QoS Management on Heterogeneous Architecture for Parallel Applications

Ying Zhang[*], Li Zhao[†], Ramesh Illikkal[†], Ravi Iyer[†], Andrew Herdrich[†], Lu Peng[*]

[*]*Department of Electrical and Computer Engineering*
Louisiana State University, Baton Rouge, LA
{yzhan29, lpeng}@lsu.edu

[†]*Intel Labs*
Intel Corporation, Hillsboro, OR
{li.zhao, ramesh.g.illikkal, ravishankar.iyer, andrew.j.herdrich}@intel.com

***Abstract*—Quality of service (QoS) management is widely employed to provide differentiable performance to programs with distinctive priorities on conventional chip multi-processor (CMP) platforms. Recently, heterogeneous architecture integrating diverse processor cores on the same silicon has been proposed to better serve various application domains and it is expected to be an important design paradigm of future processors. Therefore, the QoS management on emerging heterogeneous systems will be of great significance. On the other hand, parallel applications are becoming increasingly important in modern computing community in order to explore the benefit of thread-level parallelism on CMPs. However, considering the diverse characteristics of thread synchronization, data sharing, and parallelization pattern, governing the execution of multiple parallel programs with different performance requirements becomes a complicated yet significant problem. In this paper, we study QoS management for parallel applications running on heterogeneous CMP systems. We comprehensively assess a series of task-to-core mapping policies on a real heterogeneous hardware (QuickIA) by characterizing their impacts on performance of individual applications. Our evaluation results show that the proposed QoS policies are effective to improve the performance of programs with highest priority while striking good tradeoff with system fairness.**

Keywords—Heterogeneous, quality-of-service, parallel application

I. Introduction

In the past decade, multi-core processors have become the mainstream to provide high performance while encapsulating the processor power consumption within a reasonable envelope. Most commercial multi-core processors to date are homogeneous by replicating a number of identical cores on a single chip; however, with the rapid development of modern processors, computer scientists propose heterogeneous architectures which integrate a diversity of processors onto the same die to better serve applications from different domains. Heterogeneous architecture can be designed in various ways and several products have already been released by current industry. The AMD APU and Nvidia Tegra3 combine CPU and graphics processing units (GPU) together, aiming to mitigate the bottleneck caused by the data transfer on PCIe bus. The ARM big.LITTLE multiprocessor implements another important design philosophy by including a powerful Cortex A15 and a smaller yet power-saving Cortex A7 on the same chip, in order to deliver remarkable energy efficiency via smart task scheduling. In this paper, we concentrate on the second category where processors on chip are all general-purpose CPUs but deviating in performance, power and area.

In a practical execution scenario where a number of applications are simultaneously running on a chip-multiprocessor (CMP), the quality of service (QoS) that each individual program gets from the underlying platform largely depends on the characteristics of its co-runners and resource management schemes engaged by the system. Figure 1 illustrates the architecture of a QoS-aware CMP system where the QoS policies are employed in different hierarchies: 1) core level, 2) cache level, and 3) memory level. This hierarchical infrastructure for QoS management secures that distinctive applications (e.g., single-threaded, multi-threaded, domain-specific, etc) executed on the common platform match their respective performance expectation. To date, QoS polices have been extensively studied in cache level (cache size partitioning) and memory level (memory bandwidth allocation) in previous works [7][8][11][19] since they assume homogeneous platforms where appropriate allocation of shared resources is critical to the performance of individual programs. However, while switching to a heterogeneous platform equipped with diverse processors, core level QoS management needs to be carefully considered because the task-to-core mapping will impose significant impact on the performance of individual programs. In this situation, an application should be assigned to either powerful big processors (B) or slower small cores (S) based on its characteristic and priority, in order to achieve the desired QoS targets.

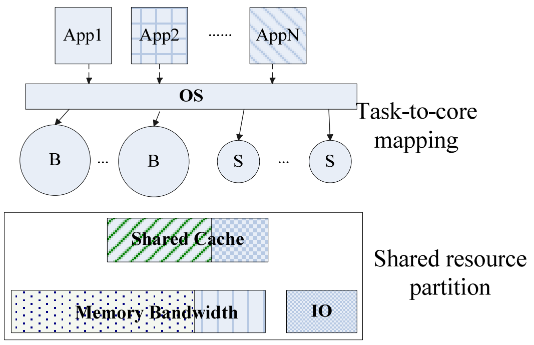
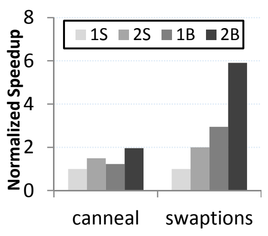
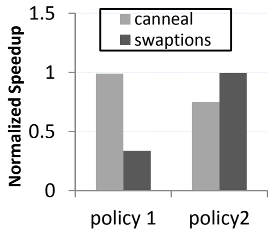


Fig. 1. QoS-aware heterogeneous CMP system: Different types of Simultaneous Workloads

While running multi-programmed single-threaded workloads on homogeneous multi-core platforms are challenging already, things become even more complicated when multiple parallel applications are executed in a heterogeneous CMP system in concurrency. Unlike single-threaded programs, parallel applications launch a large number of threads that require more than one processor for execution to fully explore the thread-level parallelism (TLP). Furthermore, contemporary multi-threaded applications demonstrate significantly different characteristics including parallelization pattern, data sharing degree, synchronization frequency, etc. As a consequence, the amount and types of cores that the system should assign to each individual application deserve careful consideration. Figure 2 demonstrates



(a)



(b)

Fig. 2. Illustrating the need for QoS: (a) performance scaling with core numbers and types (b) performance comparison with different policies (policy 1: *canneal* on 2 big cores and *swaptions* on 2 small cores. policy2: *canneal* on 2 small cores and *swaptions* on 2 big cores)

an example to highlight the importance of task-to-core mapping schemes for parallel applications running on heterogeneous platforms. We assume that two parallel programs *canneal* and *swaptions* are running on a system composed of 2 big cores and 2 small cores. Figure 2(a) graphs the relative performance of both applications while executing on different processors in isolation. The notation 1S indicates that a small core is used to run the program, while 2S, 1B and 2B means using 2 small cores, 1 big core and 2 big cores for the execution, respectively. We launch four threads for each program in all cases. It is straightforward to note the difference between the performance variations of these two programs. For *swaptions*, running it on a big core is around 3X faster than the execution on a small core. Program *canneal*, however, exhibits a completely different scaling trend that moving the application from a small core to a big core results in only 1.19X speedup while giving an extra small core is able to reduce the execution time by ~50%. Let us assume *swaptions* is co-executing with *canneal* on this platform and the former program is assigned a higher priority. A QoS-unaware system might blindly distribute *swaptions* to the small cores and *canneal* to big cores, leading to a result as shown by 'policy 1' in Figure 2(b). By involving a QoS-enabled mechanism (i.e., policy 2), *swaptions* will be assigned to big cores and *canneal* goes to small cores. As can be seen, this significantly boosts the performance of the high priority program at the expense of acceptable performance degradation of *canneal*.

While the motivation of this work is intuitive as justified by Figure 2, designing appropriate QoS policies for parallel execution scenarios requires in-depth understanding on features of typical parallel programs and their interactions with the heterogeneous architecture, which are far from obvious. In this work, we conduct a comprehensive investigation to address this problem. In general, our paper makes the following main contributions:

- To the best of our knowledge, this paper is the first attempt to provide QoS solutions to managing multi-programmed, parallel programs executing on heterogeneous CMP system. By examining the execution behaviors of representative applications, we propose that distinctive task-to-core mapping policies should be applied in different execution scenarios.
- We employ a real heterogeneous hardware to conduct the investigation of QoS management. This leads to more convincing conclusions as it avoids missing important factors that might be overlooked in simulation-based approaches. For example, our hardware-based study is able to completely execute an application while architectural simulations usually concentrate on a specific execution phase of the entire program.
- We propose two categories of task-to-core mapping schemes to meet the QoS goals in a large spectrum of parallel execution circumstances. Employing an appropriate policy significantly improves the performance of the high-priority program while leading to a reasonable balance between all programs.
- We demonstrate that finer-granularity control is important to optimize the application performance on given processor mixture. This includes un-balanced workload distribution and appropriate stage-to-core mapping.

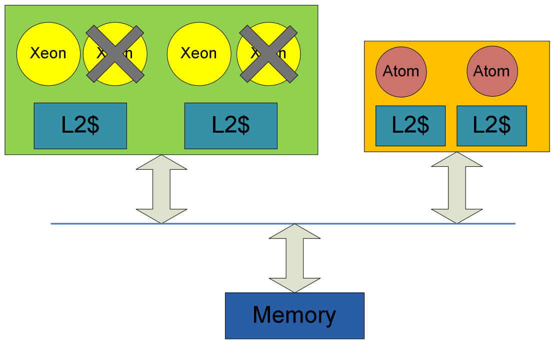


Fig. 3. Architecture of the QuickIA Experimental Hetero System

Table 1. Configuration of the QuickIA system

Parameter	Xeon (Harpertown)	Atom (Silverthorne)
#Cores	2 enabled (of 4)	2
Frequency	1.60G	1.60G
L1 Inst Cache	32KB	32KB
L1 Data Cache	32KB	24KB
L2 cache	1MB/2cores	512KB/core
HyperThreading	N/A	OFF
Pipeline	Out-of-order	In-order
Memory	16GB DDR2	
Operating System	Ubuntu Linux 2.6.32	

II. Background

A. Experimental Hetero Platform

Our evaluation is conducted on a native heterogeneous platform QuickIA [5] developed for the exploration of heterogeneous systems. It is built on the basis of a dual-socket Xeon 5400 series server, where the two CPU sockets are connected to the memory controller via the Intel Front Side Bus (FSB). We illustrate the specific configuration used in this work in Figure 3. As shown in the figure, the system is equipped with a quad-core Xeon CPU (Harpertown) with each pair of cores sharing a 1MB L2 cache and two Atom CPUs (Silverthorne) which reside on another socket. For the purpose of this study, we disable the Intel HyperThreading technique on the Atom CPUs and halt two Xeon cores, making a total of 2 Xeon and 2 Atom processors visible to the operating system. Table 1 lists the architectural parameters of integrated CPUs and other information of the system. In the following sections, we use small cores (S) to indicate the Atom processors and refer to the Xeon processors as big cores (B).

B. Parallel Applications

Parallel applications are extremely important for the exploration of ubiquitous CMP systems in current computer industry. We choose the PARSEC benchmark suite [3] for the purpose of this study. PARSEC is a widely used multi-threaded program set for contemporary chip multi-processor system evaluation. It contains 3 kernels and 10 applications that are derived from a large spectrum of real-world and emerging applications such as data mining, financial analysis, video encoding, recognition, etc.

All PARSEC applications follow a common execution pattern consisting of program initialization, parallel phase, and the completion. The parallel stage is also termed as the Region-of-Interest (ROI) as it contains all parallel executions of an application. Prior studies [3] have shown that PARSEC applications demonstrate a variety of data sharing degree, parallelization models and synchronization patterns, making them compelling tools to assess and steer the design of CMP architecture.

In this study, we simultaneously execute four programs, one of which is elected as the high-priority (HP) application while the remaining three are treated as the low-priority (LP) ones. Note that in later sections of this paper, we use the acronym HP to indicate high-priority application and use the terms LP and low-priority programs interchangeably. Each program is spawned four threads and is fed with the native input for execution. Both the HP and LP applications are executed multiple times and we report the average performance for each program. With such a setup, we mimic the execution scenario when all four applications are contending for system resources.

III. QoS Goals and Policies

A. QoS Goals

A primary goal of our QoS management is to improve performance of the program with the highest priority in shared execution mode. We use the speedup over a predefined baseline case for this program as the evaluation metric. The second goal is to increase the system performance. We employ a widely used metric [6][22][27], weighted speedup, to assess this goal. The third consideration in our QoS management is the fairness among all programs. In the scope of this paper, the fairness achieved by a specific QoS policy is evaluated with the metric unfairness [6]. A smaller unfairness value implies better balance among the involved applications. The following expressions give the calculation of employed metrics.

$$\text{Weight Speedup } W_{speedup} = \sum_{i=0}^{N-1} \frac{Perf_i^{alone}}{Perf_i^{shared}}$$

$$\text{Unfairness } UF = \frac{\max(S_0, S_1 \ldots S_{N-1})}{\min(S_0, S_1 \ldots S_{N-1})}, \text{ where } S_i = \frac{Perf_i^{shared}}{Perf_i^{alone}}$$

In these expressions, N refers to the total number of applications running on the system in concurrency and *Perf* is interpreted by the execution time. Note that in this work, we allow multiple parallel applications to simultaneously execute. Therefore, the notions $Perf^{alone}$ and $Perf^{shared}$ respectively indicate the performance of a program in the dedicated mode and shared mode.

B. QoS Policies

1) Homogeneous-mapping policies: The QoS policies proposed in this work are classified into two categories based on the types of cores assigned to the high priority application. The first group of policies is defined as the homogeneous-mapping policies with which a number of identical cores are reserved for the high priority program. This includes assigning either a group of big cores or multiple small cores to that program.

For many parallel applications, increasing the thread-level-parallelism are more effective for performance improvement compared to exploiting the instruction-level-parallelism. For instance, a thread might generate substantial off-chip traffic and spend fairly long time on waiting for the responses of memory requests. In this situation, running the program on big cores does not necessarily mean a significant reduction of the execution time, because the overall performance tends to depend on the memory subsystem. In other words, if the high-priority application exhibits such behavior, it is preferable to running it with a reasonable number of small cores.

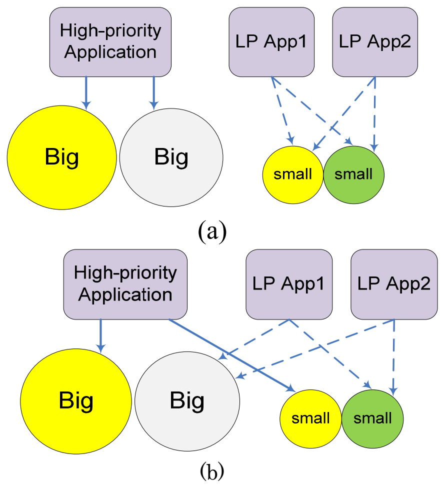


Fig. 4. QoS-aware core mapping strategies: (a) Homogeneous-mapping (big core) (b) Heterogeneous-mapping

Assigning an amount of big cores to the high-priority application is more straightforward to understand since this guarantees superior performance boost for the HP program in most scenarios, satisfying the primary QoS goal of this work. We illustrate such a policy in Figure 4(a). However, this may easily lead to unfairness among programs when low-priority applications manifest large performance degradation on small cores. In order to avoid unacceptable slowdown for LP programs in practical circumstances, it is necessary to introduce heterogeneous-mapping policies that assign programs to hybrid cores.

2) Heterogeneous-mapping policies: Heterogeneous-mapping policies correspond to the schemes which reserve a mixture of cores with diverse computing capability to the high-priority application. The low-priority programs are executed on the remaining available processors. Such strategies are intuitively effective to evade the dilemma that might be encountered in homogeneous-mapping policies. Specifically, if the HP application is granted most big cores, LP programs are thereby confined on the small cores, resulting in unacceptable performance degradation and potential throughput decrease. On the opposite, running the HP application on small cores may fail to reach the desired speedup and thus violates the first QoS requirement. Heterogeneous-mapping policies provide us a solution to effectively utilize the diversity among processors and achieve better balance between the high-priority and low-priority applications. We illustrate a possible core assignment falling into this category in Figure 4(b).

The proposed homogeneous- and heterogeneous-mapping policies both comply with the principle of resource dedication by reserving a set of cores for the HP application. We also propose a partial-dedicated policy which breaks this law by allowing part of the processors to be shared among all programs. More specifically, the high-priority application is executed on a combination of dedicated and shared cores while the low priority programs running on the shared ones and other available cores.

C. QoS policy Evaluation

The specific QoS policies that are evaluated on the QuickIA platform are as follows:

- Big+Big (BB): reserving two big cores to execute the high-priority application. The low-priority applications run on the two Atom cores.
- Small+Small (SS): running the HP application on two small cores and LP applications on all big cores. This policy together with BB belongs to the homogeneous-mapping category.
- Big+Small (BS): assigning a big and a small core to run the HP applications. All low-priority programs contend for the remaining processors. This is a heterogeneous-mapping policy.
- Big+Small+Small (BSS): giving an additional small processor to the HP application on the basis of BS. Low priority applications run the remaining big core. BSS falls to heterogeneous-mapping classification as well.
- All for HP (BBSS_BS): allowing the high-priority application to use all four cores on the platform while the low-priority programs use half of the processors (i.e., a big and a small core, corresponding to the suffix BS). Note that this is a partial-dedicated policy.

The performance of each application under all QoS policies ($Perf_i^{shared}$) is normalized to that when it is running with an Atom core alone ($Perf_i^{alone}$). Note that our QoS evaluation is conducted on an assumption that approximate features of programs which are about to execute are already known. This is fairly reasonable for many real parallel applications such as banking transactions. From this perspective, all proposed policies can be classified into static policies because the task-to-core mapping of a program is permanently set when it is ready to execute. Nevertheless, we believe that our observations on the interactions between mapping strategies and QoS results also hold in other scenarios. Dynamic policies where the core affinitization can be adjusted at runtime are left as our future work.

Note that we apply all these combinations to parallel applications, while for single-threaded multi-programmed applications, we only experiment with reserving one big core or one small core for the high priority application. For domain-specific application, we divide the four cores into two groups and let each application map onto one of them, i.e., each with dedicated cores.

IV. QoS Evaluation

In this section, we present the evaluation results of the QoS policies. We first demonstrate a general picture of the scaling behaviors for selected programs and then analyze the QoS evaluation results in detail. After that, we demonstrate the finer-granularity optimization techniques for different programs on a set of given processors.

A. A General Picture

The scaling behavior of contemporary parallel benchmarks can be found in a large body of prior studies [3][28]. However, most of the conclusions presented in those work are derived from homogeneous platform, while the execution behavior of multi-threaded benchmarks on heterogeneous architectures are not well understood.

We start our analysis by comparing the performance of PARSEC benchmarks between running on the small and big cores. This provides a general picture of characteristics of the program

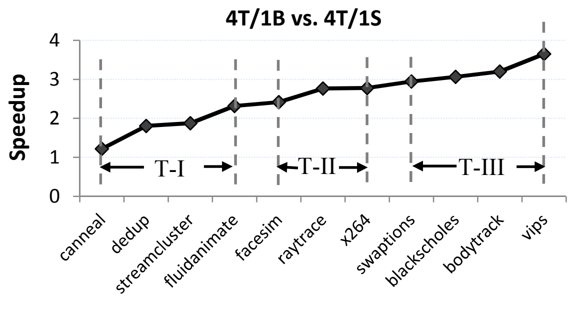


Fig. 5. Relative performance between a big and a small Core

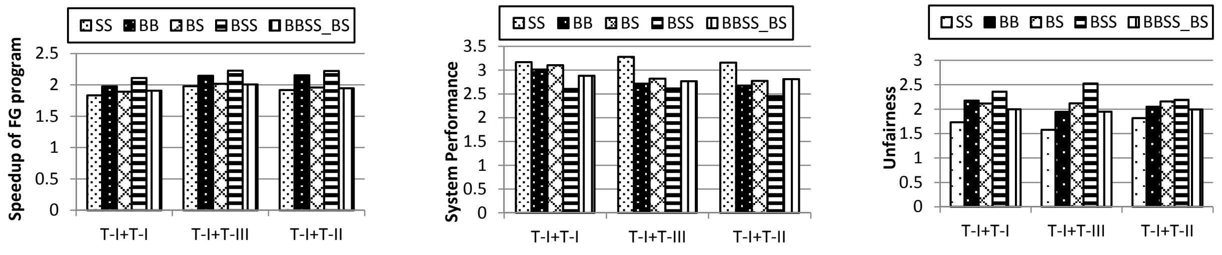
collection, with which we can choose the most suitable QoS policies in different circumstances. Figure 5 shows the speedup of all applications running on a Xeon processor over the execution on an Atom processor. Note that in both cases, each program is launched with 4 threads. As can be observed, the relative performance of these programs between the big and small cores ranges from 1.1X to 3.6X. Applications such as *canneal* which generates a large amount of off-chip memory traffic obtain quite limited performance gain from the sophisticated Xeon processor, since the memory latency is more decisive to its overall performance. On the other hand, programs including *blackscholes* and *bodytrack* contain substantial floating-point operations, thus running them on a Xeon processor where more computation resources are available can significantly improve the performance. According to the relative performance, we approximately classify all programs into three categories as marked in the figure: TypeI-programs (T-I) which demonstrate moderate performance ratio (1.1X ~ 2.3X), TypeIII-programs (T-III) that obtain fairly impressive performance improvement on the big core (>3X), and TypeII-programs (T-II) with relative performance in-between them.

B. Evaluation Results

To perform a comprehensive evaluation of the proposed QoS policies, we should consider as many execution scenarios as possible. In this study, we mimic different circumstances by combining applications with distinctive scaling behaviors and running these combinations on the underlying platform.

Recall that we classify all programs into three categories based on their performance ratios between big and small cores. We select a program from each category to be the high-priority application and co-execute it with LP workloads from different classifications. To give an example, let us assume *canneal* is chosen to be the HP application. Such execution scenarios are referred as HP_T-I because *canneal* is a typical TypeI program as shown in Figure 5. Note that the 4T/1B(1S) in the caption indicates 4 threads on a big (small) core. If the low-priority applications are also T-I programs, the specific combination is denoted by T-I+T-I. Accordingly, T-I+T-II and T-I+T-III indicate the scenarios where the LP applications are positioned in the middle and right segment of the curve in Figure 5, respectively. All of these three situations belong to the HP_T-I category, but implying distinctive execution environments. We define similar notations such as T-II+T-I of HP_T-II and T-III+T-III of HP_T-III. By doing this, we cover most circumstances that might be encountered in practice. For each specific combination like T-I+T-I, we run three different groups of programs and report the average result for this specific case.

1) HP_T-I: We first concentrate on the HP_T-I execution scenarios Figure 6(a) demonstrates the speedup of the high-priority application over the baseline (i.e., the 1S case) when the proposed QoS schemes are applied. As we expect, employing



(a) Speedup of the HP program (b) System Performance (c) Fairness

Fig. 6. Execution behaviors when a TypeI program has high priority (HP_T-I)

more dedicated processors (i.e., BSS) results in higher performance improvement for the high-priority application regardless of the characteristics of other programs running on the system. Take the T-I+T-I combination as an example. The BSS policy delivers 2.1X speedup for the high-priority program while BB and SS respectively increase the performance by 1.9X and 1.8X. We have described the reason in section 3.2 that exploring TLP is more effective to boost the performance of T-I programs. However, the BBSS_BS case is an exception that the performance gain from a quad-core execution (i.e., BBSS) is comparable to that from a dual-core running (i.e., SS/BS), but apparently worse than the situation on three cores (i.e., BSS). This actually justifies the importance of resource dedication when fast execution of the HP application is a primary QoS goal. In the BBSS_BS case, a big core and a small core are shared among all four programs and tend to be persistently busy during the execution, thus the OS scheduler is likely to assign the HP program to its dedicated cores on which only one application is running, in order to achieve a balanced load across the system.

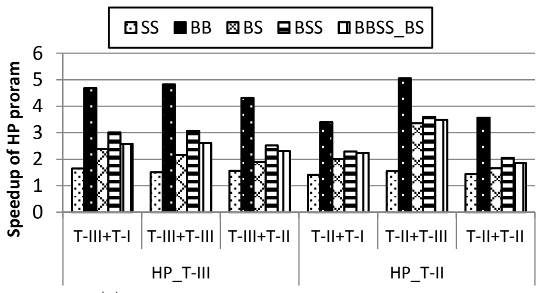
The system performance achieved with each policy is graphed in Figure 6(b). Recall that the performance of each application is normalized to that when it is running on a dedicated Atom core. As can be observed, SS is the optimal among all QoS strategies from the perspective of overall performance. For example, the SS policy delivers a weighted speedup of 3.16 for the T-I+T-III combination while employing BB, BS, BSS and BBSS_BS respectively lead to system performance 2.67, 2.92, 2.45 and 2.81. It is no surprise to see that the BSS scheme tends to largely degrade the system performance since all low-priority applications are confined on a single processor, resulting in slow executions due to severe resource contention. The BS scheme is more interesting in that it leads to comparable performance to SS in the T-I+T-I scenario, but significantly falling behind the same competitor in both T-I+T-III and T-I+T-II contexts. This is caused by the different performance scaling features of those applications. T-II and T-III programs are more sensitive to the core types and achieve much higher execution rates on a big core. As a consequence, decreasing the number of available Xeon cores (i.e., from SS to BS) for LP programs significantly prolong their execution time, thus leading to lower system performance in both T-I+T-II and T-I+T-III scenarios. In contrast, T-I programs have small performance ratio between big and small cores. Therefore, the global performance delivered by SS and BS is fairly close when the LP programs belong to the T-I category.

Figure 6(c) plots the fairness achieved with each mapping policy in the HP_T-I scenario. As can be noted, in all the three categories, the SS scheme leads to remarkably lower unfairness values compared to other policies. This is essentially determined by the slowdown of the LP applications because the high-priority programs (i.e., T-I) are not sensitive to the core type. As described earlier, the performance of individual applications tend to be largely degraded due to severe resource contention. This is exacerbated when all the shared cores are small ones (i.e., in the BB mode). In this configuration, the slowdown of the low-priority applications is quite significant, resulting in unreasonably high unfairness value. On the other hand, by employing the SS scheme, all the powerful big cores are reserved for the LP applications, which is beneficial to improving the performance of the programs running on the shared cores. Therefore, the SS mapping policy results in the most attractive balance among all involved applications. In general, the evaluation results demonstrate that using a number of small cores to run the high-priority application in HP_T-I scenario is the most preferable strategy in a QoS-aware system, because it is capable of effectively accelerating the HP program while resulting in a good tradeoff to low-priority programs.

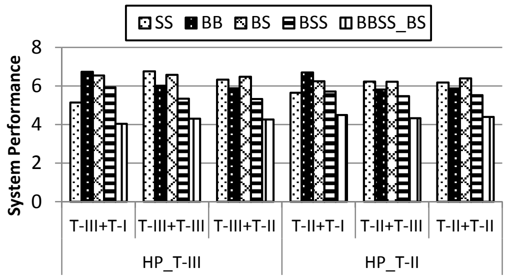
2) HP_T-II and HP_T-III: We now shift our focus to circumstances where a T-III application is assigned higher priority. The speedup of the HP program is shown in Figure 7(a). We observe that the BB policy always delivers the optimal performance for the high-priority application in all evaluated combinations. Specifically, 2 dedicated big cores are able to accelerate the high-priority application by 4.68X, 4.82X, 4.31X respectively for T-III+T-I, T-III+T-III and T-III+T-II over the baseline case. This is fairly reasonable due to the intrinsic characteristics of T-III programs. Heterogeneous-mapping policies (i.e., BS/BSS) outperform the SS strategy by providing intermediate speedup (2X ~ 4X) to the HP program. For the BBSS_BS scheme, it leads to slightly better performance than the BS scheme. This trend is similar to the observation made in Figure 7(a), indicating the significance of dedicated processors for high-priority program.

The system performance is shown in Figure 7(b). We observe that for the T-III+T-I combination, the BB policy outperforms other schemes by delivering the system performance up to 6.74. In T-III+T-III and T-III+T-II scenarios, however, BB trails the SS and BS strategies as it results in relatively lower global performance. For example, the system performance under the BS scheme is around 6.58 while adopting BB leads to a performance not exceeding 6.01 in the T-III+T-III circumstance. This observation justifies our induction described in section 3.2 that reserving many big cores for an individual application (i.e., the high-priority one) is beneficial to boost its performance without heavily degrading the performance of other programs if they are not sensitive to core types; on the contrary, when low-priority programs exhibit large slowdown on small cores, the HP program is virtually accelerated at the expense of significant performance degradation of low-priority applications.

We demonstrate the execution fairness among programs for the HP_T-III scenario in Figure 8. As can be observed from the diagram, heterogeneous-mapping policies briefly lead to more balanced performance across the programs than the homogeneous-mapping schemes because the latter ones tend to cause unreasonable disparity between the execution speed of individual



(a) Speedup of the high-priority program



(b) System Performance

Fig. 7. Execution behaviors when a TypeIII program (HP_T-III) or a TypeII program (HP_T-II) has high priority

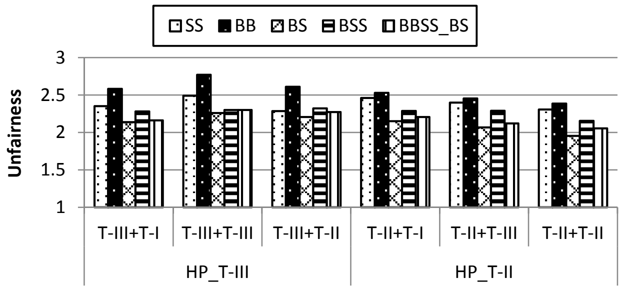


Fig. 8. Execution fairness for HP_T-III and HP_T-II scenarios

programs. For example, when a T-III+T-III application mixture execute with the BB strategy, all LP programs demonstrate significant performance degradation on the shared small cores. Meanwhile, the HP application enjoys impressive performance boost as it runs on dedicated big cores, implying an unfair execution pattern. On the contrary, when both HP and LP programs are assigned a mixture of big and small cores (i.e., heterogeneous-mapping), the execution disparity can be effectively alleviated. In general, by comprehensively evaluating the three QoS goals, it is rational for us to conclude that an appropriate heterogeneous-mapping policy is the most preferable scheme in the HP_T-III circumstance. Note that on our evaluation platform, choosing BS is more suitable than using the BSS strategy, with which only one processor is reserved for the LP programs and a biased execution is encountered as a consequence. We observe similar trends from the results of HP_T-II executions, thus omitting the detailed analysis.

C. Performance Optimization on Core Combinations

For a multi-threaded application, choosing an appropriate parallelization model is one of the most important considerations since it largely determines the program scalability and other execution behaviors. Recall the description listed in section II, the selected PARSEC benchmarks generally fall into two categories with respect to the parallelization model [3], namely data parallel and pipeline. In this situation, understanding the impact of parallelization model on performance variation stands as a key point to further improve program performance and enhance the QoS management at a finer granularity. In this subsection, we present simple yet effective approaches to optimize typical data-parallel and pipeline parallel applications. As we will demonstrate shortly, the proposed techniques are capable of efficiently utilizing assigned processors for heterogeneous-mapping policies.

1) Optimizing data-parallel application: Our first study aims to optimize the performance of data-parallel programs. We choose blackscholes as an example. Blackscholes is an important application in the high performance computing (HPC) domain. It is derived from a financial analysis problem and calculates the

Table 2. Workload distributions for *blackscholes*

Conf.	T0 work (Atom)	T1 work (Atom)	T2 work (Xeon)	T3 work (Xeon)
Balanced (default)	2500000	2500000	2500000	2500000
Conf1	2000000	2000000	3000000	3000000
Conf2	1500000	1500000	3500000	3500000
Conf3	1000000	1000000	4000000	4000000
Conf4	500000	500000	4500000	4500000

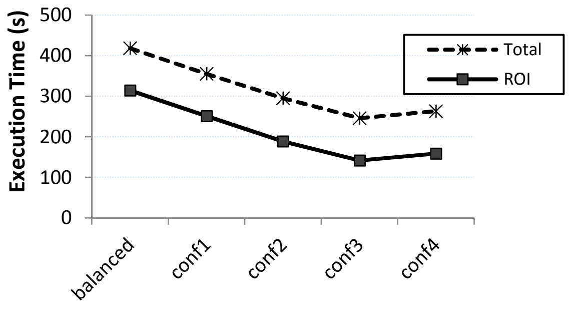


Fig. 9. Performance variation of *blackscholes* with different work load distributions

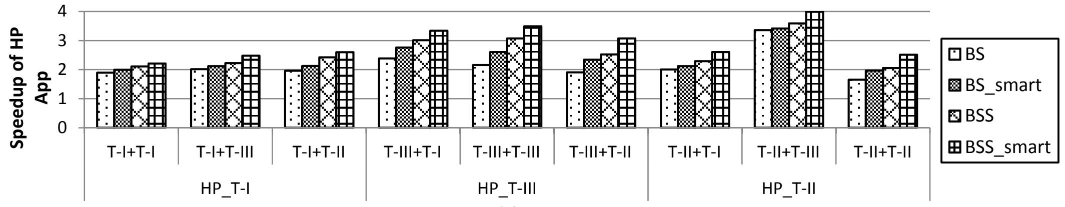
prices for a portfolio with the well-known Black-Scholes partial differential equation (PDE). The portfolio is denoted by a large amount of options which are divided into several work units equal to the number of spawned threads. As a data-parallel application, the process of each thread in blackscholes is completely parallel.

Our investigation starts from demystifying the surprising phenomenon observed from an experiment that using a big and a small core results in even worse performance than engaging an exclusive big core when executing *blackscholes*. To understand the program execution behaviors, we use emon [2] (Intel performance monitoring tool) to record the CPU utilizations. We observe that when *blackscholes* is executing on a big and a small core, both cores enter the parallel phase to process their own threads after the initialization stage. The Xeon processor completes its tasks much faster than the Atom cores; however, the program cannot proceed to the completion stage until the slow threads running on Atom finish the computations. In other words, threads assigned to Atom cores are the bottleneck of the overall performance. By digging into the source code, we find that all options are evenly distributed across worker threads, resulting in much longer execution time on Atom due to its low computation capability.

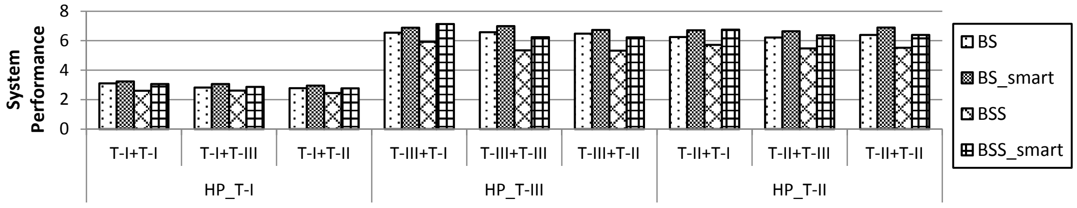
Employing an imbalanced workload distribution policy is a simple solution to increase the utilization of big cores. We thereby modify the default task division and test four different assignments as listed in Table 2. Note that the total number of options is 10000000. Also note that we always affinitize thread 0 and thread 1 on the small core while mapping other two threads to the big core. The variation of the execution time is shown in Figure 9. We

Table 3. Execution information of *dedup* with different stage pinning

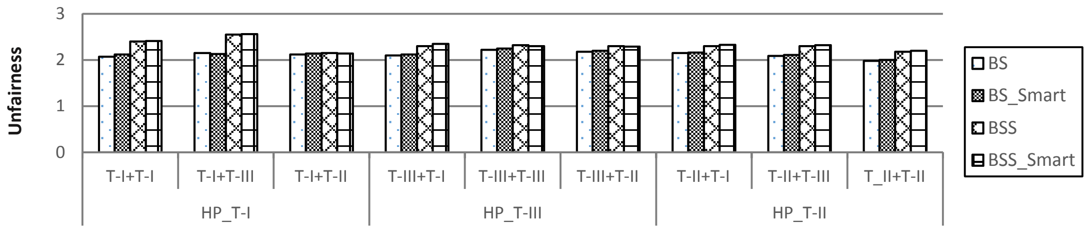
Configuration	**Para. Stage 1**		**Para. Stage 2**		**Para. Stage 3**		**ROI bottleneck**
	Core mapping	Time (s)	Core mapping	Time (s)	Core mapping	Time (s)	
conf.1	big	20	big	32	big	69	Stage 3
conf.2	small	30	big	30	big	65	Stage 3
conf.3	big	19	small	40	big	68	Stage 3
conf.4	big	19	big	28	small	112	Stage 3
conf.5	small	30	small	41	big	62	Stage 3



(a)



(b)



(c)

Fig. 10. Comparison between default policies and smart policies: (a) speedup of HP program (b) system performance (c) execution fairness

plot both the time spent in executing the parallel phase (ROI) as well as the total time. As can be observed, the execution time is decreasing as we gradually increase the work share given to big cores; the best performance is achieved when options are distributed as suggested by configuration 3, where the work share given to big cores is four times of that assigned to small cores.

Data-parallel model is a widely used programming paradigm in contemporary parallel programs. 8 out of 11 applications in the PARSEC program collection adopt this parallelization pattern. Therefore, the described uneven task distribution technique is fairly meaningful for the performance optimization in several execution scenarios. Note that the optimal workload distribution depends on the amount and types of cores given to this program, thus another policy (e.g., BSS) needs a different task assignment from configuration 3 to achieve the optimal performance.

2) Optimizing pipelined application: The second parallelization model is pipeline. With this paradigm, each stage takes as input the outcome of its previous stage, making the entire application proceed as a pipeline. Pipeline model is another important parallelization pattern in contemporary multi-threaded applications since complete data parallelism might be hard to achieve in some applications. In this case, it will be much easier to decouple the entire computation into multiple modules and parallelize each individual module. *Dedup* and *x264* from the PARSEC benchmark suite adopt this model. We choose dedup as an example to illustrate the optimization for pipelined parallel applications. *Dedup* implements a two-level data stream compression algorithm consisting of global compression and local compression. The main computation work is decomposed into five modules, corresponding to five pipeline stages. In particular, the first and the last stage are respectively responsible for breaking up the data and assembling the output stream, while the intermediate three stages perform the actual compression of data chunks. Only the intermediate three stages are parallelized and each stage has its dedicated thread pool. In addition, the number of threads spawned in each stage is identical.

Since each stage performs distinctive job and inclines to cost different time, we employ a stage-to-core mapping approach, which is similar to the scheme used in [28], to understand the execution behaviors. We assume a 1B1S core reservation and test a number of configurations as listed in Table 3, in order to evaluate how the affinity will impact the performance. We list the time spent on each pipeline stage in order to derive the bottleneck of the parallel phase (ROI) for all tested configurations. As can be observed, the third parallel stage remains the ROI bottleneck irrespective of the mapping scheme. In other word, although the execution time of all parallel modules varies across configurations, the third parallel stage always takes the longest time and determines the performance of the entire parallel phase. Due to this reason, the third parallel stage needs to be executed on big cores to achieve the optimal performance if hybrid cores are granted by the QoS policy (e.g., BS or BSS).

3) Putting all together: Putting all of these together, we revisit the heterogeneous-mapping policies and propose BS_smart and BSS_smart policies in which those optimization techniques are applied to the high-priority program. The comparison between the default and smart policies are demonstrated in Figure 10. As can be observed, the smart schemes constantly outperform the default ones by delivering higher speedup for the HP program and better system performance. We also note that introducing the finer-granularity optimization does not influence our selection on QoS policies from the fairness perspective. In specific, the smart heterogeneous-mapping schemes lead to more reasonable fairness value than the default schemes, but still trailing the SS strategy in the HP_T-I scenario. For the HP_T-II and HP_T-III combinations, the BS_smart policy further reduces the unfairness among programs, thus appearing as the most promising scheme. We do not show the figure due to space limitation.

V. Related Work

The quality of service problem has been widely studied on various domains. In recent years, researchers have introduced the QoS problem into the computer architecture area with an concentration on the management of shared resources. Iyer [7] describes a framework to enable QoS in shared caches on CMP platforms. The proposed framework implements QoS enforcement on shared cache via selective cache allocation and dynamic cache partitioning to meet the performance requirement for applications with varying locality properties and memory sensitivities. In [8], the authors further extend the work by proposing a group of specific policies and architectural support to appropriately allocate the shared cache and memory bandwidth, in order to meet QoS goals. Kannan et al. [10] propose a similar mechanism for QoS management in chip multi-processors. Qureshi and Patt [19] develop a utility-based cache partitioning technique to improve the system performance when multiple programs are simultaneously executed. The fairness via resource throttling is elaborated in [6]. The authors highlight the importance of fairness and introduce a native approach to provide fairness in shared memory systems.

There are also a large body of studies discussing thread affinities [16][17][18][23][26]. Klug et al. introduce a technique to determine the optimal thread pinning for an application at runtime based on performance monitoring events information [12]. Radojkovic presents a study on the thread to context binding for parallel network applications in multithreaded systems [20]. The authors demonstrate the impact of thread affinity on the application performance and propose a contention-aware scheduler to facilitate the thread binding problem.

VI. Conclusion

As heterogeneous chip multi-processor gradually becomes an important trend in the next decade and beyond, providing quality of service for programs running on a heterogeneous platform should be carefully considered. While prior QoS studies on traditional homogeneous system mainly concentrate on the management of shared resources including cache and memory bandwidth, task-to-core mapping plays a role while incorporating QoS with heterogeneous CMPs. This is especially important when multiple parallel programs are concurrently running on a system. To address this problem, our paper starts from profiling a wide spectrum of parallel applications on a real heterogeneous prototype, then proposes a series of policies for QoS control via appropriate thread mapping in different scenarios. The evaluation results show that the described policies effectively accelerate high-priority the program while delivering acceptable global throughput and fairness.

Acknowledgement

We acknowledge Professor Seung Eun Lee from Seoul National University of Science and Technology for his invaluable suggestions on the paper finalization. We also appreciate comments from anonymous reviewers which help us improve the overall paper quality.

References

[1] ARM Corporation. Big.LITTLE Processoing. http://www.arm.com/products/processors/technologies/bigLITTLEprocessing.php
[2] EMON. http://emon.intel.com
[3] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and Architectural Implications," Princeton University Technical Report, Jan. 2008.
[4] Black, Fischer, and Scholes, "The pricing of options and corporate liabilities," in Journal of Political Economy, 1973.
[5] B. Chitlur, G. Srinivasa, S. Hahn, et al., "QuickIA: exploring heterogeneous architectures on real prototypes," in HPCA, Feb. 2012.
[6] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt, "Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems," in ASPLOS, Mar. 2010.
[7] R. Iyer, "COoS: a framework for enabling QoS in shared caches of CMP platforms," in ICS, Jul. 2004.
[8] R. Iyer, L. Zhao, F. Guo, R. Illikkal, S. Makineni, et al., "QoS policies and architectures for cache/memory in CMP platforms," in SIGMETRICS'07.
[9] M. K. Jeong, C. Sudanthi, N. Paver, and M. Erez, "A QoS-aware memory controller for dynamically balancing GPU and CPU bandwidth in an MPSoc," in DAC 2012.
[10] H. Kannan, F. Guo, L. Zhao, R. Illikkal, R. Iyer, D. Newell, Y. Solihin, C. Kozyrakis, "From Chaos to QoS: case studies in CMP resource management," in dasCMP, Dec. 2006.
[11] S. Kim, D. Chandra, and Y. Solihin, "Fair cache sharing and partitioning in a chip multiprocessor architecture," in PACT, Sept. 2004.
[12] T. Klug, M. Ott, J. Weidendorfer, and C. Trinitis, "Autopin - automated optimization of thread-to-core pinning on multicore systems," in Transactions on High-Performance Emebedded Architectures and Compilers, 2008.
[13] R. Kumar, D. M. Tullsen, and N. P. Jouppi, "Core architecture optimization for heterogeneous chip multiprocessors," in PACT, Sept. 2006.
[14] K. Luo, J. Gummaraju, M. Franklin, "Balancing throughput and fairness in SMT processors," in ISPASS, 2001.
[15] O. Multu and T. Moscibroda, "Parallelism-aware batch scheduling: enhancing both performance and fairness of shared DRAM systems," in ISCA, Jun. 2007.
[16] S. Parekh, S. Eggers, H. Levy, and J. Lo, "Thread-sensitive scheduling for SMT processors," Technical report, Dept. of Computer Science and Engineering, University of Washington, 2000.
[17] J. Philbin, J. Edler, O. J. Anshus, C. C. Douglas, and K. Li, "Thread scheduling for cache locality," in ASPLOS, 1996.
[18] K .K. Pusukuri, R. Gupta, and L. N. Bhuyan, "Thread reinforcer: dynamically determining number of threads via OS level monitoring," in IISWC 2011.
[19] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: a low-overhead, high-performance, runtime mechanism to partition shared caches," in MICRO, Jun. 2006.
[20] P. Radojkovic, V. Cakarevic, J. Verdu, A. Pajuelo, F. J. Cazorla, M .Nemirovsky, M. Valero, "Thread to strand binding of parallel network applications in massive multi-threaded systems", in PPoPP, 2010.
[21] K. K. Rangan, M. D. Powell, G-Y. Wei, and D. Brooks, "Achieving uniform performance and maximizing throughput in the presence of heterogeneity", in HPCA, Feb. 2011.
[22] A. Snavely and D. M. Tullsen, "Symbiotic job scheduling for a simultaneous multithreading processor," in ASPLOS, Nov. 2000.
[23] S. Sridharan, B. Keck, R. Murphy, S. Chandra, and P. Kogge, "Thread migration to improve synchronization performance," in Workshop on Operating System Interference in High Performance Applications, 2006.
[24] S. Srinivasan, R. Iyer, L. Zhao, and R. Illikkal, "HeteroScouts: hardware assist for OS scheduling in heterogeneous CMPs," in ACM SIGMETRICS Performance Evaluation Review, Jun. 2011.
[25] M .A. Suleman, O. Mutlu, M. K. Qureshi and Y. N. Patt, "Accelerating critical section execution with asymmetric multi-core architectures," in ASPLOS, 2009.
[26] R. Thekkath and S. J. Eggers, "Impact of sharing-based thread placement on multithreaded architectures," in ISCA, 1994.
[27] H. Vandierendonck, and A. Seznec, "Fairness metrics for multi-threaded processors," in IEEE Computer Architecture Letters, 2011.
[28] E. Z. Zhang, Y. Jiang, and X. Shen, "Does cache sharing on modern CMP matter to the performance of contemporary multithreaded programs?" in PPoPP, Jan. 2010.