though, there is the potential for the scheduler (xSCH) to migrate the high demanding applications to a cluster that is not that loaded. Such a scheduling mechanism takes into consideration the fact that different clusters are served by different memory controllers. The current bandwidth required for each cluster is monitored by the TMoM. In terms of bandwidth demanding applications if an application surpasses the threshold specified in the RDB, the scheduler will redirect the most bandwidth-demanding threads to another cluster served by a controller with smaller bandwidth utilization. This redirection is performed using TMiM, to migrate the threads to a different core served by another controller. In a system with reconfigurability capabilities, the core may be reconnected to a different memory controller using the xREC mechanism.

To better understand the potential of this proposal we have also performed a full range of simulations using a simple random scheduler that doesn’t consider any metrics for its decisions. The chart presented in Figure 3 shows the preliminary results of the number of extra phases that are spent waiting for the memory controller to serve the required bandwidth when using a 50:0:50 workload mix, i.e.,
Table 1: Applications Description.

<table>
<thead>
<tr>
<th>Application/Benchmark</th>
<th>Brief Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPC-H</td>
<td>Decision support benchmark [14]. Queries from 1 to 16 using input tables obtained from DBGen [14] with scale factor 1.</td>
</tr>
<tr>
<td>DEX</td>
<td>A graph-based database query application [15]. Queries 4, 8, and a combination of all with input scale factors 3, 4 and 5.</td>
</tr>
<tr>
<td>MrBayes</td>
<td>Bioinformatics application that performs Bayesian inference of phylogeny [16]. Seventeen DNA data sets with various sizes were used as input workloads.</td>
</tr>
<tr>
<td>Biobench</td>
<td>Benchmark suite containing different bioinformatic algorithms [17]. We have considered phylip protdist, phylip protpars, fasta dna, fasta protein, and hmmer.</td>
</tr>
<tr>
<td>NAMD</td>
<td>Computer chemistry application for molecular dynamics simulation [18]. We have considered single and double precision versions with the Apolipoprotein A-I input.</td>
</tr>
<tr>
<td>PARSEC</td>
<td>Benchmark of representative applications from different areas [19]. We have considered blackscholes, streamcluster and freqmine.</td>
</tr>
</tbody>
</table>

half applications of high bandwidth and half of low. The bars in the chart are divided in three groups, each representing a homogeneous system with different cluster sizes, e.g., the 4 x 16 group represents a system with four clusters of 16 cores each. In each group, the bars represent (from left to right): “Random” - the average case obtained for the range of simulations using the random scheduler, the error bar corresponds to the worst and best case obtained; “Bandwidth-Aware” - the proposed scheduler; and “Oracle” - the minimum possible overhead obtained for the type of controllers used. It must be mentioned that in some cases the Random scheduler can hit the optimal static distribution of applications among the clusters. However, this is only achieved with a probability X:Y, where X represents the number of possible optimal static distributions and Y the overall number of different combinations of the applications among the clusters. The number of possible distributions increases rapidly with three factors: i) the increase of the number of cores per cluster; ii) the increase of the number of clusters; and iii) the reduction of the number of applications considered per migration. This complexity makes the Random scheduler a very inefficient solution. Also we show the speedup results, in terms of phases of long execution time applications relevant to the “Random” scheduler in Figure 4. From the results obtained by the proposed mechanism it is important to observe that the obtained speedup, in terms of phases compared to the “Random” scheduler, shows that in general a significant performance improvement can be achieved regarding the common scenario when the proposed scheduler is used. Namely, we achieve average speedups of 1.36x for the Low bandwidth applications, 1.48x for the Medium bandwidth applications, and 1.46x for the High bandwidth applications with the proposed scheduler mechanism. Overall, speedups of near to 2x are achieved. It is interesting to observe that the proposed bandwidth-aware scheduler is able to nearly match the results of the oracle scheduler, thus significantly reducing the time in comparison with a random scheduler. These results show that a bandwidth-aware scheduler can improve performance of future multi-cores.

Figure 4: Speedup results obtained for the long class of applications with different combinations, from left to right 00:50:50, 33:33:33, 50:00:50, 50:50:00.

3.4 Implementation of Mechanisms

Different approaches can be adopted to implement a bandwidth-aware scheduling mechanism in our proposed virtualization layer. One approach is to extend the already existing scheduler of the Xen hypervisor [5]. Performance counters [6] can be used by the scheduler to gather statistics from applications’ execution, which are then used by the scheduler to improve performance. It is important to mention that the online monitoring of the executed threads will add the minimum possible overhead to their execution. Moreover, in addition to the presented mechanism that targets bandwidth utilization among the memory controllers of the system most of the mechanisms proposed can be implemented as given modules on the virtualization layer. As mentioned in the previous section, due to the modular implementation of the proposed virtualization layer, implementing and integrating additional mechanisms can be done without completely redesigning the virtualization layer. More specifically, the Xen hypervisor allows the integration of user defined modules.
4 Conclusions

In this paper we address the issue of managing and exploiting large-scale multi-core processors. For efficient execution of applications we propose a Morphable Multi-Core Architecture along with a Virtualization Layer, *i.e.* a complete system able to wrap the complexity of the underlying hardware, through an hypervisor module. In this system the resources are managed by the hypervisor to transparently tune and achieve an improvement of the overall system efficiency.

The hypervisor proposed in this paper is composed of both Basic and Extension Mechanisms which collect information and accordingly tune the execution on the available architecture. Both by selecting the better match of application demands with the available hardware and triggering re-configuration of the underlying hardware. This functionality is ideally provided transparently to both the applications and the Operating System. As a case-study we presented the implementation of a Bandwidth-Aware Scheduler as an Extension Mechanism for the virtualization layer.

References


international conference on Parallel architectures and compilation techniques, New York, NY, USA. ACM (2008) 72–81