In *12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2006)*, San Jose, CA, April 2006.

METERG: Measurement-Based End-to-End Performance Estimation Technique in QoS-Capable Multiprocessors

Jae W. Lee and Krste Asanović
MIT Computer Science and Artificial Intelligence Lab (CSAIL)
The Stata Center, 32 Vassar Street, Cambridge, MA 02139
{leejw, krste}@csail.mit.edu

Abstract

Multiprocessor systems present serious challenges in the design of real-time systems due to the wider variation of execution time of an instruction sequence compared to a uniprocessor system. Even if non-determinism is tightly controlled by adding conventional QoS support, it is generally difficult to find the minimal hardware resource request settings (e.g., memory bandwidth) for a given user-level performance goal (e.g., transactions per second). In this paper, we introduce the METERG (Measurement-Time Enforcement and Run-Time Guarantee) QoS system that provides an easy method of obtaining a tight estimate of the lower bound on end-to-end performance for a given configuration of resource reservations. Every QoS-capable block in the METERG system supports two operation modes for each task requiring QoS: **enforcement** *mode for estimating the lower bound on a task's execution time and* **deployment** *mode for maximizing its performance. We evaluate the effectiveness of our approach with an execution-driven multiprocessor simulator implementing the METERG QoS memory subsystem. We show that the performance lower bound is easy to obtain by simply running an application in enforcement mode, and that this estimated lower bound is tight.*

1 Introduction

As semiconductor technology continues to improve and reduces the cost of transistors, computing devices are becoming more pervasive. Computing systems are now called upon to process a wide variety of complex application workloads with demands for real-time or near real-time processing of real-world data streams. The difficulty of meeting power-performance goals with uniprocessor designs has led to the adoption of multiprocessor architectures in all scales of system, from handheld devices to rack-mounted servers. Existing techniques for managing real-time systems can provide hard guarantees for small applications running on uniprocessor systems, but there is a growing need to support complex multiprogrammed workloads, including both soft real-time and best-effort tasks, running on multiprocessor systems. For example, a handheld device might support various types of media codec running alongside best-effort background tasks on an embedded multiprocessor core [14]. As another example, the provider of a co-located server farm may use OS virtualization to reduce computing costs by supporting multiple customers on a single multiprocessor server but would like to offer varying levels of guaranteed performance.

Conventional multiprocessor systems present severe challenges when trying to provide even soft real-time guarantees. Multiple concurrent processes exhibit complex non-deterministic interactions when sharing common system resources, such as the memory system, causing execution time to exhibit much wider variation than in a uniprocessor. As the number of processors increases, problems are usually exacerbated. For example, in Intel's Paragon system [8] with 1,024 nodes, the theoretical worst-case message delivery time is several days [10]. Although extreme, this example underscores the need for Quality-of-Service (QoS) support in multiprocessor systems. A typical QoS system has hardware-enforced resource allocation controlled by an operating system, which allocates resources and performs admission control to ensure QoS resources are not oversubscribed.

Even though QoS-capable hardware components provide performance isolation and allow tighter bounds on a task's execution time, a major problem still remains: users would like to request only the minimal resources needed to meet the desired performance goal. That is, translating a high-level performance goal for a complex application (e.g., transactions per second) into minimal settings for the hardware QoS components (e.g., memory bandwidth and latency) is challenging and usually intractable.

One approach is for the user to try to measure the performance of their code running on the target system with varying settings of the resource controls. Unfortunately, a conventional QoS-capable shared resource usually distributes unused resources to the sharers so as to maximize the overall system's throughput. Consequently, even with an identical setting of resource reservation parameters, the observed performance of an instruction sequence fluctuates widely depending on how much additional resource it receives. Even if competing jobs are run to cause severe resource contention, we cannot safely claim the observed performance is really representative of worst case contention.

In this paper, we propose a new measurement-based technique, METERG (Measurement-Time Enforcement and Run-Time Guarantee), where QoS blocks are modified to support two modes of operation. During performance measurement, resource guarantees in the QoS blocks are treated as an upper bound, while during deployment, resource guarantees are treated as a lower bound. The METERG QoS system enables us to easily estimate the maximum execution time of the instruction sequence produced by a program and input data pair, under the worst-case resource contention. In this way, we can guarantee measured performance during operation.

Unlike static program analysis techniques, which cover all possible input data, the METERG methodology is based on measurement of execution time for a certain input set. However, this is still useful, especially in soft real-time systems, where we can use representative input data to get a performance estimate and add a safety margin if needed. In a handheld device, for example, a small number of deadline violations may degrade end users' satisfaction but are not catastrophic. Such potential violations are tolerable in exchange for the reduced system cost of shared resources. In contrast, hard real-time systems, such as heart pacemakers, car engine controllers, and avionics systems, justify the extra cost for dedicated resources with possibly suboptimal throughput.

The rest of this paper is organized as follows. In Section 2, we review related work. In Section 3, we first overview the METERG QoS system with the set of assumptions made, followed by the details of the system. In Section 4, we give a preliminary evaluation of a METERG QoS system using our execution-driven multiprocessor simulator. Finally, we summarize our work and suggest directions for future research.

2 Related Work

The end-to-end performance (e.g. execution time) of a program running on a multiprocessor is the result of complex interactions among many factors. We identify three major factors causing non-deterministic performance. First, performance depends on the input data to the program, as this determines the program's control flow and data access patterns. Second, even if we run the program with identical input data multiple times, the performance of each run can vary widely because of the different degrees of contention in accessing shared system resources. Third, *performance anomalies* [23, 18] in microprocessors also affect the end-to-end performance.

The first factor has been actively investigated in the real-time system community in Worst Cast Execution Time (WCET) studies [26, 3, 16, 7, 21, 2]. Researchers have made efforts to estimate a program's WCET across all possible input data. The third factor is not problematic as long as a processor is *performance monotonic*, i.e., longer access time to a resource always leads to equal or longer execution time. Performance monotonicity holds for simpler processors, but not for some complex out-of-order processors. A code modification technique [23], or an extra safety margin can be added to the estimated lower bound of performance to cope with performance non-monotonicity. We do not cover this issue further in this paper. Our work reduces the non-determinism caused by the second factor: resource contention.

The goal of QoS research is to limit the impact of the non-determinism coming from resource contention. QoS was originally introduced in the context of long-haul IP networks, where a network that can provide different levels of packet delivery service in terms of latency, bandwidth, or both, is said to support QoS [15]. The idea of QoS has since been applied to other shared resources such as multiprocessor interconnection networks [10, 25, 19], memory subsystems [4, 20], and storage and I/O systems [1, 24].

Recently, researchers have investigated QoS support for smaller-scale shared resources, especially those within processors. Iyer addresses the QoS issue in shared caches of CMP platforms [9]. Kalla et al. [13] allocate shared resources to concurrent threads on a simultaneous multithreaded (SMT) processor, such as instruction issue bandwidth, based on a fixed static priority.

Most QoS research projects mentioned above, however, focus on bounding the non-determinism of individual resources, and their link to the user-observable end-to-end performance still remains unclear. Therefore, users often have to rely on the high-level knowledge of a program in setting up the resource reservation parameters appropriately to meet a certain end-to-end performance goal.

One interesting exception is the work done by Cazorla et al [5]. On an SMT processor, for a given performance goal specified by a user metric (instructions-per-cycle, or IPC), a dynamic adjustment is made to allocation of shared resources such as renaming registers, instruction/load/store queues, L2 cache, to meet the IPC goal. However, the adjustment is made based on observations during the previous sampling period, which could be in a different program phase with completely different IPC characteristics.

Note that our work differs from real-time scheduling research [17, 22, 11, 6]. In the conventional formulation of the real-time scheduling problem, it is assumed that, for a given set of n tasks ($T_1, T_2, \cdots, T_n$), the execution time of each task (or its upper bound) is known in advance. Then each task is scheduled for execution at a particular time. In contrast, our work provides performance guarantees to ensure the estimated execution time of each task is not violated in practice because of severe resource contention. Our work is independent of the scheduling algorithm.

In order to estimate a lower bound on end-to-end performance, existing frameworks for WCET estimation can be used [26, 3, 16, 7, 21, 2]. By assuming all memory accesses take the longest possible latency, the execution time, given worst-case resource contention, can be calculated. However, use of WCET is limited. Most WCET estimation techniques require several expensive procedures: elaborate static program analysis, accurate hardware modeling, or both. These techniques cannot be easily applied, if at all, to complex hardware components and programs.

3 The METERG QoS System

3.1 Overview

In the METERG QoS system, each process requiring QoS support (QoS process for brevity) can run in two operation modes: *enforcement* and *deployment*. In enforcement mode, a QoS process cannot take more than its guaranteed resource share from each QoS block even when there are additional resources available. In deployment mode, however, the process is allowed to use any available additional resources (we do not discuss policies for sharing excess resources among multiple QoS processes in this paper). If a QoS block supports the two operation modes as described above, we call it a *METERG QoS block*. If every shared resource within a system is a METERG QoS block, the system is termed a METERG system.

With a METERG system, a user first measures execution time of a given code sequence with a given input in enforcement mode with given resource reservation parameters. The METERG system then guarantees that a later execution of the same code sequence in deployment mode will perform as well as or better than the previous execution in enforcement mode, provided the same parameters are used for resource reservation. More specifically, a user takes the following steps:

- **Step 1** Given an input set and a performance goal for execution time (T_{GOAL}), the user runs the program with the input set in enforcement mode. The system provides APIs to set resource reservation parameters to arbitrary values. (Setting these parameters to reasonable values can reduce the number of iterations in Step 2.)

- **Step 2** Iterate Step 1 with various resource reservation vectors to find a minimal resource request setting that still meets the performance goal. If the measured execution time (T_{MEAS}) is smaller than $T_{GOAL} - T_{MARGIN}$ (where T_{MARGIN} is a safety margin, if needed), the user may try a smaller resource reservation vector. If not, they may increase the amount of reserved resources. We believe that this process can be automated, but do not consider an automated search in this paper.

- **Step 3** The estimated minimal vector of resource reservation is stored away, for example, as an annotation in the binary. There can be multiple minimal vectors, or pareto-optimal points, meeting the performance goal, to enable better schedulability for the program.

- **Step 4** Later, when the program is run in deployment mode, the OS uses a stored resource reservation vector to configure the hardware accordingly. Note that the system may reject the resource reservation request if oversubscribed. If the request is accepted, the execution time (T_{DEP}) is guaranteed to be no greater than $T_{GOAL} - T_{MARGIN}$. Any performance slack ($T_{GOAL} - T_{MARGIN} - T_{DEP}$) of the program in runtime can be exploited to improve the system throughput or reduce energy consumption as described in [2], but we do not address this issue further.

Figure 1 depicts the METERG system model. There are n processors and m METERG QoS blocks (e.g. memory, I/Os). These QoS blocks are shared among all n processors, and can reserve a certain amount of resource for each processor to provide guaranteed service (e.g. bandwidth, latency). Unlike conventional guaranteed QoS blocks, they accept an extra parameter, $OpMode_i$, from a processor to request the QoS operation mode. If the processor requests enforcement mode, the strict upper bound on runtime usage of a resource is enforced in every METERG QoS block. If the processor requests deployment mode, it can use additional otherwise unclaimed resources.

We assume that the j-th METERG QoS block (where $1 \leq j \leq m$) maintains a resource reservation vector of n real numbers, $(x_{1,j}, x_{2,j}, \ldots, x_{n,j})$, where the i-th entry ($x_{i,j}$) specifies the fraction of available resource reserved for processor i. We use a single number per processor for simplicity. For example, a shared memory bus may take the vector to determine the fraction of available bandwidth it will allocate to each processor, and set up the bus arbiter properly (time sharing). Another example could be a shared cache to determine how many ways or sets it will allocate to

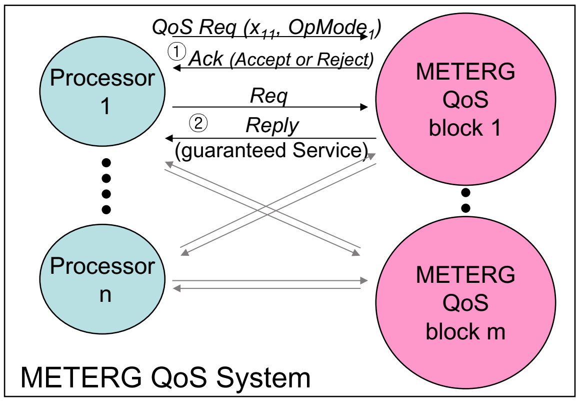

QoS Req (x_{11}, OpMode$_1$)
① Ack (Accept or Reject)
Req
② Reply
(guaranteed Service)
Processor 1
Processor n
METERG QoS block 1
METERG QoS block m
METERG QoS System
Phase ①: Resource reservation phase (setting up a service level agreement)
Phase ②: Operation phase
(Requests are serviced with bandwidth/latency guarantees.)

Figure 1. The METERG QoS system. Each QoS block takes an extra parameter (OpMode) as well as a resource reservation parameter ($x_{i,j}$) for each processor.

each processor (space sharing). The range of $x_{i,j}$ is between 0 and 1 by convention, where $x_{i,j}$=1 indicates that the i-th processor will monopolize all of the available resource from the QoS block, and $x_{i,j}$=0 indicates no guaranteed share for the i-th processor or only a *best-effort* service from the QoS block. Note that guaranteed services can be provided only to a limited number of sharers meeting the condition $\sum_{i=1}^{n} x_{i,j} \leq 1$.

Before further discussion, we present a set of assumptions for the rest of this paper:

- We limit ourselves to dealing with *single-threaded* programs in the multiprogrammed environment. We do not consider interprocess communication or shared objects among multiple threads running on different processors.
- At any point in time, there can be only one (or no) active thread running on a processor – that is, there is no interference such as cache contamination between multiple threads on the same processor. Combined with the previous assumption, this implies a one-to-one relationship between running processes and processors.
- We neither consider the performance variation from OS operations (e.g. flushing cache or branch predictor states after a context switch), nor address OS design issues (e.g. admission control, scheduling). A scheduled program runs to completion without preemption by other processes.
- We are not concerned about the performance variation coming from the processor's initial state, whose performance impact can be easily amortized over a program's longer execution time in most cases.

3.2 Safety-Tightness Tradeoff: Relaxed and Strict Enforcement Modes

We propose two types of enforcement mode,*relaxed* and *strict*, to reflect a tradeoff between *safety* and *tightness* of performance estimation. The estimated execution time should not be violated (safety), but be as close as possible to typical execution times (tightness). So far, we have accounted for the amount of allocated resources (i.e., bandwidth), but not the access latency to each QoS block. However, the execution time of a program is heavily dependent on latency as well. In Figure 2, we show an example of two network connections between Node 0 (N0) and Node 3 (N3) with a simple fixed frame-based scheduling. Although Connection B receives twice as much bandwidth as Connection A, its latency is longer. This longer latency could lead to a longer execution time for a given instruction sequence. Therefore, we introduce strict enforcement mode to enforce constraints on both bandwidth and latency. Condition 1 below (Bandwidth) is met by both relaxed and strict enforcement modes, but Condition 2 (Latency) is met only in strict enforcement mode.

Condition 1 *For a given processor i, and* $\forall j (1 \leq j \leq m)$,

$$Bandwidth_{DEP}(x_{i,j}) \geq Bandwidth_{ENF}(x_{i,j})$$

Condition 2 *For a given processor i, and* $\forall j (1 \leq j \leq m)$,

$$MAX\{Latency_{DEP}(x_{i,j})\} \leq MIN\{Latency_{ENF}(x_{i,j})\}$$

Assuming performance monotonicity of a processor, we can safely claim that the estimated performance lower bound in strict enforcement mode will not be violated in deployment mode. The estimated lower bound of execution time in strict enforcement mode is safer than that in the relaxed enforcement mode but is not as tight.

3.3 METERG QoS Memory Subsystem: An Example

We can build METERG QoS blocks meeting Condition 1 by slightly modifying conventional QoS blocks supporting per-processor resource reservation. The QoS scheduling algorithm is modified to exclude processors running in enforcement mode when distributing unclaimed slots.

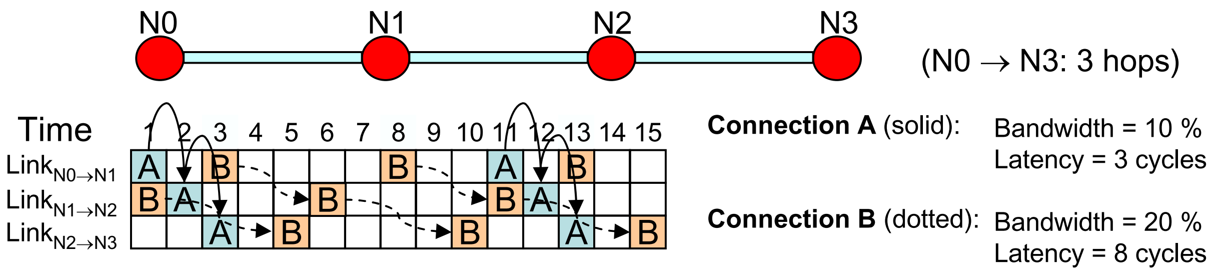

N0
N1
N2
N3
(N0 → N3: 3 hops)
Time
Link N0→N1
Link N1→N2
Link N2→N3
Connection A (solid): Bandwidth = 10 % Latency = 3 cycles
Connection B (dotted): Bandwidth = 20 % Latency = 8 cycles

Figure 2. An example of having longer latency in spite of more bandwidth.

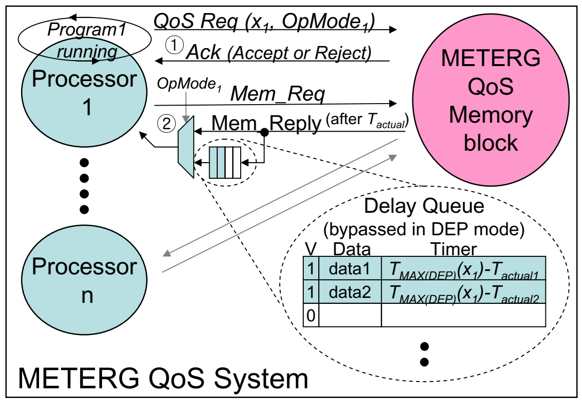
Latency guarantees are more difficult to implement than bandwidth guarantees. Fortunately, although conventional latency guarantees require a certain time bound not to be exceeded (absolute guarantee), the METERG system requires only the inequality relationship in Condition 2 to hold (relative guarantee). Given METERG QoS blocks capable of bandwidth guarantees, we can implement the METERG system supporting strict enforcement mode by inserting a delay queue between every QoS block-processor pair on the reply path.

Figure 3 depicts an example of a METERG QoS memory subsystem supporting strict enforcement mode. For the rest of this paper, we drop the QoS block identifier in resource reservation parameters, for we only have one shared resource for illustrative purposes, i.e., $x_i \equiv x_{i,j}$. Because the memory block is capable of resource reservation, we can assume that the latency to this block in deployment mode is upper bounded. The upper bound, $T_{MAX(DEP)}(x_1)$, is determined by the resource reservation parameter x_1. Note that the bounded latency is not guaranteed in enforcement mode, because the program receives no more resource than specified by x_1.

The delay queue is used only in enforcement mode; it is simply bypassed in deployment mode. If a memory access in enforcement mode takes T_{actual} cycles, which is smaller than $T_{MAX(DEP)}(x_1)$ due to lack of contention, the network interface (NI) places the reply message in the delay queue with the entry's timer set to $T_{MAX(DEP)}(x_1) - T_{actual}$. The timer value gets decremented every cycle. The NI will defer signaling the arrival of the message to the processor until the timer expires. Hence, the processor's observed memory access latency is no smaller than $T_{MAX(DEP)}(x_1)$ and Condition 2 holds.

Because the processor needs to allocate a delay queue entry before sending a memory request in enforcement mode, a small delay queue may cause additional memory stall cycles. However, this does not affect the safety of the measured performance in enforcement mode, but only its tightness.

The memory access time under the worst-case contention for a given resource reservation parameter (x_1), $T_{MAX(DEP)}(x_1)$, depends on the hardware configuration and the scheduling algorithm. In some systems, the latency lower bound is known or can be calculated [15, 4, 10]. For example, for connection A in the setup shown in Figure 2, it is straightforward. Because it uses simple frame-based scheduling with ten time slots in each frame and at least one out of every ten is given to the process, each hop cannot take more than 10 time slots (if there is no buffering delay caused by a large buffer). Therefore, the worst-case latency between Node 0 and 3 will be 30 time slots. If the estimation process is not trivial, however, one may use an observed worst-case latency with some safety margin.



(Only the operations from Processor 1 are shown.)

Phase ①: Resource reservation phase (setting up a service level agreement)
Phase ②: Operation phase
(Requests are serviced with bandwidth/latency guarantees.)

Figure 3. An implementation of the METERG system supporting strict enforcement mode. We use delay queues to meet the latency condition (Condition 2) required by strict enforcement mode.

4 Evaluation

In this section, we present a preliminary evaluation of the METERG system. We evaluate the performance of a single-threaded application with different degrees of resource contention on our system-level multiprocessor simulator.

4.1 Simulation Setup

We have added a strict METERG memory system to a full-system execution-driven multiprocessor simulator based on Bochs IA-32 emulator [12]. Figure 4 depicts the organization of our simulated platform. Although our simulator only supports the METERG QoS in the memory subsystem, we believe that the QoS support can be extended to other shared I/O devices (e.g. disk, network).

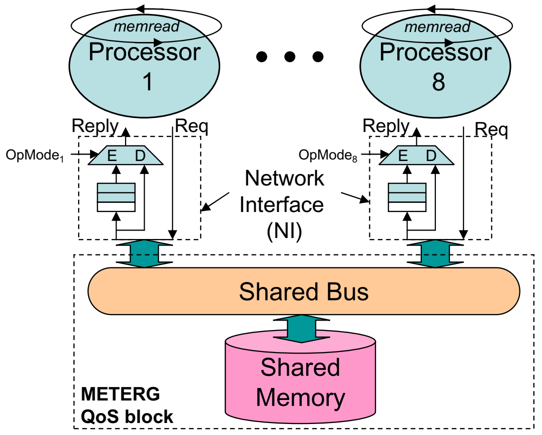


Figure 4. A simple bus-based METERG system. The memory subsystem is capable of METERG QoS.

Our processor model is a simple in-order core capable of executing one instruction per cycle, unless there is a cache miss. A cache miss is handled by the detailed memory system simulator and takes a variable latency depending on the degree of contention. The processor's clock speed is five times faster than the system clock (e.g. 1 GHz processor with 200 MHz system bus, which is reasonable in embedded systems). The processor has a 32-KB direct-mapped unified L1 cache, but no L2 cache. The cache is blocking, so there can be at most one outstanding cache miss by any processor.

Our detailed shared memory model includes primary caches and a shared bus interconnect, and we have augmented it with the METERG QoS support. We have used a simple magic DRAM which returns the requested value in the next bus cycle; otherwise, long memory access latencies of the detailed DRAM, combined with our blocking caches, would make the memory bandwidth underutilized. Note that QoS support is meaningful only when there is enough resource contention. Severe memory channel contention is feasible in multiprocessor embedded systems where resources are relatively scarce and bandwidth requirements for applications (e.g. multimedia) are high.

The shared bus interconnect divides a fixed-size time frame into multiple *time slots*, which are the smallest units of bandwidth allocation, and implements a simple time division multiplexing (TDM) scheme. For example, if Processor 1 requests QoS with the resource allocation parameter (x_1) of 0.25, one out of every four time slots will be given to the processor. Hence, the access time is bounded by $\lceil 1/x_1 \rceil$=4 time slots in this case. An unclaimed time slot can be used by any other processors not in enforcement mode (work conserving).

We use a synthetic benchmark, called `memread`, for our evaluation to mimic the behavior of an application whose performance is bounded by the memory system performance. It runs an infinite loop which accesses a large memory block sequentially to generate cache misses, with a small amount of bookkeeping computation in each iteration.

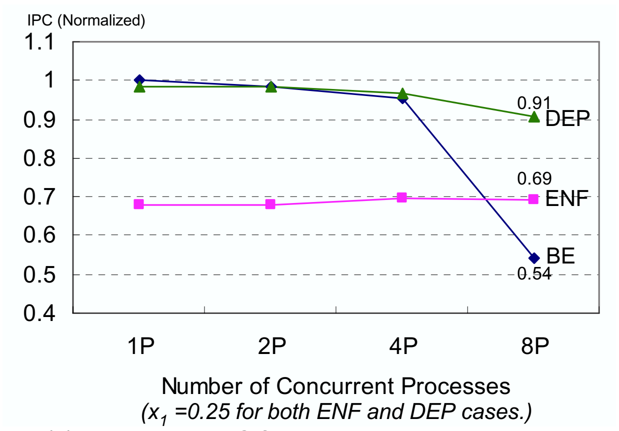
4.2 Simulation Results

Performance Estimation

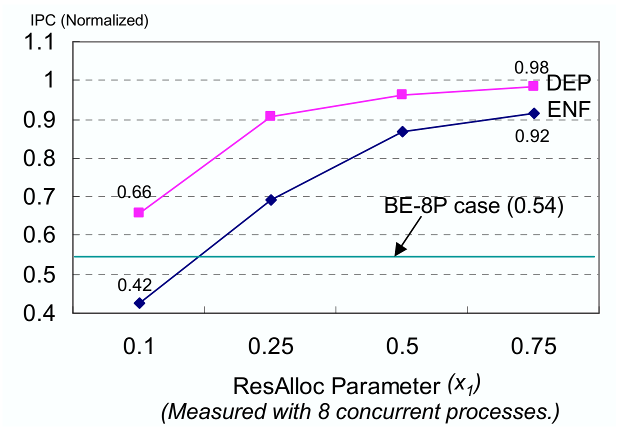
Figure 5 compares the performance of `memread` in various configurations with different operation modes ($OpMode$), degrees of contention, and resource allocation parameters (x_1). We use instructions per cycle (IPC) as our performance metric and all IPCs are normalized to the best possible case (denoted by `BE-1P`), where a single `memread` in best-effort mode monopolizes all the system resources. In best-effort (BE) mode, all processes run without any QoS support. In enforcement (ENF) or deployment (DEP) mode, only one process runs in QoS mode (either enforcement or deployment), and the remaining concurrent processes run in best-effort mode to generate memory contention. The figure depicts the single QoS process' performance.

In Figure 5(a), we first measure performance with varying degrees of resource contention. Without any QoS support (denoted by `BE`), we observe the end-to-end performance degradation of a single process by almost a factor of 2, when the number of concurrent processes executing `memread` increases from 1 (`BE-1P`) to 8 (`BE-8P`).

On the other hand, a QoS process in either enforcement or deployment mode is well protected from the dynamics of others. At any given time, a process in deployment mode always outperforms its counterpart in enforcement mode for a given resource allocation parameter (x_1). There is a significant performance gap between the two to give a



(a) Normalized IPC for variable resource contention.



(b) Normalized IPC for variable *ResAlloc* parameters.

Figure 5. Performance of `memread` in various configurations. BE, ENF, and DEP stand for best-effort, enforcement-mode, and deployment-mode execution, respectively. In (a), as the number of concurrent processes increases from 1 to 8, the performance of a process degrades by 46 % without QoS support, but only by 9 % in deployment mode. The estimated performance from a measurement in strict enforcement mode indicates the performance degradation for the given resource reservation to be 31 % in the worst case. In (b), we observe that the performance estimation in strict enforcement mode becomes tighter as the resource allocation parameter (x_1) increases.

safety margin for estimated performance. The performance gap can be explained by two factors. First, there is extra bandwidth given to the process in deployment mode, which would not be given in enforcement mode. Second, regardless of the actual severity of contention, every single memory access in enforcement mode takes the longest possible latency. Note that, although rare, this could happen in a real deployment-mode run. Hence, an enforcement-mode execution provides a safe and tight performance lower bound for a given x_1. Because a memory access in enforcement mode always takes the worst possible latency, there is little variation of the performance across 1 (`ENF-1P`) through 8 (`ENF-8P`) concurrent processes.

In Figure 5(b), we run the simulation with different resource reservation parameters. We observe that as we increase the parameter value, the performance gap between the two modes shrinks. This is because extra bandwidth beyond a certain point gives only a marginal performance gain to the QoS process in deployment mode, but improves the performance in enforcement mode significantly by reducing the worst-case memory access latency.

Interactions among Processes

Because we have dealt with a single QoS process so far, a question naturally arises about the interactions of multiple concurrent QoS processes. Figure 6 shows the performance variation when multiple QoS processes are contending against each other to access the shared memory.

The figure demonstrates that, even if we increase the number of QoS processes from one to four, the performance of QoS processes in deployment mode degrades very little (by less than 2 %) for a given parameter (x_i=0.25) and the performance lower bound estimated by an enforcement-mode execution is strictly guaranteed. The amount of reserved resource for each process is given in the resource allocation vector. Note that x_i=0 means no resource is reserved for processor i (best-effort), and that we use x_4=0.20 rather than x_4=0.25 in the case of 4 QoS + 4 BE so as not to starve the best-effort processes.

As we increase the total amount of resources reserved for QoS processes, the performance of best-effort processes is degraded as expected. We observe that the system provides fairness so that their execution time differs only by less than 1 %. Fairness is also provided to the QoS processes having the same resource reservation parameter.

5 Conclusion

Although conventional QoS mechanisms effectively set a lower bound on a program's performance for a given resource reservation parameter, it is not easy to translate a user-level performance metric into a vector of hardware resource reservations. To facilitate performance estimation,

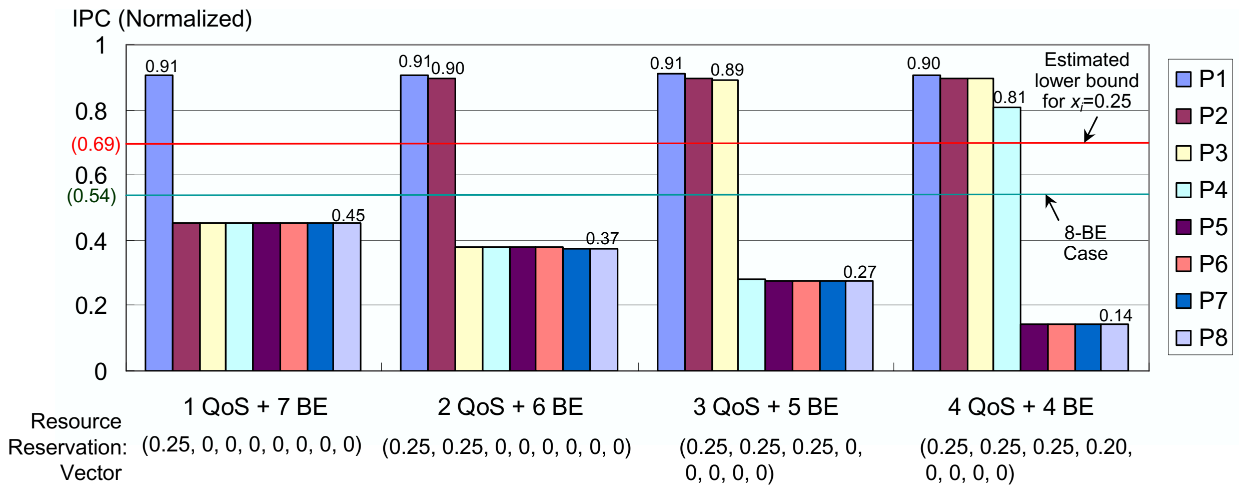


Figure 6. Interactions between QoS and Best-effort (BE) processes running `memread`. All QoS processes are running in deployment mode. Even if the number of QoS processes increases, the performance of QoS processes degrades very little as long as the system is not oversubscribed.

we argue for having every QoS-capable component support two operation modes: enforcement mode for estimating end-to-end performance and deployment mode for maximizing performance with a guaranteed lower bound. Our approach does not involve any expensive program analysis or hardware modeling often required in conventional approaches for performance estimation. Instead, we use simple measurement. In order to demonstrate its effectiveness, we have implemented a multiprocessor system simulator and estimated the lower bound on the execution time of a bandwidth-intensive synthetic benchmark.

Acknowledgments

We thank the SCALE group at MIT and the anonymous reviewers for their helpful comments. This work is partially supported by NSF CAREER Award CCR-0093354, the Cambridge-MIT Institute, and an equipment donation from Intel corporation.

References

[1] C. Akinlar and S. Mukherjee. Bandwidth Guarantee in a Distributed Multimedia File System Using Network Attached Autonomous Disks. In *IEEE Real Time Technology and Applications Symposium*, 2000.

[2] A. Anantaraman, K. Seth, K. Patil, E. Rotenberg, and F. Mueller. Virtual Simple Architecture (VISA): Exceeding the Complexity Limit in Safe Real-time Systems. In *ISCA '03: Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 350–361, New York, NY, USA, 2003. ACM Press.

[3] R. Arnold, F. Mueller, D. B. Whalley, and M. Harmon. Bounding Worst-case Instruction Cache Performance. In *Proceedings of the Real-Time Systems Symposium (RTSS)*, Dec. 1994.

[4] B. Case. First Trimedia Chip Boards PCI Bus. *Microprocessor Report*, Nov 1995.

[5] F. J. Cazorla, A. Ramírez, M. Valero, P. M. W. Knijnenburg, R. Sakellariou, and E. Fernández. QoS for High-Performance SMT Processors in Embedded Systems. *IEEE Micro*, 24(4):24–31, 2004.

[6] A. El-Haj-Mahmoud and E. Rotenberg. Safely Exploiting Multithreaded Processors to Tolerate Memory Latency in Real-time Systems. In *CASES '04: Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 2–13, New York, NY, USA, 2004. ACM Press.

[7] C. A. Healy, R. D. Arnold, F. Mueller, M. G. Harmon, and D. B. Walley. Bounding Pipeline and Instruction Cache Performance. *IEEE Trans. Comput.*, 48(1):53–70, 1999.

[8] Intel Corp. Paragon XP/S Product Overview. Intel Corporation, 1991.

[9] R. Iyer. CQoS: a Framework for Enabling QoS in Shared Caches of CMP Platforms. In *ICS '04: Proceedings of the 18th annual International Conference on Supercomputing*, pages 257–266, New York, NY, USA, 2004. ACM Press.

[10] Jae H. Kim and Andrew A. Chien. Rotating Combined Queueing (RCQ): Bandwidth and Latency Guarantees in Low-cost, High-performance Networks. In *ISCA '96: Proceedings of the 23rd Annual International Symposium on*

Computer Architecture, pages 226–236, New York, NY, USA, 1996. ACM Press.

[11] R. Jain, C. J. Hughes, and S. V. Adve. Soft Real- Time Scheduling on Simultaneous Multithreaded Processors. In *IEEE Real-Time Systems Symposium*, 2002.

[12] K. Lawton. Bochs: The cross-platform IA-32 emulator. `http://bochs.sourceforge.net`.

[13] R. N. Kalla, B. Sinharoy, and J. M. Tendler. IBM Power5 Chip: A Dual-Core Multithreaded Processor. *IEEE Micro*, 24(2):40–47, 2004.

[14] K. Krewell. ARM Opens Up to SMP. *Microprocessor Report*, May 2004.

[15] Larry L. Peterson and Bruce S. Davie. *Computer Networks: a Systems Approach, 2nd Ed.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

[16] S. Lim, Y. Bae, G. Jang, B. Rhee, S. Min, C. Park, H. Shin, K. Park, and C. Kim. An Accurate Worst Case Timing Analysis Technique for RISC Processors. In *Proceedings of the 15th IEEE Real-Time Systems Symposium (RTSS)*, pages 97–108, 1994.

[17] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 20(1):46–61, 1973.

[18] N. Kushman. Performance Nonmonotonicities: A Case Study of the UltraSPARC Processor. Technical Report MIT/LCS/TR-782, 1998.

[19] Sathish Gopalakrishnan and Lui Sha and Marco Caccamo. Hard Real-Time Communication in Bus-Based Networks. In *Proceedings of the 25th IEEE Real-Time Systems Symposium (RTSS 2004), 5-8 December 2004, Lisbon, Portugal*, pages 405–414. IEEE Computer Society, 2004.

[20] M. Shalan and V. J. Mooney. A Dynamic Memory Management Unit for Embedded Real-time System-on-a-chip. In *CASES '00: Proceedings of the 2000 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 180–186, New York, NY, USA, 2000. ACM Press.

[21] F. Stappert, A. Ermedahl, and J. Engblom. Efficient longest executable path search for programs with complex flows and pipeline effects. In *CASES '01: Proceedings of the 2001 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 132–140, New York, NY, USA, 2001. ACM Press.

[22] J. Sun and J. Liu. Synchronization Protocols in Distributed Real-Time Systems. In *ICDCS '96: Proceedings of the 16th International Conference on Distributed Computing Systems (ICDCS '96)*, Washington, DC, USA, 1996. IEEE Computer Society.

[23] Thomas Lundqvist and Per Stenstrom. Timing Anomalies in Dynamically Scheduled Microprocessors. In *RTSS '99: Proceedings of the 20th IEEE Real-Time Systems Symposium*, Washington, DC, USA, 1999. IEEE Computer Society.

[24] J. Wu and S. Brandt. Storage Access Support for Soft Real-Time Applications. In *IEEE Real Time Technology and Applications Symposium*, 2004.

[25] K. H. Yum, E. J. Kim, C. R. Das, and A. S. Vaidya. MediaWorm: A QoS Capable Router Architecture for Clusters. *IEEE Trans. Parallel Distrib. Syst.*, 13(12):1261–1274, 2002.

[26] N. Zhang, A. Burns, and M. Nicholson. Pipelined Processors and Worst Case Execution Times. Technical Report University of York - CS 198, 1993.