This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

QoS Management on Heterogeneous Architecture for Multiprogrammed, Parallel, and Domain-Specific Applications

Ying Zhang, Li Zhao, Ramesh Illikkal, Ravi Iyer, Andrew Herdrich, and Lu Peng

***Abstract*—Quality-of-service (QoS) management is widely employed to provide differentiable performance to programs with distinctive priorities on conventional chip-multiprocessor (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. Workloads on heterogeneous architectures can be multiprogrammed, heterogeneous, and/or domain specific depending on the form factor and device of interest. Considering the diverse characteristics of these three classes of workloads is important when managing QoS on heterogeneous architectures. For example, for parallel applications, considering the diverse characteristics of thread synchronization, data sharing, and parallelization pattern of representative parallel applications, 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 multiprogrammed, parallel, and domain-specific 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.**

***Index Terms*—Heterogeneous systems, performance attributes.**

I. Introduction

IN the past decade, multicore processors have become the mainstream to provide high performance while encapsulating the processor power consumption within a reasonable envelope. Most commercial multicore 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 that integrate a diversity of processors onto the same die to better serve applications from different domains.

Manuscript received October 5, 2014; revised March 14, 2015, May 13, 2015, and June 26, 2015; accepted July 26, 2015.

Y. Zhang is with Intel Corporation, Santa Clara, CA 95054 USA (e-mail: ying.e.zhang@intel.com).

L. Zhao, R. Illikkal, R. Iyer, and A. Herdrich are with Intel Labs, Intel Corporation, Hillsboro, OR 97123 USA (e-mail: li.zhao@intel.com; ramesh.g.illikkal@intel.com; ravishankar.iyer@intel.com; andrew.j.herdrich@intel.com).

L. Peng is with the Department of Electrical and Computer Engineering, Louisiana State University, Baton Rouge, LA 70802 USA (e-mail: lpeng@lsu.edu).

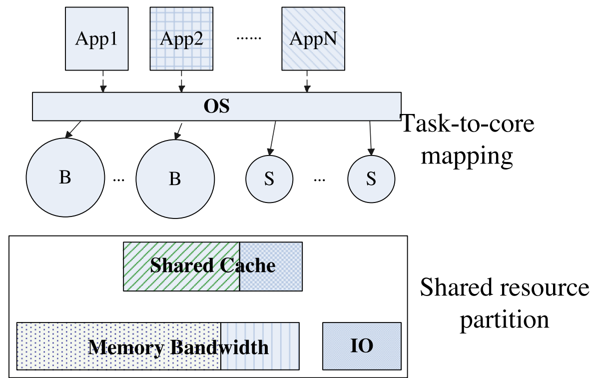
Digital Object Identifier 10.1109/JSYST.2015.2464153

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 corunners and resource management schemes engaged by the system. Fig. 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, multithreaded, 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 [6]–[8], [24] 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.

In this paper, we study three different types of workloads running on such heterogeneous architectures: multiprogrammed single-threaded workloads, parallel workloads, and domain-specific applications. While running multiprogrammed single-threaded workloads on homogeneous multicore platforms are challenging already, things become even more complicated when multiple parallel applications are executed in a heterogeneous CMP system in concurrence. 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 multithreaded 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. Fig. 2 demonstrates 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 two big cores and two small cores. Fig. 2(a) graphs the relative performance of both applications while executing on different processors in isolation. The notation 1S indicates

1932-8184 © 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.



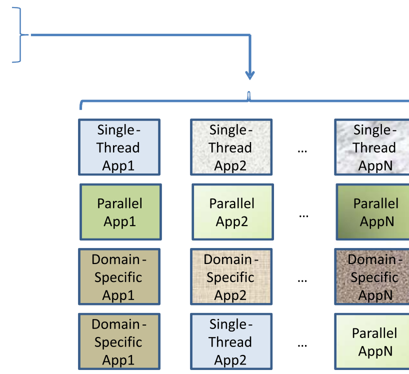


Fig. 1. QoS-aware heterogeneous CMP system. Different types of simultaneous workloads.

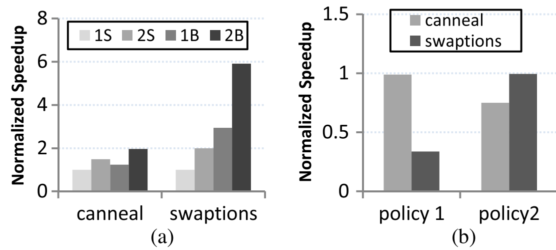


Fig. 2. Illustrating the need for QoS. (a) Performance scaling with core numbers and types. (b) Performance comparison with different policies (policy 1: *canneal* on 2 big cores and *swaptions* on 2 small cores. policy2: *canneal* on 2 small cores and *swaptions* on 2 big cores).

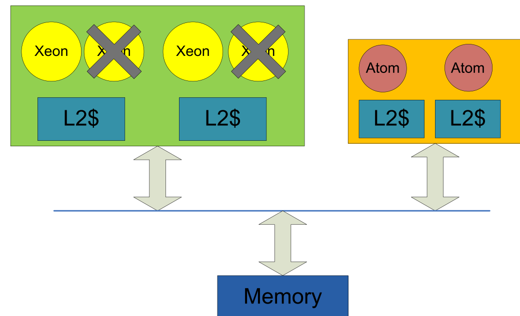


Fig. 3. Architecture of the QuickIA Experimental Heterosystem.

that a small core is used to run the program, whereas 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 three times 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 1.19× speedup while giving an extra small core is able to reduce the execution time by ∼50%. Let us assume *swaptions* is coexecuting 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 Fig. 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 HP program at the expense of acceptable performance degradation of *canneal*.

In this paper, we aim at addressing the QoS problem on heterogeneous platforms and make the following main contributions.

1) To the best of our knowledge, this paper is the first attempt to provide QoS solutions to managing multiprogrammed, parallel and domain-specific 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.
2) 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, whereas architectural simulations usually concentrate on a specific execution phase of the entire program.
3) 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 HP program while leading to a reasonable balance between all programs.
4) We demonstrate that finer-granularity control is important to optimize the application performance on given processor mixture. This includes unbalanced workload distribution and appropriate stage-to-core mapping.

II. Background and Motivation

A. Experimental Heteroplatform

Our evaluation is conducted on a native heterogeneous platform QuickIA [4] 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 paper in Fig. 3. As shown in the figure, the system is equipped with a quad-core Xeon CPU (Harpertown) with each pair of cores sharing a 1-MB 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

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

TABLE I
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
Pipeline	Out-of-order	In-order
Memory	16GB DDR2	
Operating System	Ubuntu Linux 2.6.32	

TABLE II
DESCRIPTION OF PARSEC BENCHMARKS

Application	Description	parallel model
blackscholes	Financial engineering	data-parallel
bodytrack	Body tracking	data-parallel
canneal	Simulation annealing	unstructured
dedup	Compression	pipeline
facesim	Face simulation	data-parallel
fluidanimate	Fluid dynamics	data-parallel
raytrace	Raytracing	data-parallel
streamcluster	Data clustering	data-parallel
Swaptions	Portfolio pricing	data-parallel
Vips	Image processing	data-parallel
X264	Video encoding	pipeline

CPUs and halt two Xeon cores, making a total of 2 Xeon and 2 Atom processors visible to the operating system. Table I 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. Parallel Applications on Heteroarchitectures

Parallel applications are extremely important for the exploration of ubiquitous CMP systems in the current computer industry. We choose the PARSEC benchmark suite [2] for the purpose of this study. A brief description of the applications is listed in Table II. PARSEC is a widely used multithreaded program set for contemporary chip-multiprocessor 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 [2] have shown that PARSEC applications demonstrate a variety of data sharing degrees, parallelization models and synchronization patterns, making them compelling tools to assess and steer the design of CMP architecture.

C. Multiprogrammed Applications on Heteroarchitectures

In addition to parallel applications, we also look study the effects of multiprogrammed single-threaded applications running simultaneously on heterogeneous multicore architectures.

TABLE III
DESCRIPTION OF SPEC BENCHMARKS

Category	Application
Computation-intensive	dealII, h264ref, calculix, tonto, sphinx3, games
Memory-intensive	lbm, mcf, milc, soplex, omnetpp, astar

TABLE IV
DESCRIPTION OF DOMAIN-SPECIFIC BENCHMARKS

Application	Category	Description
sphinx3	Single-thread	Speech recognition
bodytrack	Multi-thread	Computer vision

For this study, we picked groups of SPEC2006 applications to run simultaneously. Table III shows the set of SPEC2006 applications chosen for this paper We chose two groups based on the compute-intensive and memory-intensive behavior of the applications.

D. Domain-Specific Applications on Heteroarchitectures

Apart from the extensively studied general-purpose applications, investigating the performance and QoS challenges specifically for domain-specific applications running on heterogeneous multicore architectures is also fairly important. In this paper, we chose two domain-specific applications, respectively from speech recognition and computer vision to conduct the study, in order to characterize and understand the execution behaviors when multiple domain-specific applications are running on a heterogeneous architecture in concurrence. Specifically, *sphinx3* from SPEC2006, a single-threaded program implementing a speech recognition algorithm, and *bodytrack* from PARSEC, which stands as a representative parallel computer vision application, are selected for the investigation. The goal is to understand whether it is better to run such applications on small or big cores since these workloads tend to have soft or hard real-time performance needs. A side effect is also to study the benefits of using big core as a domain-specific accelerator for small cores. A summary of the selected applications are listed in Table IV.

E. Workload Construction

For PARSEC benchmarks, we simultaneously execute four programs, one of which is elected as the high-priority (HP) application, whereas 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 HP application and use the terms LP and LP 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. In addition, with such a setup, we mimic the execution scenario when all four applications are contending for system resources. For multiprogrammed applications, we run eight programs simultaneously. Similar to parallel applications, we choose one of them as the HP application and the rest are treated as LP ones. For domain-specific application, we run the two benchmarks together and choose either one as a HP application.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

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, weighted speedup, to assess this goal. Note that, in this paper, we use the term system throughput and system performance interchangeably. The third consideration in our QoS management is the fairness among all programs. The execution fairness can be quantified in different manners [5], [26], [29]. In the scope of this paper, we adopt the metric unfairness defined in [5] for the evaluation since it is widely used in computer architecture studies concentrating on multiprogram scenarios. A smaller unfairness value implies better balance among the involved applications. The following expressions give the calculation of employed metrics:

$$\text{Speedup of HP application Speedup} = \frac{Perf^{\text{QoS}}}{Perf^{\text{baseline}}}$$

$$\text{Weighted Speedup } W_{\text{speedup}} = \sum_{i=0}^{N-1} \frac{Perf_i^{\text{QoS}}}{Perf_i^{\text{baseline}}}$$

$$\text{Unfairness } UF = \frac{\max(S_0, S_1, \ldots, S_{N-1})}{\min(S_0, S_1, \ldots, S_{N-1})}$$

$$\text{where } S_i = \frac{Perf_i^{\text{QoS}}}{Perf_i^{\text{baseline}}}.$$

In these expressions, N refers to the total number of applications running on the system in concurrence and $Perf$ is interpreted by the execution time. Note that, in this paper, we allow multiple parallel applications to simultaneously execute. The notions $Perf^{\text{baseline}}$ and $Perf^{\text{QoS}}$, respectively indicate the performance of a program under a baseline configuration without QoS mechanism and that with a QoS policy involved. In addition, $Perf^{\text{baseline}}$ is measured as the performance of an application executing on a dedicated small core.

B. QoS Policies

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

Assigning an amount of big cores to the HP 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 paper. We illustrate such a policy in Fig. 4(a). However, this may easily lead to unfairness among programs when LP 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.

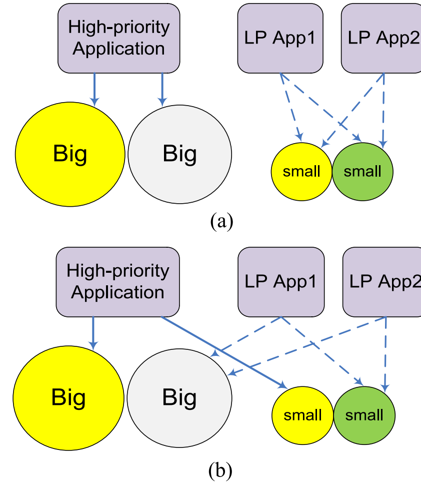


Fig. 4. QoS-aware core-mapping strategies. (a) Homogeneous mapping (big core). (b) Heterogeneous mapping.

2) Heterogeneous-Mapping Policies: Heterogeneous-mapping policies correspond to the schemes that reserve a mixture of cores with diverse computing capability to the HP application. The LP 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 high 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 HP and LP applications. We illustrate a possible core assignment falling into this category in Fig. 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 that breaks this law by allowing part of the processors to be shared among all programs. More specifically, the HP application is executed on a combination of dedicated and shared cores, whereas the LP programs running on the shared ones and other available cores.

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

1. **Big + Big (BB)**: reserving two big cores to execute the HP application. The LP applications run on the two Atom cores.
2. **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.
3. **Big + Small (BS)**: assigning a big and a small core to run the HP applications. All LP programs contend for the remaining processors. This is a heterogeneous-mapping policy.
4. **Big + Small + Small (BSS)**: giving an additional small processor to the HP application on the basis of BS. LP applications run the remaining big core. BSS falls to heterogeneous-mapping classification as well.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

5. **All for HP (BBSS_BS)**: allowing the HP application to use all four cores on the platform, whereas the background 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.
6. **Memory bandwidth (Mem_BW)**: This is a policy derived from an existing QoS management strategy for multisocket systems where significant contention on memory subsystem is present [24]. We extend this approach and apply it to our heterogeneous CMP platform for comparison. Recall the platform description given in Section II. Since there is no last-level cache shared between the big and small processors, the main resource under contention on QuickIA is the off-chip memory bandwidth. Therefore, we define the following QoS policy. We monitor the off-chip memory access requests initiated by each program during the execution. At the end of an interval, we compare the number of memory accesses (i.e., off-chip bandwidth requirement) of all programs and migrate the most memory-intensive one to the Atom processor, aiming to slowing down its memory request issue rate. By doing so, we avoid a single program dominating the off-chip bandwidth; thus, other applications also get a fair share of the resource under contention. Note that this approach is designed to optimize the system throughput. In addition, note that the interval length is set to 5 ms, which is in the range of the typical Linux scheduler time slice [25]. We consider this QoS policy as one of the representative and optimal stat-of-art QoS management schemes and compare it with our proposed strategies.

The performance of each application under all QoS policies ($Perf_i^{\mathrm{QoS}}$) is normalized to that when it is running with an Atom core alone ($Perf_i^{\mathrm{baseline}}$). 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, whereas for single-threaded multiprogrammed applications, we only experiment with reserving one big core or one small core for the HP 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

A. Parsec Application

1) General Picture: 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 collection, with which we can choose the most suitable QoS policies in different circumstances. Fig. 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 $1.1\times$ to $3.6\times$. Applications such as *canneal*, which generate a large amount of off-chip memory traffic obtain quite limited performance gain from the Xeon processor. On the other hand, programs including *blackscholes* and *bodytrack* contain substantial floating-point operations, thus running them on a Xeon processor can significantly improve the performance. According to the relative performance, we approximately classify all programs into three categories as marked in the figure: Type-I programs (T-I), which demonstrate moderate performance ratio ($1.1\times$–$2.3\times$), Type-III programs (T-III), which obtain fairly impressive performance improvement on the big core ($>3\times$), and Type-II programs (T-II) with relative performance in-between them.

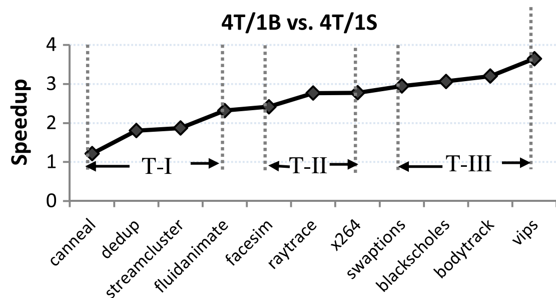


Fig. 5. Relative performance between a big and a small Core.

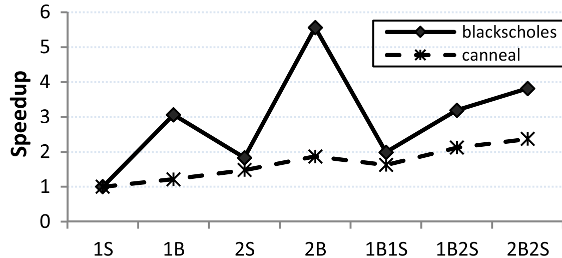


Fig. 6. Two performance scaling trends. (1) *anneal* favoring more cores (2) *blackscholes* favoring big cores.

Aside from the sensitivity to core types, the performance scaling with varying number of cores is another important feature for parallel applications. We compare the execution time of all applications while running on the following core combinations: 1S, 1B, 2S, 2B, 1B1S, 1B2S, and 2B2S. The notation 1S indicates that a small core is used to execute the benchmark. Similar explanations apply to other configurations. Note that these processor mixtures are selected in compliance with two principles: 1) the total number of available cores is gradually increased; and 2) both heterogeneous and homogeneous configurations are considered.

Fig. 6 demonstrates the performance variation of two representative programs when executed on the designated core combinations. For program *blackscholes*, the performance heavily fluctuates when the underlying hardware is changed. However, it is straightforward to observe that including big cores is more decisive to the performance improvement compared with increasing the number of cores. This indicates that for programs exhibiting similar behavior as *blackscholes*, the inclusion of big cores is a key factor to boost the performance. On the contrary, *canneal* is more sensitive to the number of cores used for execution. As can be seen, its execution time keeps decreasing

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

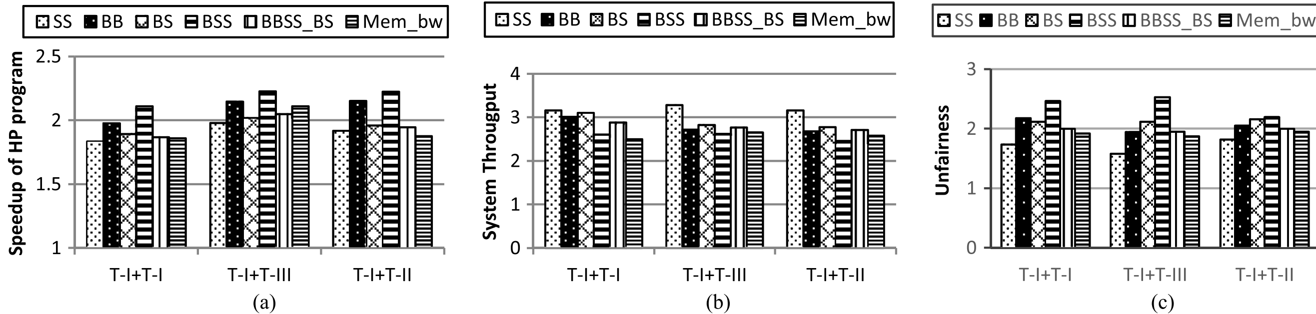


Fig. 7. Execution behaviors when a Type-I program has HP (HP_T-I). (a) Speedup of the HP program. (b) System performance. (c) Fairness.

as extra processors is granted irrespective of the core types. In general, it is rational to conclude that using more cores is more effective to increase the performance for applications such as *canneal*. This essentially matches the information conveyed by Fig. 5 that T-III programs favor big processors for faster execution, whereas T-I programs might prefer more cores for higher TLP.

2) Evaluation Results: To perform a comprehensive evaluation of the proposed QoS policies, we should consider as many execution scenarios as possible. In this paper, 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 HP application and coexecute 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 Type-I program, as shown in Fig. 5. 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 Fig. 5, respectively. All of these three situations belong to the HP_T-I category, but implying distinctive execution environments. By doing this, we cover most circumstances that might be encountered in practice.

HP_T-I: We first concentrate on the HP_T-I execution scenarios. Fig. 7(a) demonstrates the speedup of the HP application over the baseline (i.e., the 1S case) when different QoS schemes are applied. As we expect, employing more dedicated processors (i.e., BSS) results in higher performance improvement for the HP 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 2.1× speedup for the HP program, whereas BB and SS, respectively increase the performance by 1.9× and 1.8×. We have described the reason in Section III-B that exploring TLP is more effective to boost the performance of T-I programs. However, the BBSS_BS case is an exception since the performance gain from the execution on four cores (i.e., BBSS) is similar to that from a dual-core running (i.e., SS/BS), but apparently worse than the situation on three cores (i.e., BSS). This 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 speedup of the HP application delivered by the Mem_bw scheme is also in the range of 1.7×–2.1×. This is close to the performance when other competing QoS schemes are employed. The reason is that a T-I application is relatively less sensitive to the underlying processor type; thus, a core migration is unlikely to introduce noticeable performance variation. However, an important issue that deserves to notice is the impact of migration overhead. As we will demonstrate shortly, the overhead can result in significant performance degradation in particular contexts.

The system performance achieved with each policy is graphed in Fig. 7(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 LP 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. Now, let us shift our concentration to the Mem_bw scheme. As demonstrated in the graph, this scheme is not showing notable benefit on boosting system throughput compared with our proposed policies. This is surprising since the Mem_bw strategy is designed from the throughput optimization perspective. A key reason of the discrepancy is the migration overhead introduced by this approach. This particularly impacts the performance of memory-intensive applications (e.g., in the T-I + T-I scenario) because there are a large amount of cache rewarming after migration. As a consequence, the overall throughput benefit is largely mitigated.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

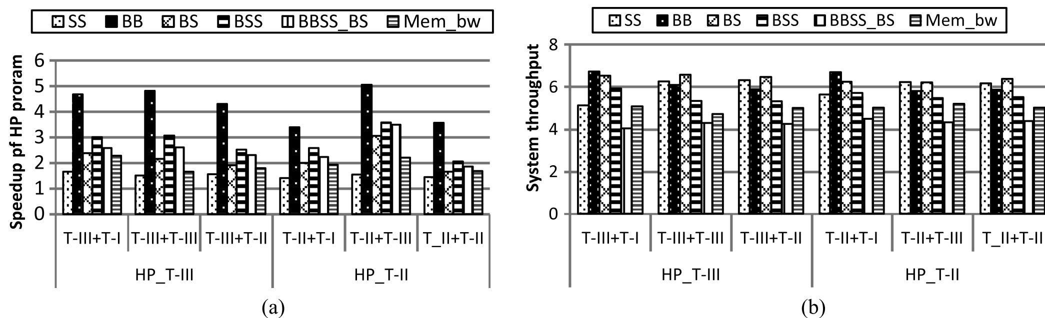


Fig. 8. Execution behaviors when a Type-III program (HP_T-III) or a Type-II program (HP_T-II) has HP. (a) Speedup of the HP program. (b) System performance.

Fig. 7(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 with other policies. This is essentially determined by the slowdown of the LP applications because the HP 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 LP 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. The fairness associated with the Mem_bw scheme is fairly close to the SS scheme. This is because the HP T-I application is likely to be frequently scheduled on the Atom processor due to its high memory bandwidth requirement. In general, the evaluation results demonstrate that using a number of small cores to run the HP 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 LP programs.

HP_T-III and HP_T-II: 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 Fig. 8(a). We observe that the BB policy always delivers the optimal performance for the HP application in all evaluated combinations. Specifically, 2 dedicated big cores are able to accelerate the HP application by 4.68×, 4.82×, 4.31×, 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 (2×– 4×) to the HP program. For the BBSS_BS scheme, it leads to a slightly better performance than the BS scheme. This trend is similar to the observation made in Fig. 7(a), indicating the significance of dedicated processors for HP programs. On the other aspect, the speedup of the HP application obtained from the Mem_bw scheme largely depends on the behavior of its corunners. Let us focus on the HP_T-III scenario. As can be observed from the figure, the acceleration is acceptable when the LP jobs are from the T-I category (i.e., T-III+T-I). This is reasonable because that a T-I application usually initiates a large amount of off-chip memory requests and is likely to be assigned to the Atom processor, leaving the HP job to execute on the Xeon core and benefit from its powerful computing capability. Note that in this scenario, the performance improvement in Mem_bw still largely trails the BB scheme. This is mainly due to the resource contention on the big core because it is not a dedicated resource. On the contrary, when all applications demonstrate similar memory behavior (T-III+T-III/T-II), the HP job is highly likely to be scheduled on the small core if its off-chip memory accesses are slightly higher than its corunners. In this case, its performance will be close to that delivered by the SS scheme.

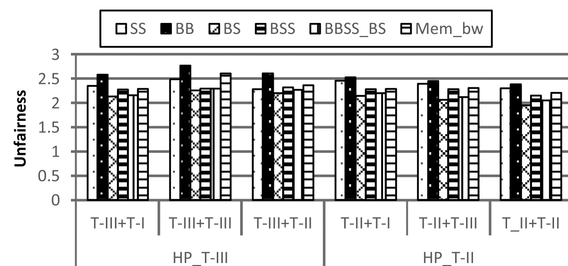


Fig. 9. Execution fairness for HP_T-III and HP_T-II scenarios.

The system performance is shown in Fig. 8(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 III-B that reserving many big cores for an individual application (i.e., the HP 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 LP programs exhibit large slowdown on small cores, the HP program is virtually accelerated at the expense of significant performance degradation of LP applications.

We demonstrate the execution fairness among programs for the HP_T-III scenario in Fig. 9. 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 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

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

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, the execution disparity can be effectively alleviated. Similar to the trend shown in Fig. 7(a), the execution fairness resulted from the Mem_bw scheme is sensitive to the behaviors of the corunners. For example, in the T-III+T-III scenario, we observe large bias among the performance of different applications. Since a program may execute on the Atom processor for relatively long time, its performance is largely degraded and leads to unbalanced execution rate. In general, by comprehensively evaluating these 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.

3) Performance Optimization on Core Combinations: For a multithreaded 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 Table I, the selected PARSEC benchmarks generally fall into two categories with respect to the parallelization model [2], 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. Here, we present simple yet effective approaches to optimize typical data-parallel and pipeline parallel applications. As we will demonstrate shortly, these techniques are capable of efficiently utilizing assigned processors for heterogeneous-mapping policies.

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 prices for a portfolio with the well-known Black-Scholes partial differential equation (PDE) [3]. 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 in Fig. 6 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 (Intel performance monitoring tool) [1] to record the CPU utilizations. Fig. 10 graphs the utilizations when *blackscholes* is executing on a big and a small core. As can be seen, 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 [27] is a simple solution to increase the utilization of big cores.

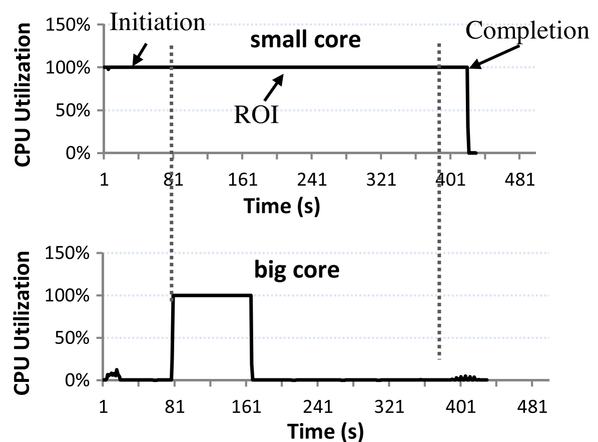


Fig. 10. CPU utilizations of *blackscholes* running with 1B1S.

TABLE V
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

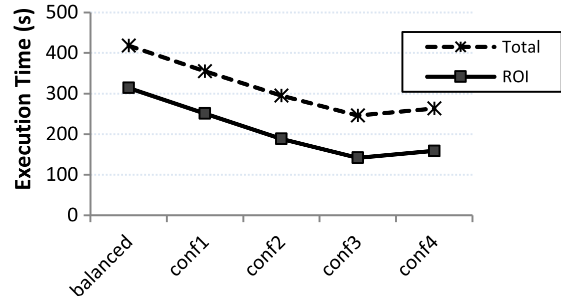


Fig. 11. Performance variation of *blackscholes* with different work load distributions.

We thereby modify the default task division and test four different assignments, as listed in Table V. Note that the total number of options is 10 000 000. In addition, 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 Fig. 11. We 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.

Optimizing pipeline 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 multithreaded applications since complete data parallelism might be hard to achieve in

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

TABLE VI
EXECUTION INFORMATION OF DEDUP WITH DIFFERENT STAGE PINNING

Configuration	Para. Stage 1		Para. Stage 2		Para. Stage 3		ROI bottle-neck
	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

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, whereas 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 [27], to understand the execution behaviors. We assume a 1B1S core reservation and test a number of configurations, as listed in Table VI, 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).

The impact of the sequential phase: While the parallel execution phase encompasses the most important and interesting parts of a multithreaded application, the program serial portion also plays a role in deciding its overall performance. As indicated by Amdahl's law, the maximal performance gain of a program running on a multiprocessor system over a uniprocessor system is essentially determined by the length of its serial portion. This implies that the time spent on the sequential part should be minimized. Note that in the scope of this paper, the serial portion is specialized to program initiation and finalization. With this kept in mind, we shall execute the program stages that are before and after the parallel phase on a big core.

Putting all together: Putting all of these together, we rerun *blackscholes* and *dedup* with a big and a small core. We compare the resultant performance to that of previous executions. As can be seen from Fig. 12, using a big and a small core with finer-granularity control is obviously faster than the execution on a single big core and nonoptimal 1B1S combination for both programs. Specifically, the optimized configuration (1B1S_opt) accelerates *blackscholes* and *dedup* by 20% and 9.8% over the single big core execution (1B), respectively. In addition, we revisit the heterogeneous-mapping policies and propose BS_smart and BSS_smart policies, in which those optimization techniques are applied to the HP program. The comparison between the default and smart policies are demonstrated in Fig. 13. As can be observed, the smart heterogeneous-mapping 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. In addition, we include the results associated with the Mem_bw scheme in the graph as well to make the comparison results more comprehensive. Not-surprisingly, our enhanced policy outperforms Mem_bw on all QoS goals in study.

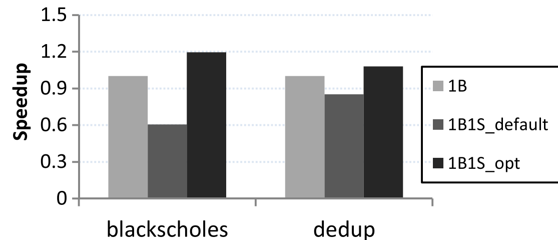


Fig. 12. Performance comparison between default execution and execution with finer-granularity control.

B. Multiprogrammed Application

As described in the previous section, we use SPEC2006 workload for multiprogrammed applications. Fig. 14 shows the performance speedup of running on Xeon core versus on Atom core. We can see that all these workloads will benefit from big core, with various speedup from 1.5× up to 4.5×. In general, most memory intensive workloads such as *milc* and *astar* get less performance speedup than those computation intensive workloads such as games and *zeusmp*. When multiple applications are running simultaneously on the heterogeneous architecture, it is uncertain which application will get the big core and which one gets the small core. In addition, even with small core, it could be shared between two applications if the number of applications is more than the number of cores available. To showcase the benefit of QoS policy, we select seven computation

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

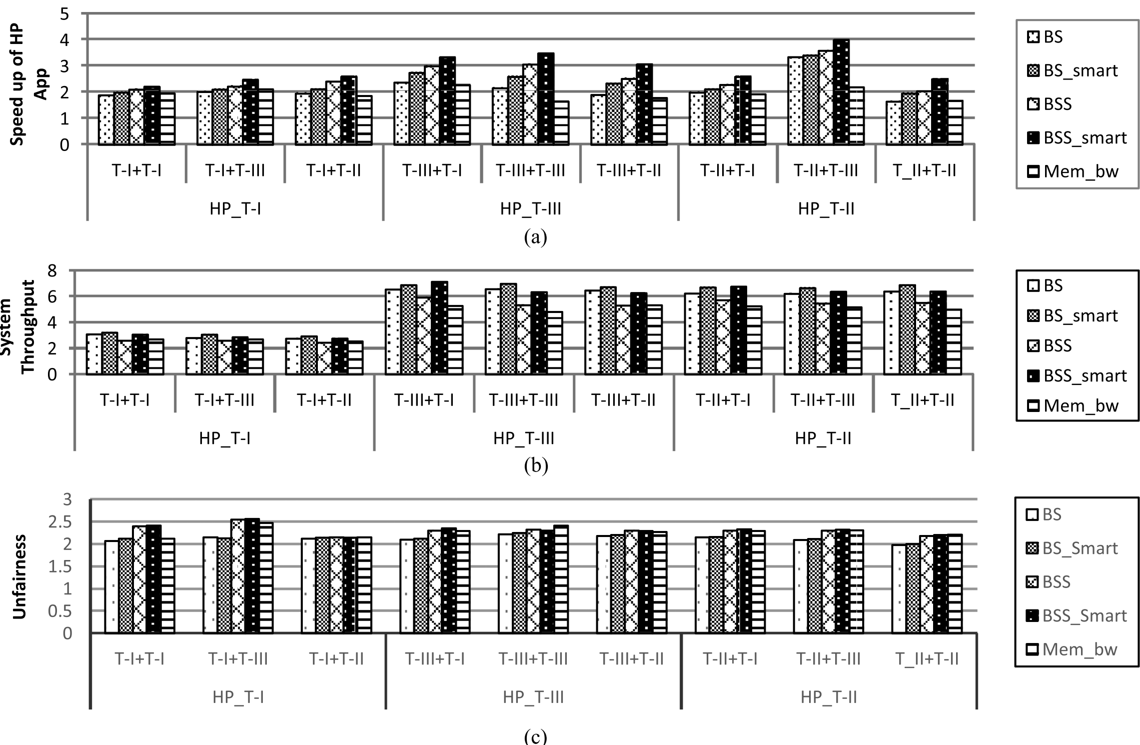


Fig. 13. Comparison between default policies and smart policies. (a) Speedup of HP program. (b) System performance. (c) Execution fairness.

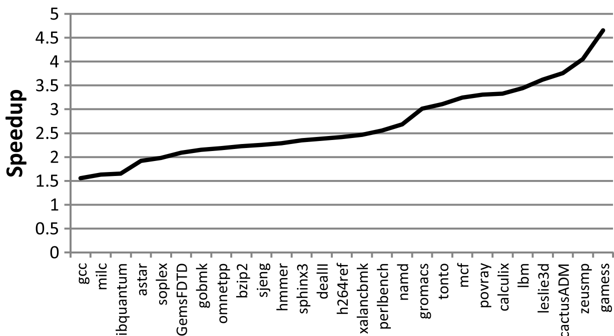


Fig. 14. Speedup of SPEC2006 running on big core versus on small core.

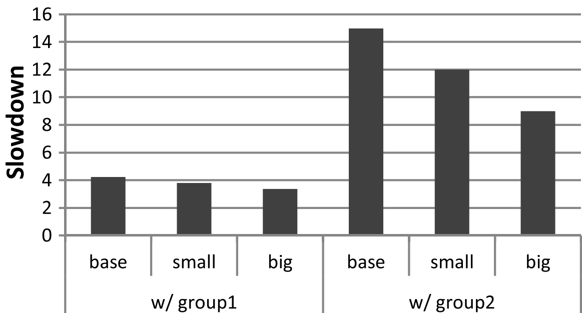


Fig. 15. Slowdown of *sphinx3* running with group 1 and group 2 compared with running alone

intensive workloads as group1 and a mix of three computation and four memory intensive workloads as group 2. We choose *sphinx3* to run together with these two groups, respectively and show its performance without QoS policy and with QoS policy. As shown in Fig. 15, the base case is when *sphinx3* shares all four cores with the other seven workloads. The second and third case indicated by "small" and "big" is when sphinx3 is treated

Fig. 16. Speedup of *bodytrack* compared with running on one small core with various configurations.

as HP and run on the dedicated small and big core, respectively. The y-axis shows the slowdown compared with the case where it is running alone without competition with other applications. We notice that sphinx3 suffers from fierce contention, whereas it is coexecuting with other applications in a QoS-unaware environment, leading to significant performance degradation. Specifically, when running with computation intensive workloads (i.e., group 1), its slowdown is more than four times; whereas the competitor is a mixture of computation-intensive and memory-intensive applications (i.e., group 2), however, the execution time of sphinx3 is prolonged by up to 15X due to the resource contention. In this situation, involving an appropriate QoS policy is capable of providing impressive benefit for the HP application and reducing its performance loss.

C. Domain-Specific Application

We choose *sphinx3* from SPEC2006 and *bodytrack* from PARSEC for concurrent execution, in order to mimic the workload for perceptual computing domain. Fig. 16 shows

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

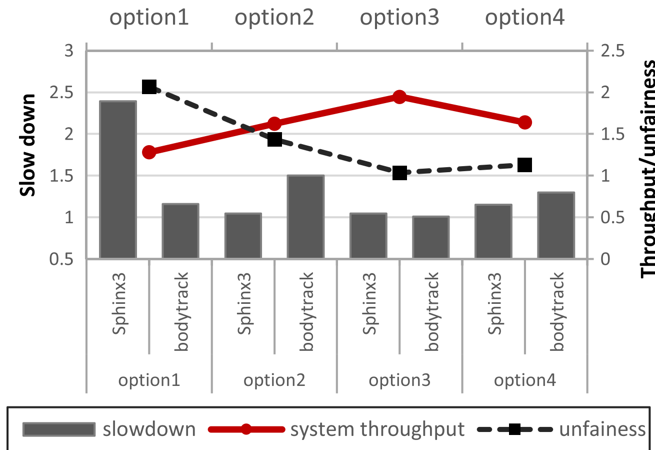


Fig. 17. Execution results of running two domain-specific applications simultaneously with four QoS policies.

the speedup of running *bodytrack* on various configurations compared with when it is running on one small core. As can be observed, using more cores is able to improve its performance; moreover, big cores are able to deliver more significant benefit compared with the small cores. For example, running *bodytrack* on two small cores doubles the performance, but it is not as good as running on one big core. Essentially this indicates that *bodytrack* prefers big cores. Fig. 17 shows the result of running these two applications simultaneously, whereas four different QoS policies are employed. Option1 is to run *sphinx3* on a small core; alternatively, option2 is to run *sphinx3* on a big core. Option 3 refers to our proposed heterogeneous-mapping policy with finer-granularity optimization for *bodytrack*. In this particular scenario, we decide to assign *sphinx3* on the big core based on the result of option 2. Therefore, option 3 is essentially BSS_smart as described in Section IV-A. Finally, with option 4, the Mem_bw scheme is utilized. In all cases, we let *bodytrack* use the rest cores without sharing any core with *sphinx3*. The y-axis on the left shows the slowdown compared with the optimal execution configuration: running on big core for *sphinx3* and running on all four cores for *bodytrack*. The y-axis on the right shows the system throughput and execution unfairness. The figure clearly shows the tradeoff between the QoS goals: the variation trend of unfairness is approximately opposite to that of the system throughput, and option 3 delivers the highest throughput because it results in the most balanced execution. This is consistent with our observation made in Section IV. Note that with option 3, the performance of *bodytrack* is fairly close to its performance with four cores, although it is assigned three cores by the policy. This is not surprising due to the finer-granularity optimization.

D. Summary

Fig. 18 shows the average speedup of SPEC2006/PARSEC/Domain-Specific workloads considered in this paper running on one big core versus on one small core. Compared × SPEC2006 programs, PARSEC benchmark gain relatively less benefit from the big core, but by nature they will benefit from more cores. For the domain-specific application, both application benefit from the big core at almost the same level. This indicates that

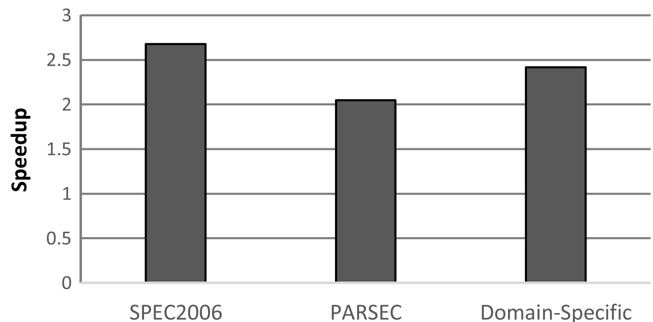


Fig. 18. Average speedup of SPEC2006/PARSEC/Domain-Specific benchmarks running on one big core versus on one small core.

appropriate QoS policies need to be carefully applied to meet the user requirement.

V. Related Work

In recent years, researchers have introduced the QoS problem into the computer architecture area with special concentration on the management of shared resources. Iyer [6] 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 [7], the authors further extend the work by proposing a group of specific policies and architectural sup-port to appropriately allocate the shared cache and memory bandwidth, in order to meet preset QoS goals on CMP systems. Kannan *et al.* [8] propose a similar mechanism for QoS management in CMPs. Luo *et al.* [12] focus on balancing the execution throughput and fairness on SMT processors, and propose the fetch gating to achieve the desired goal. Qureshi and Patt [17] 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 [5]. Mutlu [13] propose a technique for parallelism-aware batch scheduling by designing a shared DRAM controller which provides QoS for threads.

There are also studies discussing thread affinities. Klug *et al.* introduce a technique to determine the optimal thread pinning for an application at runtime based on performance monitoring events information [10]. The optimization of critical section execution on heterogeneous system is studied by Suleman *et al.* [21]. Poovey and others propose dynamic thread mapping on asymmetric CMPs based on parallelization patterns [15].

Our work deviates from the aforementioned studies in that we first comprehensively characterize the execution behaviors of different types of parallel applications running on a real heterogeneous platform, and then propose a set of simple yet effective QoS polices which exploit the heterogeneity across cores. In addition, compared with our previous work [28], this study extends the scope of the application to multiprogrammed and domain-specific workloads, thus making the evaluation more comprehensive. By identifying the most appropriate stage-to-core mapping, the proposed schemes are able to provide expected performance for concurrent-running applications with different priorities.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

VI. Conclusion

As heterogeneous CMP gradually become an important trend in the next decade and beyond, providing QoS 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 particularly important when multiple single-threaded and parallel programs are simultaneously running on the system. To address this problem, our paper starts from profiling a wide spectrum of parallel applications and typical domain-specific programs 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.

References

[1] EMON. [Online]. Available: http://emon.intel.com

[2] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," Princeton Univ., Princeton Township, NJ, USA, Princeton Univ. Tech. Rep. TR-811-08, 2008.

[3] F. Black and M. Scholes, "The pricing of options and corporate liabilities," *J. Political Econ.*, vol. 81, no. 3, pp. 637–654, 1973.

[4] N. Chitlur *et al.*, "QuickIA: Exploring heterogeneous architectures on real prototypes," in *Proc. HPCA*, New Orleans, LA, USA, 2008, pp. 1–8.

[5] 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 *Proc. ASPLOS*, Pittsburgh, PA, USA, 2010, pp. 335–346.

[6] R. Iyer, "CQoS: A framework for enabling QoS in shared caches of CMP platforms," in *Proc. ICS*, Saint-Malo, France, pp. 257–266.

[7] R. Iyer *et al.*, "QoS policies and architecture for cache/memory in CMP platforms," in *Proc. SIGMETRICS*, San Diego, CA, USA, pp. 25–36.

[8] H. Kannan *et al.*, "From chaos to QoS: Case studies in CMP resource management," in *Proc. dasCMP*, Orlando, FL, USA, 2006, pp. 1–10.

[9] S. Kim, D. Chandra, and Y. Solihin, "Fair cache sharing and partitioning in a chip multiprocessor architecture," in *Proc. PACT*, Antibes Juan-les-Pins, France, 2004, pp. 111–112.

[10] 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 Embedded Architectures and Compilers*. Berlin, Germany: Springer-Verlag, 2011.

[11] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen, "Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction," in *Proc. MICRO*, San Diego, CA, USA, 2003, p. 81.

[12] K. Luo, J. Gummaraju, and M. Franklin, "Balancing throughput and fairness in SMT processors," in *Proc. ISPASS*, Tucson, AZ, USA, 2001, pp. 164–171.

[13] O. Mutlu and T. Moscibroda, "Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems," in *Proc. ISCA*, Jun. 2008, pp. 63–74.

[14] J. Philbin, J. Edler, O. J. Anshus, C. C. Douglas, and K. Li, "Thread scheduling for cache locality," in *Proc. ASPLOS*, Cambridge, MA, USA, 1996, pp. 60–71.

[15] J. A. Poovey, M. C. Rosier, and T. M. Conte, "Pattern-aware dynamic thread mapping mechanisms for asymmetric manycore architectures," Georgia Inst. Technol., Atlanta, GA, USA, Tech. Rep. 2011-1, 2011.

[16] K. K. Pusukuri, R. Gupta, and L. N. Bhuyan, "Thread reinforcer: Dynamically determining number of threads via OS level monitoring," in *Proc. IISWC*, Austin, TX, USA, 2011, pp. 116–125.

[17] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *Proc. MICRO*, Orlando, FL, USA, 2006, pp. 423–432.

[18] J. C. Saez, A. Fedorova, D. Koufaty, and M. Prieto, "Leveraging core specialization via OS scheduling to improve performance on asymmetric multicore systems," *ACM Trans. Comput. Syst.*, vol. 30, no. 2, p. 6, Apr. 2012.

[19] A. Snavely and D. M. Tullsen, "Symbiotic job scheduling for a simultaneous multithreading processor," in *Proc. ASPLOS*, Cambridge, MA, USA, 2000, pp. 234–244.

[20] S. Sridharan, B. Keck, R. Murphy, S. Chandra, and P. Kogge, "Thread migration to improve synchronization performance," in *Proc. OSIHPA*, 2006, pp. 1–8.

[21] G. Edward Suhm, L. Rudolph, and S. Devadas, "Dynamic cache partitioning for simultaneous multithreading systems," in *Proc. PDCS*, Los Angeles, CA,USA, 2001, pp. 116–127.

[22] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt, "Accelerating critical section execution with asymmetric multi-core architectures," in *Proc. ASPLOS*, Washington, DC, USA, 2009, pp. 253–264.

[23] M. A. Suleman, M. K. Qureshi, Y. Khubaib, and N. Patt, "Feedback-directed pipeline parallelism," in *Proc. PACT*, Vienna, Austria, 2010, pp. 147–156.

[24] L. Tang, J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa, "The impact of memory subsystems resource sharing on datacenter applications," in *Proc. ISCA*, San Jose, CA, USA, 2011, pp. 283–294.

[25] L. A. Torrey, J. Coleman, and B. P. Miller, Comparing Interactive Scheduling in Linux. [Online]. Available: http://pages.cs.wisc.edu/~ltorrey/papers/torrey_spe06.pdf

[26] H. Vandierendonck, and A. Seznec, "Fairness metrics for multi-threaded processors," *Comput. Architect. Lett.*, vol. 10, no. 1, pp. 4–7, Jan.–Jun. 2011.

[27] E. Z. Zhang, Y. Jiang, and X. Shen, "Does cache sharing on modern CMP matter to the performance of contemporary multithreaded programs?" in *Proc. PPoPP*, Bangalore, India, 2010, pp. 203–212.

[28] Y. Zhang *et al.*, "QoS management on heterogeneous architecture for parallel applications," in *Proc. IEEE ICCD*, 2014, pp. 332–339.

[29] H. Zhao and R. Sakellariou, "Scheduling multiple DAGs onto heterogeneous systems," in *Proc. IPDPS*, 2006, pp. 1–14.

Authors' photographs and biographies not available at the time of publication.