CoQoS: Coordinating QoS-aware shared resources in NoC-based SoCs

Bin Li a,∗, Li Zhao a, Ravi Iyer a, Li-Shiuan Peh b, Michael Leddige a, Michael Espig a, Seung Eun Lee a, Donald Newell a,1

a Intel Corporation, Hillsboro, OR, USA
b MIT, Cambridge, MA, USA

1 This work was conducted while Seung Eun Lee and Donald Newell were at Intel.

Keywords:
Network-on-chip
System-on-chip
Quality-of-service
Memory architecture
Resource management

ABSTRACT

Contention in performance-critical shared resources affects performance and quality-of-service (QoS) significantly. While this issue has been studied recently in CMP architectures, the same problem exists in SoC architectures where the challenge is even more severe due to the contention of shared resources between programmable cores and fixed-function IP blocks. In the SoC environment, efficient resource sharing and a guarantee of a certain level of QoS are highly desirable. Researchers have proposed different techniques to support QoS, but most existing works focus on only one individual resource. Coordinated management of multiple QoS-aware shared resources remains an open problem. In this paper, we propose a class-of-service based QoS architecture (CoQoS), which can jointly manage three performance-critical resources (cache, NoC, and memory) in a NoC-based SoC platform. We evaluate the interaction between the QoS-aware allocation of shared resources in a trace-driven platform simulator consisting of detailed NoC and cache/memory models. Our simulations show that the class-of-service based approach provides a low-cost flexible solution for SoCs. We show that assigning the same class-of-service to multiple resources is not as effective as tuning the class-of-service of each resource while observing the joint interactions. This demonstrates the importance of overall QoS support and the coordination of QoS-aware shared resources.

© 2010 Elsevier Inc. All rights reserved.

1. Introduction

Embedded handheld devices, such as Apple’s iPhone [19], have become increasingly popular. To speed up the time-to-market, systems-on-a-chip (SoCs) have become the design platform for these handheld devices. SoCs typically consist of several intellectual property (IP) blocks, accelerators (video decoders, graphics, imaging, etc.), along with general-purpose cores (such as Intel’s ATOMTM [4] and ARM’s Cortex A9 [9]). In such a highly heterogeneous architecture, the requirements placed by the processing on the core versus the processing on the IP blocks are radically heterogeneous in nature and impose different requirements on the platform architecture. Traditionally, in order to reduce power consumption and improve system throughput, the heterogeneous processing agents in SoC architectures share resources in the network-on-chip and the memory subsystem. However, more recently, solutions such as ARM Cortex A9 [9] have started sharing the last-level cache resources between the cores and tightly coupled accelerators. As a result, cores and IP blocks share cache space, NoC bandwidth and memory bandwidth simultaneously. In this environment, if the shared resources are not managed efficiently, this can cause severe resource contention problem and affect the performance of the applications running on the platform in unpredictable ways [16,21,5]. This performance unpredictability is not suited for future uses of SoCs. Oftentimes, many applications will require a guarantee of a certain level of performance. It becomes important to manage the allocation of these shared resources effectively to provide better overall throughput as well as a degree of quality-of-service (QoS) for each of the individual agents (core, accelerators or IP blocks).

While the effects of resource contention have been studied by several researchers in the context of CMP platforms [8,20,21,32,33,36,37,42,47], when comparing CMP architectures to SoC architectures, there are two major differences. The first lies in the type of processing agents (only cores in CMP versus cores together with hardware IP blocks in SoC). While the cores tend to be both latency and bandwidth sensitive, they do not have stringent requirements on memory accesses. On the other hand, hardware IP blocks impose very different requirements and depend highly on the processing involved. For example, display controllers have specific bandwidth requirements, which if not met, cause glitches in the display as observed by the users. The second key difference is that the amount of resources of CMP is much higher than that of SoC. This in turn causes resource arbitration to be of more
significance in SoC architecture. In this context, it is important to ensure that the interference between the accelerators and the cores is carefully handled to ensure appropriate QoS.

Researchers have proposed different QoS frameworks to provide certain levels of performance for different applications. One key objective of QoS is to provide applications a soft upper bound on the execution time by allocating a certain amount of resources to each application. However, most existing works focus on only one individual resource for QoS support and management. For example, there are works that focus on cache QoS management, but ignores contention in the shared communication fabric [34,42,23,47,21,16,37]. These techniques work well when only one or two resources are shared and when the number of processing agents that share those resources is small. However, future SoC architectures are expected to have tens of heterogeneous agents that share cache space, NoC bandwidth and memory bandwidth simultaneously. An application’s guaranteed service level is thus determined by the weakest guarantee for any of its shared resources [24]. A guarantee of one individual resource is not sufficient to support the overall performance in future SoCs. It becomes important to support QoS at system level by coordinating the management of cache, NoC and memory in SoCs.

In this paper, we propose a CoQoS architecture framework that enables coordinated management of three critical shared resources (cache, NoC and memory). Our proposed architecture can expose the CoQoS to OS, runtime layers and device drivers so that they can control or guide the class-of-service for the cores as well as the IP blocks. The proposed architecture has low cost, and is thus suitable for SoC architectures. We evaluate the CoQoS techniques individually as well as in combination for SoC configurations that consist of general-purpose cores and hardware IP blocks. The results show that careful control of the class-of-service assignment to each individual resource is required. To the best of our knowledge, this is the first work to consider all the three critical shared resources (cache, NoC and memory) in the NoC-based SoC architectures.

The remainder of this paper is organized as follows. Section 2 provides some background on NoC-based SoC architectures and motivates the need for joint resource allocation in these platforms. Section 3 presents the details behind our proposed class-of-service based CoQoS architecture. Section 4 presents the evaluation methodology and simulation results. Section 5 describes prior related work in this area. Section 6 summarizes the paper by presenting our conclusions and future work in this area.

2. Resource contention in NoC-based SoCs

In this section, we introduce NoC-based SoC architectures and its components. We then study the effects of individual QoS support on future SoC systems and compare the results with the coordinated QoS support.

2.1. System-on-chip overview

SoC solutions are being employed for a variety of different industry products ranging from sensors, handhelds, set-top-boxes to routers. Fig. 1 presents an overview of a future SoC architecture in small form factor (handheld and set-top-box style) architectures. In future SoC architectures, it is expected that there will be a few cores (Cortex A9 [9] for example) and several different IP blocks that are dedicated to accelerating application-specific processing (graphics, media, audio, security, imaging, etc.). The cores and the accelerators are interconnected through a network-on-chip fabric for inter-agent communication as well as communication to cache and memory resources. As a result, the cores and IP blocks contend for cache space, NoC bandwidth and memory bandwidth resources.
memory QoS support, the IPC improvement for the two SPEC applications is 13.8%. Finally, when we apply QoS support for all the three shared resources together, a 32.6% improvement on IPC is achieved compared to the unmanaged scheme. From this example, we can see that no individual QoS scheme can achieve the performance improvement that comes from applying QoS to all the three shared resources simultaneously. Hence, there is a crucial need for QoS support by jointly managing all the three critical on-chip shared resources together.

In this paper, we propose a unified class-of-service resource management architecture that can control three different shared resources (cache, NoC and memory) simultaneously and in a coordinated fashion, and provide QoS support for different applications. Our focus is not on improving the individual QoS technique for cache, NoC and memory. Rather, we employ existing QoS techniques and propose a unified QoS architecture for SoCs where the QoS can be exposed to the OS, runtime layer and device drivers so that they can control or guide the class-of-service appropriately across the entire SoC platform. We show that using simple localized solutions for QoS-aware arbitration along with a class-of-service model to tie them together is quite powerful and provides a flexible solution for future SoC architectures.

In the next section, we describe our proposed class-of-service based resource management architecture in detail.

3. CoQoS: a class-of-service architecture

There are several approaches that can be considered for implementing QoS in SoC architectures. The approaches range from soft real-time guarantees to hard real-time guarantees. Soft real-time guarantees tolerate occasional deadline misses while hard real-time guarantees have to meet all deadlines. In this paper, we focus on soft real-time guarantees as a first step towards a unified CoQoS approach, and leave hard real-time guarantees as future work. We propose a class-of-service based approach [21] for CoQoS support primarily because it reduces the support needed in each resource as well as the end-to-end support. Given that SoC architectures are typically severely constrained for area and power, we expect that such an approach is most prudent. In addition, we also have taken care to provide sufficient hooks within the CoQoS architecture such that hard guarantees could be introduced subsequently in the future if necessary.

Fig. 3 gives an overview of our proposed class-of-service based QoS architecture. As shown in Fig. 3(a), the proposed CoQoS architecture consists of three key components. The first is class-of-service (CoS) assignment. This step assigns each application a class-of-service ID (CoS-ID) for the shared resources. This CoS-ID is a thread identifier which indicates the service level the application will receive during its execution. The second step is class-of-service mapping. In this step, the appropriate limits or weights for each CoS-ID are specified and mapped onto cache, NoC and memory. The third step is class-of-service based resource management. This step manages shared resources for each application based on their CoS-ID. Next we explain each step in detail.

3.1. Class-of-service assignment

The first step is for the platform to allow a CoS-ID to be assigned for each application. This CoS-ID indicates the service level the application can receive for the shared resources. Fig. 3(b) illustrates the class-of-service assignment approach. This exposure differs somewhat between general-purpose cores and IP blocks. For general-purpose cores, the simplest approach is to expose a new architecture register (CoQoS-Register or CQR). The OS or runtime layer will essentially write the CoS-ID into this register whenever it schedules an application or software thread to run on the core. For IP blocks, a similar register needs to be exposed, but the mechanism by which it is exposed and the extent to which it is exposed is as follows:

- **CQR-within-IP.** Since the IP block is a black-box provided by a third party provider, it is possible to require the IP provider to design a CQR register on the IP block and expose it on its interface. In addition to the ability to write the CoS-ID into the CQR register, the IP provider will also need to expose the CoS-ID every time a transaction is issued from the IP block so that the CoS-ID can be associated with the transaction. The advantage of this approach is that the IP block provider can determine (at a fine grain level) which transactions should be assigned which CoS-ID based upon the type of transaction or the criticality of it.

- **CQR-around-IP.** An alternative approach to implement the CQR for the IP blocks is to design it into the bridge that connects the IP block to the NoC and the rest of the platform. By implementing the CQR in such a manner, the burden falls on the SoC integrator to expose the register by associating it with the IP block configuration and allowing OS, driver or runtime layer to access it.

3.2. Class-of-service mapping

The second step in this process is to appropriately encode the class-of-service specification so that it can be mapped to shared resource management flexibly. There are two approaches that can encode the class-of-service specification onto CQR. The first approach is to encode a single CoS-ID in the CQR for all of the shared resources in question. However, the mapping of a single class-of-service ID to every resource does not make sense since different agents (cores, IP blocks, etc.) will require different levels of QoS for different resources. As an example, some IP blocks are streaming in nature (implying a low class-of-service in cache) and more bandwidth sensitive (implying a high class-of-service in NoC and memory). As a result, the most flexible approach to encoding and mapping classes-of-service to resources is to allow a separate class-of-service ID for each resource in the CQR register. Fig. 3(c) illustrates the encoding of three classes-of-service IDs (C-ID for cache, N-ID for NoC and M-ID for memory). In our experiments, we use separate class-of-service ID for each shared resource. Note that in our scheme, multiple applications can be mapped to the

![Fig. 3. Class-of-service (CoS) based CoQoS architecture and key components.](image-url)
same CoS-ID. That is, a class-of-service level may consist of one application or a group of applications. All the applications in a given service class share the resources allocated for that class.

Once the class-of-service encoding is available, the next step is to allow the class-of-service to be mapped onto cache, NoC and memory resource management. This requires a configuration process to essentially map each class-of-service ID to the appropriate limits or weights that are to be used within the resource for resource management. The limits/weights for each class-of-service ID is maintained in a table of registers that can be either statically specified (on boot-time) or dynamically determined (by the OS). In this paper, we experiment primarily with a static configuration approach, leaving the dynamic modification (through either application profiling or resource usage profiling) of configuration as future work. There have been numerous algorithms proposed for resource allocation [23, 43, 12, 38, 25, 39, 49]. We assume that the specific limits/weights are provided by OS or runtime layer to ensure a certain level of performance. The goal of our work is to provide an architectural and a micro-architectural mechanism to enforce the resource allocation.

3.3. Class-of-service-based shared resource management

As mentioned above, the classes-of-service are encoded in the CQR and mapped to a table of limits/weights that are required for use in shared resource management. For cache, NoC and memory resource management, we require that every transaction or request arriving at the resource is tagged with the CoS-ID. This can be done by encoding the CoS-ID in the header flit of each packet transmitted through NoC. Once the CoS-ID is available, the resource locally looks up the limits/weights associated with the CoS-ID and performs the QoS-aware resource management appropriately. For instance, the proposed QoS-aware resource management for cache is limit-based (where a maximum space is specified for each class-of-service). The proposed QoS-aware resource management for NoC is weight-based proportional arbitration locally at each router input as well as globally across router inputs at each router output. The proposed QoS-aware resource management for memory bandwidth arbitration is again weight-based proportional scheduling at the memory controller. We choose limit-based QoS for cache and weight-based QoS for NoC and memory for our CoQoS architecture because SoC architectures are typically highly constrained in area and power, and we believe that these simple techniques are practical to implement at low area and power overheads. The details of each of these resource management techniques are described in the following sections.

3.3.1. Limit-based CoQoS in shared cache

Our QoS-aware shared cache management achieves class-of-service based resource allocation by keeping track of the space used in cache by each class-of-service and ensuring that this does not cross the maximum limits specified in the CoQoS mapping table (as shown in Fig. 4). In order to do so, two key mechanisms are added:

(i) Class-of-Service Tagging. In order to track the space usage for each class-of-service, every line in the cache is tagged by the class-of-service ID that is currently using that line. When a new line is allocated, the counter associated with the new line’s class-of-service is incremented in the CoQoS mapping table. If a victim line was evicted from the cache for this allocation, then the counter associated with the victim line’s class-of-service is decremented from the CoQoS mapping table.

(ii) Class-of-Service Aware Replacement. In order to ensure that the limits specified by the CoQoS mapping table are not violated, the replacement policy for the cache is modified to be CoS aware as follows: (a) if a class-of-service has reached its limits, but the cache is not full (e.g., other service classes have not fully utilized their allocated cache space yet), the request from this class-of-service can choose an invalid line in the cache and exceed its limits. This ensures that if the cache space allocated to one class-of-service is not fully utilized, other applications can use this space to avoid resource waste; (b) if the space usage of a class-of-service has not reached its limits, but all the cache space is fully utilized now (e.g., the cache space is used by other applications when the cache is not full), it is ensured that a line is chosen from another class-of-service that is farthest away from the limit specified (the class-of-service that has the highest positive distance between its utilization and its limit); (c) if the space usage of a class-of-service has reached its limit and the cache space is fully utilized, it is ensured that the victim line is always chosen from the same class-of-service. Fig. 4 also indicates these replacement criteria. In order to implement such a replacement policy, the simplest approach is to keep track of the “farthest-distance class-of-service” (one that has the highest positive distance between its utilization and its limit) by calculating this every time the CoQoS table is updated. In addition, an “about-to-exceed” bit is also maintained in the CoQoS table every time the utilization is close to exceeding the limit (by a single line).

3.3.2. Weight-based CoQoS in NoC routers

A typical NoC router consists of several inputs (each with one or more buffers) contending for switch allocation to one of the several outputs. In this paper, we employ a 5 × 5 router consisting of 1 local port and 4 interconnect ports for a typical mesh-based NoC. CoQoS in the NoC router ensures that a higher class-of-service gets more switch bandwidth (through successful switch allocations) than a lower class-of-service. In order to enforce this, we employ a two-level weight-based arbitration mechanism (illustrated in Fig. 5).

Researchers have shown the effectiveness of different QoS support
techniques in NoC [46,6,7,13,14,18,24,28,37]. We choose weight-based arbitration for NoC QoS because it is simple to implement and provides guarantees for bandwidth allocation and aligns with our system-level QoS support mechanism, thus suitable for our CoQoS architecture. Instead of proposing new NoC QoS techniques, we employ a representative cost-effective NoC QoS technique. Other techniques can be used in a similar way.

For weight-based NoC QoS, the weights for both levels of the arbitration are specified in the CoQoS mapping table for NoC (identical to weight-based table shown in Fig. 3). At each input port of the router, each class-of-service is assigned its own buffer queue. The first-level of the weight-based arbitration is local to each input port. Within each input port, when arbitrating between the buffer queues to find a candidate for global switch allocation, we ensure that the highest class-of-service queue is serviced first in any given cycle until it overflows its weight threshold within a pre-specified duration of time (referred to as ResetAt). The same procedure is repeated in the second-level arbitration when selecting between the candidates across input ports that contend for the same output port. This global arbitration step is performed within switch allocator as shown in Fig. 5. It should be noted that the approach described above ensures that if the NoC bandwidth is underutilized by a high priority class, then a lower priority class can use those resources. Besides, we choose a weight-based arbitration instead of fixed priority arbitration to avoid starvation for lower priority applications.

3.3.3. Weight-based memory CoQoS

For CoQoS memory bandwidth allocation, we employ a similar weight-based approach as described for the NoC above. As shown in Fig. 6, the incoming requests are directed towards different input queues in memory controllers based on the class-of-service. When arbitrating amongst input queues, the request queue with the highest class-of-service that has a pending request and has not extinguished its weights (based on the current usage in the table in Fig. 6) during a specified duration (based on the ResetAt counter) is chosen. When the current time counter (CurrentCnt) reaches the weights specified in the ResetAt counter, this counter and all of the current entries in the table are automatically reset back to zero. By doing so, it is ensured that during any specified duration, the bandwidth provided to the classes-of-service are proportional to the weights indicated for the classes-of-service in CoQoS memory mapping table.

3.4. Hardware overhead analysis

In this section, we analyze the overhead for the modifications required to support CoQoS.

**Cache QoS overhead.** We use limit-based cache control for cache QoS support, which requires a CoQoS mapping table which is a single table for the entire cache. We consider 3 classes-of-service at each shared resource in our experiment. This can be readily expanded to more class-of-service levels. In the mapping table, maximum and utilization are represented in terms of number of cache lines. In our experiment, the last-level cache is 2MB with 64B line size, thus the total number of cache lines is 215. Hence, the number of bits that are needed to encode maximum, and utilization is 15 each. We also need 1 bit to indicate that usage is about to exceed a threshold. As a result, 4 bytes are required for each class-of-service. For a 3 class-of-service system, this mapping table consumes 12 bytes in total. Another overhead for cache QoS is the 2 bits added for each tag to keep track of the class-of-service. We use Cacti 5.3 [45] to simulate the two cache configurations (one with 2 additional tag bits and the one without) in our experiments at 65 nm technology (detailed cache setting can be found in Table 1). The simulation results show that for the one with cache QoS support, its area overhead is only 0.23% and the power overhead is only 0.78% compared to the scenario without cache QoS support. This is consistent with prior estimates of the overhead of limit-based cache QoS. Furthermore, since victim selection is overlapped with DRAM data fetch, it is not on the critical path and does not affect cache miss latency.

**NoC QoS overhead.** The overhead for NoC-QoS is at the switch allocator in every router. With weighted NoC-QoS, each arbiter in the switch allocator now needs a counter to record the number of cycles that has been assigned to each class within a reset window. It also needs a comparator to check if each class has reached its allocated bandwidth within a reset window. However, the virtual channel allocator circuit will become simpler. For example, in our experimental setup, when there is no NoC-QoS, each input port has three virtual channels. When weighted NoC-QoS is supported, each class-of-service is allocated one virtual channel. As a result, the number of requests for each arbiter in the virtual channel allocator is smaller, resulting in lower power consumption and smaller area. To quantify the overall overhead for weighted NoC QoS, we first use Synopsys Design Compiler [44] to synthesize a switch allocator with weighted round-robin arbiters in the 90 nm technology. We then scale the power and area down to 65 nm using linear scaling. These new power and area numbers for a switch allocator with weighted round-robin arbiters are then added into ORION 2.0 [22]. We use this modified ORION 2.0 to simulate the two routers in our experiments in the 65 nm technology: one with weighted NoC QoS, and the other without (Table 1 lists detailed router settings). The modeling results show that the area overhead for NoC QoS support is 1.8% while each router now only consumes 97% of the total power compared to a router that does not support
QoS. In addition, the synthesized result shows that even thought the latency for switch allocator increases, it still fits within one clock cycle. As we assign each service class a dedicated queue, when the number of class-of-service increases, the area overhead increases accordingly. However, we believe that the number of classes-of-service will not increase proportional to the number of agents in the SoC. Fewer classes-of-service are sufficient to provide the differentiation between simultaneously running applications. As a result, the overhead for NoC QoS is negligible.

Memory QoS overhead. For weighted memory QoS, a weighted arbiter is needed at the memory controller to determine the specific class-of-service queue that will be granted access to a memory bank. We use Synopsys Design Compiler to synthesize a weighted round-robin arbiter in the 90 nm technology and scale the area down to 65 nm using linear scaling. We also scaled memory controller’s area in the area down to 65 nm using linear scaling. The configuration employs three types of agents in [3] from 130 nm to 65 nm. We then compared the area overhead for adding weighted arbiter in memory controller and found that the area overhead is 3%.

In short, the overhead required to support CoQoS is negligible, and the cost for implementation is low. Besides, the hardware overhead increases when the number of classes-of-service increases, not necessarily correlating with the size of the system (except for the cache QoS which increases linearly with the size of the cache). Hence, CoQoS is flexible, scalable, and adds little power and area overhead, making it suitable for NoC-based SoCs.

4. Evaluations

In this section, we explore the impact of different static QoS assignment policies on different classes using the class-of-service based QoS framework we proposed.

4.1. Simulation methodology

In order to evaluate CoQoS, we developed a detailed SoC platform based on ASPEN [29] and GARNET [1]. ASPEN is a trace-driven platform simulator that models a detailed cache hierarchy, coherence protocol and memory subsystem. However, ASPEN lacked a detailed interconnect model. To address this issue, we integrated GARNET into ASPEN. GARNET is a detailed NoC model that was originally integrated into the GEMS full-system simulation infrastructure [27]. It allows different interconnect topologies, and different router parameters such as buffer sizes, virtual channels, etc. We integrated ASPEN and GARNET to simulate a detailed NoC-based SoC platform.

Our experiments are conducted to simulate the SoC configuration illustrated in Fig. 7. The configuration employs three types of agents: (a) two cores, (b) two graphics processing units, and (c) two IP blocks. These agents are interconnected to a shared last-level cache that is physically distributed into two blocks (i.e. NUCA) and a memory controller that provides the interface to low power DRAM. All of these different agents (totally 9) are interconnected in a 3 × 3 mesh NoC. Although only one configuration is shown, the placement options could differ in terms of proximity to cache, proximity to memory and the proximity between agents. However, we will use this default topology in Fig. 7 for experimention in this paper. The reason we choose this topology is that high priority agents (cores) are placed close to cache and memory controller. In terms of workloads, we ran the following workloads simultaneously to simulate a SoC-like environment. The two cores ran SPEC CPU2000 applications (art and mcf chosen for their compute/memory intensive nature). The graphics processing units ran 3DMark2005 (to represent intense graphics processing required). The IP blocks ran a mobile augmented reality (MAR) application [26]. MAR is an emerging usage model on handheld devices that automatically recognizes user input object and displays information regarding the object. It represents a streaming processing engine that does not have high bandwidth requirement.

In our study, we assume three service classes: Class 0 is the high-priority application class (referred to as HPA), Class 1 is the medium-level priority application class (referred to as MPA), and Class 2 is the low-priority application class (referred to as LPA). Clearly, more classes can be readily supported. We group the six workloads into three service classes, as shown in Table 1: Class 0 consists of two SPEC applications (art and mcf), Class 1 consists of two graphics applications (Gfx1, Gfx2), and Class 2 consists of two MAR accelerators (IP1 and IP2). We assign cores the highest priority and other agents lower priorities in decreasing order of their bandwidth requirements as is a typical case in the SoC environment [46]. However, the priority assignment also depends on the specific market segment that the SoC devices targeted at as well as the applications running simultaneously on the platform.

Table 1 shows the rest of the simulation parameters (for core, NoC, cache, memory, etc.). Note that for NoC, when NoC QoS is not supported as in baseline, each input port has 3 virtual channels with 5 buffers per virtual channel. When NoC QoS is enabled, there are still 3 virtual channels at each input port, but each class-of-service is assigned 1 virtual channel and can only use its allocated virtual channel buffers. This is to ensure that the total number of buffers and virtual channels are the same as in the baseline for a fair comparison. We assume three service priority levels for each shared resource, with 0 being the highest priority level, 1 the medium priority level, and 2 the lowest priority level for C-ID, N-ID and M-ID, respectively.

In the next few sections, we perform design explorations to study the impact of different static QoS policies on the performance of each application class in NoC-based SoC. We first study the impact of each individual QoS on the three service classes and then show the impact with coordinated QoS support. We use the overall performance denoted by IPC (Instructions Per Cycle) as our evaluation metric.

<table>
<thead>
<tr>
<th>Subsystem</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core</td>
<td>1 GHz low power core (2-issue)</td>
</tr>
<tr>
<td>Cache</td>
<td>2 × 1 MB shared L2 cache (5 core clocks)</td>
</tr>
<tr>
<td></td>
<td>32 KB private L1 cache (1 core clock)</td>
</tr>
<tr>
<td>Graphics units</td>
<td>Based on Intel’s graphics media accelerator</td>
</tr>
<tr>
<td>IP blocks</td>
<td>One MAR match accelerator and one MAR hessian accelerator</td>
</tr>
<tr>
<td>NoC</td>
<td>1 GHz, 3 × 3 2D mesh, dimension order routing, 64 bits flit, 4-cycle router pipeline, 9 flits data packet, 2 flits request packet, 3 VCs, 5 buffers per VC (1 VC per class-of-service for NoC QoS)</td>
</tr>
<tr>
<td>Memory Bandwidth</td>
<td>3.2 GB/s maximum sustainable memory bandwidth for SoC configuration</td>
</tr>
<tr>
<td>Memory Latency</td>
<td>100 core clocks</td>
</tr>
</tbody>
</table>
4.2. Cache QoS impact

In this section, we study the performance impact when we only apply cache QoS on the three service classes in our SoC environment (NoC and memory are unmanaged). In the base case where no QoS is enforced, all applications issue as many requests as they can. Therefore, MPA takes 64% of the L2 cache, while HPA only takes 22% of the L2 cache. When we apply cache QoS policy, HPA (art and mcf) has the highest QoS and is assigned to C-ID0. We vary the cache space limit for HPA from 40%–100% of the total cache space (if HPA is allocated 100% cache space, MPA and LPA will bypass L2 cache). The parameters for NoC and memory are the same as in baseline shown in Table 2. Note that in the following experiments, for the resources that are not specified, the parameters are the same as in baseline. Fig. 8(a) shows the impact of cache QoS on HPA in terms of cache miss rate. As we increase the cache space limit for HPA, cache miss rate for HPA continues to decrease as shown in Fig. 8(a). Fig. 8(b) shows the impact of cache QoS on the three service classes in terms of IPC, which is normalized to the base case where no QoS is used. We find that the IPC for HPA increases first and then starts to decrease while the IPC for MPA and LPA continue to decrease. For example, when HPA can take 70% of the cache space, the IPC for HPA is increased by 17.8%. However, when HPA can take more than 70% of the cache space, the IPC for HPA starts to decrease. This is because even though the cache miss rate for HPA continues to decrease when we increase its cache space, the cache miss rate for MPA and LPA increase rapidly, causing serious contention in NoC and memory. As a result, the performance for all applications are degraded, leading to reduced overall system performance. From this study, we find that improvement in cache miss rate for HPA does not always lead to its system-level performance improvement. We need to carefully consider the system-level impact when allocating cache space.

4.3. NoC QoS impact

In this section, we study the impact of NoC QoS on the performance of the three workloads. We perform two studies here: the first is to study the effects of weight assignments on performance, and the second is to study the effects of priority assignments on performance.

We first study NoC weight assignment effect. We set HPA as high NoC priority (N-ID0) and vary the weights assigned for each priority class as shown in Table 3. In the first experiment, we assign weights to each application class based on their average bandwidth requirement (marked as noc_qos_exp1). In the second experiment, we assign more weights to HPA to account for bursty requirement rather than average bandwidth requirement (marked as noc_qos_exp2). In both experiments, HPA is prioritized over MPA and LPA until it exceeds its allocated weights. Note that for NoC, if the allocated weights are not fully utilized within a reset window, they can be used by lower priority applications. As a result, there is no bandwidth waste. Fig. 9 shows the impact of these two weight assignment policies on the performance of the three service classes. From the figure, we can see that if the weight assignment is based on average bandwidth requirement, the performance improvement for HPA is 3.1% (noc_qos_exp1). When we increase the weight assigned for HPA, its IPC is increased to 3.9% as in noc_qos_exp2. This shows that latency is more important for HPA. On the other hand, the IPC for MPA and LPA are decreased. Note that the IPC improvement for HPA is not dramatic compared to the improvement from cache QoS. There are two reasons. The first one is that at the injection port when there is no NoC QoS enabled, the network packet can use all the virtual channels available in the input port. However, when NoC QoS is supported, the packet can only inject into its own allocated virtual channels within its class, causing packets to be blocked at the network interface. So for high priority class, even though the latency in the network is reduced, the queuing latency at the network interface is increased. The overall improvement is the combination of both queuing latency and network latency. This is why the overall IPC improvement is not larger. This reflects the overhead that comes with most QoS enforced systems. This issue can be mitigated by dynamically allocating the number of virtual channels each priority class can use so that the virtual channels can be utilized more efficiently.
Table 4 Experimental setup for N-ID mapping in NoC QoS.

<table>
<thead>
<tr>
<th>Class</th>
<th>noc qos_exp3</th>
<th>noc qos_exp4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class 0 (HPA)</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Class 1 (MPA)</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Class 2 (LPA)</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

The second reason is that we use a $3 \times 3$ NoC network and the cores are placed close to the last-level cache as well as the memory controller. As a result, the time spent in NoC is small for HPA. We expect that the IPC improvement for HPA will be more significant for a larger NoC system.

Next, we study how different NoC priority assignments affect the performance of the three service classes. We ran two experiments as shown in Table 4. In the first experiment (marked as noc qos_exp3), we set HPA as high NoC priority (N-ID0). In the second experiment (marked as noc qos_exp4), we set MPA as high NoC priority (N-ID0). Fig. 10 shows the performance impact of these two NoC QoS mapping schemes. As can be seen from Fig. 10, when HPA has high NoC priority as in noc qos_exp3, its IPC is increased by 3.9%, while the IPC for MPA is decreased by 8.6%. When HPA has medium NoC priority as in noc qos_exp4, its IPC is almost the same as that in baseline (increased by only 0.3%), while the IPC for MPA is decreased by 3.7% now. From this experiment, we can see that both HPA and MPA are sensitive to NoC latency. Higher priority in NoC leads to better performance.

Table 5 Experimental setup for weight assignment in memory QoS.

<table>
<thead>
<tr>
<th>Class</th>
<th>mem qos_exp1</th>
<th>mem qos_exp2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class 0 (HPA)</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>Class 1 (MPA)</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>Class 2 (LPA)</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Fig. 9 Impact of NoC weights.

Fig. 10 Impact of NoC priority.

4.4. Memory QoS impact

In this section, we study the impact of different memory QoS assignment policies on the performance of the three service classes. Again, we perform two studies here: one is weight assignment and the other is priority assignment.

First, we study how different memory weight assignments affect the performance for the three service classes. We set HPA as high memory priority (M-ID0) and vary the weights assigned as shown in Table 5. In the first experiment, we assign weights based on the average bandwidth requirement from each application class (marked as mem qos_exp1). In the second experiment, we assign more weights to HPA to account for bursty requirement rather than average bandwidth requirement (marked as mem qos_exp2). In both assignments, HPA is prioritized over MPA and LPA until it exceeds its allocated weight. Fig. 11 shows the simulation results. From the figure, we can see that the performance for HPA increases from 11.5% (mem qos_exp1) to 13.8% (mem qos_exp2) when we increase the weights assigned for HPA. This is because HPA’s memory traffic is bursty. When we increase the weights assigned for HPA, its memory latency is reduced, leading to better performance improvement. This comes at the cost of reduced performance for MPA as expected. We need to consider this factor when allocating weights for HPA.

Next, we study how different memory priority assignments affect the performance of the three service classes. We ran two experiments as listed in Table 6. In the first experiment (marked as mem qos_exp3), HPA was set as high memory priority (M-ID0). In the second experiment (marked as mem qos_exp4), MPA was set as high memory priority (M-ID0). Fig. 12 shows the impact of these two memory QoS mapping schemes. From Fig. 12, we can see that in the first experiment, HPA gets 13.8% improvement on IPC while MPA’s IPC is decreased by 6.3%. In the second experiment, HPA gets 7.0% improvement on IPC while MPA’s IPC is decreased by 3.2% now. This shows that HPA is more sensitive to memory latency than MPA in this experiment.

4.5. Coordinated cache, NoC and memory QoS

After studying the impact of each individual QoS on the performance improvement for HPA, we now study the combined effects...
Table 7: Experimental setup for all QoS limit/weight assignments.

<table>
<thead>
<tr>
<th>Class</th>
<th>C-ID</th>
<th>Cache limit (%)</th>
<th>N-ID</th>
<th>Weight (local/global)</th>
<th>M-ID</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class 0 (HPA)</td>
<td>0</td>
<td>70</td>
<td>0</td>
<td>10/40</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>Class 1 (MPA)</td>
<td>1</td>
<td>20</td>
<td>1</td>
<td>6/24</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Class 2 (LPA)</td>
<td>2</td>
<td>10</td>
<td>2</td>
<td>2/8</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Fig. 11. Impact of memory weights.

Fig. 12. Impact of memory priority.

The limits/weights are chosen based on our performance study for each individual resource QoS above. For SPEC applications, the performance depends on the cache miss rate as well as the cache miss penalty which is affected by NoC latency and memory latency. As a result, we allocate 70% cache space to SPEC applications to reduce the cache miss rate and allocate more NoC and memory weights to accommodate latency requirement in bursty traffic. Fig. 13 shows the simulation results. From the figure, we can see that when HPA is mapped to high priorities in cache, NoC and memory simultaneously, its IPC is improved by 32.6% compared to the base case, which is a tremendous improvement. Furthermore, we compare this joint QoS result with each individual QoS scheme where HPA is mapped on C-ID0 for cache QoS only, N-ID0 for NoC QoS only, and M-ID0 for memory QoS only, respectively. The comparison results are shown in Fig. 14. We can see in Fig. 14 that for HPA, no individual QoS scheme can achieve the performance improvement that comes from joint QoS in all the three critical shared resources simultaneously. This demonstrates the crucial need for jointly managing all three critical on-chip shared resources together.

We also measure overall system performance using weighted speedup metric [40], which is a commonly used multi-program performance metric. Weighted speedup is calculated by summing up the relative IPC performance of each application when it runs alone versus when it runs together with other applications. Weighted speedup = \( \sum \frac{IPC_{shared}}{IPC_{alone}} \). Fig. 15 shows the effects of individual QoS as well as joint QoS on overall system performance. From Fig. 15, we can see that joint QoS improves system performance by 2.7% over baseline unmanaged scheme, which no individual QoS can achieve. This is because HPA can make faster progress with joint QoS while MPA and LPA are not slowed down significantly. With NoC QoS, we note that the system performance is decreased compared to the baseline. Again, this is because for NoC QoS, the packet can only inject into its own allocated virtual channels, causing packets to be blocked at the network interface for each service class, resulting in less system performance.

Next, we show the breakdown of the total latency for each application class in terms of latency spent in NoC, latency spent in memory, as well as the queuing latency spent in network interface and cache queues for our joint QoS results compared to unmanaged baseline in Fig. 16. The latency here is the round-trip latency for read request when it misses in both L1 and L2 caches and goes to memory to fetch the data. The measurement starts from the time the request misses in L1 cache and is inserted into the network interface queue, until the whole data is returned to the L1 cache. The results are averaged for all read requests that miss in L1 and L2 caches. From Fig. 16, we can see that for HPA, when we apply joint QoS, NoC latency, memory latency are both reduced, while queuing latency is increased. This is because with NoC QoS, there is less virtual channels available for each service class, causing increased queuing latency at the network interface. However, the reduction in NoC latency and memory latency is more significant than the increase in queuing latency, leading to better performance improvement for HPA, with the average latency reduced by 20.7%.
Table 8
Experimental setup for all QoS mappings.

<table>
<thead>
<tr>
<th>all_qos_exp1</th>
<th>all_qos_exp2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>C-ID</td>
</tr>
<tr>
<td>Class 0 (HPA)</td>
<td>0</td>
</tr>
<tr>
<td>Class 1 (MPA)</td>
<td>1</td>
</tr>
<tr>
<td>Class 2 (LPA)</td>
<td>2</td>
</tr>
</tbody>
</table>

Next, we measure the read request latency variation, or jitter, for each service class. Fig. 17 shows the maximum latency before and after applying QoS support. In Fig. 17, the results are normalized to the average latency in baseline unmanaged scheme. From Fig. 17, we can see that when we apply QoS, the maximum latency perceived by HPA is decreased by 40.8%, which is a significant improvement over unmanaged scheme. This comes at the cost of increased average and maximum latencies for MPA and LPA as expected.

We next study how different priority assignments affect the performance of each service class. The reason we study this is because assigning the same class-of-service to multiple resources may not be as effective as tuning the class-of-service to each resource while observing the joint interactions. Managing of shared resources in coordination can more effectively utilize the valuable on-chip resources and improve system-level throughput. In this study, we ran two sets of experiments, as listed in Table 8. In the first experiment (marked as all_qos_exp1), HPA has the highest priority in the cache, NoC and memory (C−ID0+N−ID0+M−ID0).

In the second experiment (marked as all_qos_exp2), HPA has high cache priority, but mid-level NoC and memory priorities (C−ID0+N−ID1+M−ID1). From Fig. 18, we can see that in the first experiment, when HPA has the highest priority in the cache, NoC and memory, its IPC is increased by about 32.6% compared to the base case unmanaged scheme, while MPA's IPC is decreased by about 13.7% compared to the unmanaged scheme. In the second experiment, we map MPA to high NoC and memory priorities. We find that MPA's IPC is decreased by 9.2% now, which improves by 4.5% compared to the first experiment. Although HPA is mapped to mid-level NoC priority, its IPC is still increased by 26.7%. Furthermore, the total system throughput in all qos_exp4 is improved by 1.3% compared to all_qos_exp3. This shows the trade-off for resource allocation and demonstrates the importance of coordinately managing the shared resources so that the high-
priority application gains performance improvement while the overall system throughput is improved. From this example, we can see that if the maximum performance is needed for high priority application, it should always be mapped onto high priority service level for cache, NoC and memory. If the high priority application can tolerate some degree of performance loss and the overall system throughput desirable, then different priority applications can be mapped onto different levels of QoS for different shared resources. Our proposed class-of-service based architecture supports this flexible assignment for each shared resource so that different goals can be realized.

4.6. Resource sensitivity discussion

In this section, we study the performance of HPA when applying only two out of the three QoS schemes. We ran three experiments here. In each experiment, only two QoS schemes are enabled while the third resource is left unmanaged. In the first experiment, we enable cache and NoC QoS, and set HPA as high cache and NoC priorities (marked as cache_noc_qos). In the second experiment, we enable cache and memory QoS, and set HPA as high cache and memory priorities (marked as cache_mem_qos). In the third experiment, we enable NoC and memory QoS, and set HPA as high NoC and memory priorities (marked as noc_mem_qos). Fig. 19 shows the performance impact for HPA compared to when we apply all three QoS schemes simultaneously. All the results are normalized to the baseline unmanaged scheme. From Fig. 19, we can see that in cache_noc_qos, HPA’s IPC is increased by 23%. In cache_mem_qos, HPA’s IPC is increased by 29.6%. In noc_mem_qos, HPA’s IPC is increased by 15.9%. However, none of them achieves the performance improvement when we apply all the three QoS schemes simultaneously. This is because guarantees in two shared resources can still cause contention in the third shared resource. For example, in experiment cache_noc_qos, even though giving higher priority to some traffic in NoC also prioritizes their access to the memory, this may not be enough as higher priority traffic can still be blocked by lower priority traffic in memory. Allocating each service class a dedicated queue in memory QoS can ensure that higher priority traffic is not blocked by lower priority traffic until it exceeds its allocated weights. Overall, this group of experiments shows that for applications that contend for cache, NoC and memory resources simultaneously, all the cache, NoC and memory QoS schemes are important to achieve the overall performance improvement for HPA.

In addition to providing QoS to ensure performance guarantees for higher priority applications, designers can also increase the amount of the shared resources so that the contention is reduced. For example, designers can increase the cache size so that the working sets for all the simultaneously running applications can be fitted into the cache. If this is the case, then cache QoS is not necessary as there will be no contention for cache space. Designers can also keep increasing the NoC and/or memory bandwidth to reduce the contention in NoC/memory. This will reduce the importance of NoC/memory QoS. However, NoC/memory QoS may still be needed if there are times that requests from different agents contend for NoC/memory bandwidth at the same time. In this case, NoC/memory QoS helps to ensure that higher priority traffic be arbitrated first. As increasing the size of the shared resources helps to relieve the contention, it will increase power and area in the platform, leading to higher cost that is not desirable in the SoC domain. Our proposed class-of-service based QoS architecture provides a low-cost solution for future SoCs, while is very powerful in providing system-level QoS-aware resource management in platforms where applications contend for cache space, NoC bandwidth and memory bandwidth simultaneously.

5. Related work

In future SoC environments, caches, NoC and memory are likely to be performance-critical resources shared between general-purpose cores and fixed-function or programmable IP blocks. Applications running on such SoC platforms typically have different hardware resource requirements as well as different sensitivity to allocation of all the resources on-chip. However, most existing works in QoS support focus on only one single resource. **Cache/memory QoS.** Most studies on cache/memory resource management targeted either fairness [32,42,47,30,31,48] or throughput [36], rather than providing differentiated services to individual applications. For example, Suh et al. [42] proposed a dynamic cache partitioning policy that improves system throughput while Kim et al. [23] proposed a cache partitioning policy that attempts to equalize the amount of performance degradation of equal-priority applications. Mutlu et al. [30] proposed a stall-time fair memory scheduler that can equalize the slowdown to each thread. Later, Mutlu et al. [31] proposed a parallelism-aware batch scheduler for memory which further improves both fairness and throughput. Zhang et al. [48] proposed an execution throttling technique to provide fair cache/bandwidth sharing in the multi-core systems.

Recent studies have recognized the need to provide different QoS levels to various applications running simultaneously on the platform. For example, Iyer proposed CQoS framework to address
priority-based QoS in shared caches [20]. Later, Iyer et al. proposed a QoS enabled memory architecture that can effectively manage cache/memory and introduced resource-based and performance-based metrics [21]. Rafique et al. proposed a hardware-based cache management technique that allows OS-level cache partitioning to ensure performance differentiation [37]. Yeh et al. proposed a distributed NUMA cache with an adaptive sharing mechanism that provides per-core QoS guarantees while maximizing overall system performance [47]. Nesbit et al. [32] proposed fair queuing memory scheduler to provide minimum service guarantee to each thread. The memory schedulers proposed by Mutlu et al. [30, 31] incorporate a priority-level for each thread so that higher priority threads can be scheduled faster. However, this priority-level cannot provide any bandwidth guarantee to threads. Akesson et al. [3] proposed a memory controller design that provides a guaranteed minimum bandwidth and maximum latency bound. These prior works established the need for application-specific QoS.

NoC QoS. Besides cache and memory QoS, researchers have also investigated QoS support in NoCs [46, 6, 13, 14, 18, 24, 28, 37, 10, 15]. Providing QoS for NoC can be done in the time domain by reserving bandwidth or in the space domain by reserving buffer space. Examples of works that leverage the time domain are:There [14] and Nostrum [28], which implement guaranteed performance services by scheduling the communication channels through time division multiplexing. There are also several works that utilize the space domain instead. For example, Bjerregaard et al. proposed MANGO, which utilizes virtual channels to provide connection-oriented guaranteed services and connection-less best-effort routing [6]. Bolotin et al. proposed QNoC router design, which uses separate buffers for each service level [7]. There are also works that partition time into epochs to provide bandwidth and latency guarantees. For example, Weber et al. [46] proposed an arbitration scheme based on epochs to provide different QoS services (low latency service, bandwidth service, and best-effort service). Lee et al. [24] proposed globally synchronized frames that allocates frames of buffers at the network interface in CMPs to provide bandwidth and latency guarantees to applications. Das et al. [10] proposed a batch-based application-aware prioritization policies in NoC to improve system throughput. This work also allows requests from higher priority applications to be scheduled faster within a batch in the network. However, the main goal of this work is to improve overall system throughput and fairness. It does not provide any bandwidth or latency guarantees to applications. Grot et al. [15] proposed a preemptive virtual clock NoC QoS scheme that provides bandwidth guarantees for applications. It does not require per-flow buffering in the router. However, packets may subject to drop, thus requiring an ACK network and source buffering for retransmission. This proposal works well in CMP domain. However, the hardware overhead cost is still high for SoCs where cost is a very important design factor. In our paper, we employed a weight-based arbitration for NoC QoS because it is cost-effective and aligns with our system-level QoS support mechanism. Other NoC QoS techniques which provide bandwidth and latency guarantees can be used in our framework in a similar way.

Joint resource QoS. Even though QoS support for cache/memory and NoC have been studied intensively, most prior works focus on just one individual resource, which does not ensure that system-level performance will be guaranteed or that all resources will be used efficiently. There has only been a few works that jointly explored chip resources: Bitirgen et al. proposed coordinated runtime resource management for CMPs using artificial neural networks [5]. The shared resources considered were cache, memory and power budget, ignoring the interconnect. In addition, the goal was to maximize total system performance, and cannot provide differentiated performance to individual applications. Nesbit et al. proposed a virtual private machine for joint resource allocation [35]. However, they did not implement/evaluate a scheme for doing it. Hansson et al. [17] proposed a composable and predictable multi-processor SoC platform to fully remove the interference between applications and provide hard real-time guarantees by time division multiplexing. However, the capacity unused by one application cannot be given to another application. Besides, the shared resources considered in this work are NoC and SRAM. There are no caches and memory controllers on the platform. Later, the Predator memory controller proposed by Akesson et al. [3] has been integrated into this framework [2]. Ebrahimi et al. [11] took an alternative approach to improve fairness in the entire shared memory system. Instead of partitioning shared resources to provide fairness, they proposed a source-based fairness control mechanism which constrains the rate at which an application’s memory requests are injected to the shared resources.

In this paper, we studied the effects of all three shared resources, caches, NoC and memories, and proposed a class-of-service based architecture that can map effectively to all three resources. Our focus was not on improving the individual QoS techniques for cache, NoC and memory. Rather, we employ existing QoS techniques and propose a unified NoC-based SoC architecture where the CoQoS can be exposed to OS, runtime layer and device drivers so that they can control or guide the class-of-service appropriately across the entire SoC platform. We show that using simple localized solutions for CoQoS-aware arbitration and a class-of-service model to tie them together and expose them to the OS and software is quite powerful and provides a flexible solution for future SoC architectures.

6. Conclusions and future work

This paper focused on the quality-of-service problem in NoC-based SoC architectures. We showed that cache, NoC and memory resources will be shared between cores and IP blocks and require a flexible QoS-aware arbitration policy and mechanism. We introduced a class-of-service based CoQoS architecture that is based on three major mechanisms: (a) class-of-service assignment where the class-of-service indication is exposed to the OS for guidance, (b) class-of-service mapping where each class-of-service is mapped (sometimes quite differently) to each of the shared resources in consideration, and (c) class-of-service based arbitration at each of the resources locally based on mapping tables. Through detailed simulation experiments of CoQoS, we showed that joint CoQoS mechanisms are very effective in providing system-level QoS-aware resource management that no individual QoS support can achieve. Furthermore, we also showed that providing different class-of-service assignment in each resource is important because of the difference in resource usage as well as interaction effects.

While we provided resource-based class-of-service hooks for SoCs in this paper, mapping performance-based class-of-service solutions on top of the resource-based hooks to allow performance QoS to be effective and efficient is a direction for future work. Performing detailed prototyping of CoQoS experiments in the context of many IP blocks and core is also an important future direction to provide valuable insights into the challenges for such highly heterogeneous NoC-based SoC architectures.

Acknowledgments

We would like to thank Jaipeed Moses of Intel Corporation for his help in understanding and using of the Aspen simulator, Tushar Krishna of MIT for his help on synthesis, and Zhen Fang of Intel Corporation for valuable discussions. We also thank anonymous reviewers for their valuable suggestions and feedbacks.
Bin Li is currently a research scientist in Intel Labs. She received the Ph.D. Degree in Electrical Engineering from Princeton University, Princeton, NJ, in 2009, and the B.S. Degree in Electrical Engineering from Xi’an Jiaotong University, Xi’an, China, in 2004. Her research interests include system-level power and performance modeling and analysis, on-chip interconnection networks, and shared resource managements in many-core systems.

Michael Leddige is a Senior Staff engineer at Intel Corporation in the Physical Technologies Lab of Intel Labs. Mike’s research interests include on-die and off-die interconnects, signal integrity and investigation of material properties for silicon packaging. Prior to joining Intel in 1996, he was a staff electrical analysis engineer for IBM Corporation. He has filed over 20 patents in the areas of board, package and die level interconnects. Mike has an M.S.E.E. and B.S. Degree in electrical engineering from the University of North Dakota.

Li Zhao is currently a research scientist in SoC Platform Architecture group, Intel Corporation. She received her Ph.D. Degree in Computer Science from University of California, Riverside in 2005. Her research interests include computer architecture, network computing and performance evaluation.

Michael Espig joined Intel in 1994 and is currently a research scientist in the Microprocessor and Programming Research Lab in Intel Labs. He received his M.S. in Computer Science from Georgia Institute of Technology. His research interests include CPU and server architecture, throughput computing, communications and performance evaluation.

Ravi Iyer is a Principal Engineer in Intel Labs. He directs research on SoC and CMP architectures. His research interests are in cache/memory subsystems, small core architectures, accelerators, interconnects, emerging workloads and performance analysis. Ravi has published over 100 papers in conferences and journals. He has also filed 30+ patent applications. Ravi frequently participates in conference program committees. He is currently General Co-Chair for the ISCA 2011 conference and Program Co-Chair for the ANCS 2010 conference. He is also currently an Associate Editor for ACM TACO. Ravi received his Ph.D. from Texas A&M University in 1999. He is a senior member of the IEEE.

Seung Eun Lee is an assistant professor in the Department of Electronic and Information Engineering at the Seoul National University of Science and Technology, Seoul, Korea. Dr. Lee has been involved in research and development in the area of computer architecture, multi-processor system-on-chip, network-on-chip, and VLSI design since he received Ph.D. Degree in electrical and computer engineering from the University of California at Irvine in 2008. He is a member of IEEE and has served as a program committee member on several international conferences.

Li-Shiuan Peh is Associate Professor of Electrical Engineering and Computer Science at Massachusetts Institute of Technology. She graduated with a Ph.D. in Computer Science from Stanford University in 2001, and a B.S. in Computer Science from the National University of Singapore in 1995. Her research focuses on low power interconnection networks, on-chip networks and parallel computer architectures, and is funded by several grants from the National Science Foundation, the DARPA MARCO Gigascale Systems Research Center and Interconnect Focus Center as well as Intel Corporation. She was awarded the CRA Anita Borg Early Career Award in 2007, Sloan Research Fellowship in 2006, and the NSF CAREER award in 2003.

Michael Espig joined Intel in 1994 and is currently a research scientist in the Microprocessor and Programming Research Lab in Intel Labs. He received his M.S. in Computer Science from Georgia Institute of Technology. His research interests include CPU and server architecture, throughput computing, communications and performance evaluation.

Michael Espig joined Intel in 1994 and is currently a research scientist in the Microprocessor and Programming Research Lab in Intel Labs. He received his M.S. in Computer Science from Georgia Institute of Technology. His research interests include CPU and server architecture, throughput computing, communications and performance evaluation.

Seung Eun Lee is an assistant professor in the Department of Electronic and Information Engineering at the Seoul National University of Science and Technology, Seoul, Korea. Dr. Lee has been involved in research and development in the area of computer architecture, multi-processor system-on-chip, network-on-chip, and VLSI design since he received Ph.D. Degree in electrical and computer engineering from the University of California at Irvine in 2008. He is a member of IEEE and has served as a program committee member on several international conferences.

Donald Newell was a Sr. Principal Engineer at Intel leading the SoC Platform Architecture group when this research was conducted. Don’s research interests include networking, server platform architecture, cache and memory hierarchy and heterogeneous computing. He has more than 60 referred journal or conference publications. Don has a B.S. in Computer Science from the University of Oregon.