Document downloaded from:

http://hdl.handle.net/10251/71918

This paper must be cited as:

Feliu Pérez, J.; Sahuquillo Borrás, J.; Petit Martí, SV.; Duato Marín, JF. (2013). L1-Bandwidth Aware Thread Allocation in Multicore SMT Processors. IEEE. doi:10.1109/PACT.2013.6618810.

The final publication is available at

http://dx.doi.org/10.1109/PACT.2013.6618810

Copyright IEEE

Additional Information

© 2013 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

L1-Bandwidth Aware Thread Allocation in Multicore SMT Processors

Josué Feliu, Julio Sahuquillo, Salvador Petit, and José Duato
Department of Computer Engineering (DISCA)
Universitat Politècnica de València
València, Spain
Email: jofepre@fiv.upv.es, {jsahuqui,spetit,jduato}@disca.upv.es

***Abstract*—Improving the utilization of shared resources is a key issue to increase performance in SMT processors. Recent work has focused on resource sharing policies to enhance the processor performance, but their proposals mainly concentrate on novel hardware mechanisms that adapt to the dynamic resource requirements of the running threads.**

This work addresses the L1 cache bandwidth problem in SMT processors experimentally on real hardware. Unlike previous work, this paper concentrates on thread allocation, by selecting the proper pair of co-runners to be launched to the same core. The relation between L1 bandwidth requirements of each benchmark and its performance (IPC) is analyzed. We found that for individual benchmarks, performance is strongly connected to L1 bandwidth consumption, and this observation remains valid when several co-runners are launched to the same SMT core.

Based on these findings we propose two L1 bandwidth aware thread to core (t2c) allocation policies, namely Static and Dynamic t2c allocation, respectively. The aim of these policies is to properly balance L1 bandwidth requirements of the running threads among the processor cores. Experiments on a Xeon E5645 processor show that the proposed policies significantly improve the performance of the Linux OS kernel regardless the number of cores considered.

***Keywords*—SMT, thread allocation, bandwidth-aware scheduling**

I. Introduction

Simultaneous multithreading (SMT) processors exploit both instruction-level and thread-level parallelism by issuing instructions from different threads in the same cycle. Thread-level parallelism increases the chance of having instructions ready to be issued so reducing the vertical waste at the issue logic [1]. Because of instructions from different threads can be launched each cycle, threads are continuously sharing processor resources. This means that performance of SMT cores strongly depends on how resources are shared among threads.

The subset of processor resources that are shared depends on the actual SMT implementation but it typically includes, among others, functional and arithmetic units, instruction queues, renaming registers and first-level caches. If at a given time of the execution, the demand for a given resource exceeds what that resource can provide, the processor performance can be damaged. Thus, smart thread to core (t2c) mapping policies can help alleviate the contention in shared resources in current multicore multithreaded processors. As opposite, a *naive* policy could even create a new bottleneck by increasing the contention for a given resource.

A critical shared resource in any current multicore system is the memory bandwidth. Main memory bandwidth is shared by all the processor cores. For a given system, the higher the number of cores the higher the main memory bandwidth contention. Climbing the memory hierarchy, LLC caches (and caches of higher levels) are also typically shared by a subset or all the cores; thus, bandwidth contention can rise at different points of the memory hierarchy. Main memory [2] [3] and LLC bandwidth [4] [5] [6] have been addressed in many recent research works that have shown the potential performance improvements that bandwidth-aware scheduling policies can offer by providing a better sharing of the memory hierarchy resources.

In summary, research works focusing on SMT processors have concentrated on enhancing the utilization of shared resources in the core, while research works focusing on CMPs have proposed scheduling strategies to avoid bandwidth contention in main memory and shared caches. However, to the best of our knowledge, L1 bandwidth, which is private to SMT cores in CMP systems but shared to threads in the core, has not been addressed yet neither in scheduling nor resource sharing strategies.

This paper has two main contributions. First, we explore the connection between the L1 bandwidth and performance of processes. The experiments performed in a real system show that the performance of a given process is strongly connected with its L1 bandwidth consumption while the amount of L1 bandwidth each thread consumes depends on the bandwidth requirements of the threads running concurrently on the same core (known as co-runners). Rises and drops in the L1 bandwidth utilization of a thread have a direct impact on its IPC, and affect in an opposite way the IPC of the co-runner, since the amount of available L1 bandwidth changes. Therefore, it is expected that the more balanced the L1 bandwidth is, the highest the performance.

To leverage this finding, we propose two thread allocation strategies, namely Static (St2c) and Dynamic (Dt2c), with the goal of balancing L1 bandwidth requirements of the running threads among the processor cores. The first policy uses the average L1 bandwidth requirements of the threads measured in standalone execution to perform the thread to core (from now on t2c) mapping. In contrast, the Dynamic policy uses performance counters to update the L1 bandwidth requirements

of the threads dynamically at runtime to adapt the t2c mapping to the bandwidth requirements that threads exhibit at each point of time.

Experimental results on a Xeon E5645 processor show that the proposed policies can significantly improve the performance over Linux OS scheduler. The Dynamic policy offers the best performance under the evaluated workloads, achieving performance improvements up to 6.54% over Linux OS scheduler. Moreover, the policy can be combined with memory bandwidth-aware schedulers or resource sharing strategies to provide farther performance benefits.

The rest of this paper is organized as follows. Section II discusses related work. Section III describes the platform where the experiments have been carried out. Section IV analyses the relationship between L1 bandwidth and performance of processes when running alone and with a co-runner. Section V proposes novel thread allocation policies designed to improve performance by enhancing L1 bandwidth distribution. Section VI explains the evaluation methodology. Section VII evaluates the performance of the proposals. Finally, Section VIII presents some concluding remarks.

II. Related Work

A large amount of research work has analyzed the impact of resource sharing in modern multicores as well as scheduling and thread allocation strategies to exploit resource sharing while avoiding negative effects on performance.

Some preliminary works [2] [3] on this topic focused on main memory bandwidth contention. Antonopoulos et al. [2] proposed several scheduling policies trying to match the total bandwidth requirements of the running processes to the peak memory bus bandwidth. In [3], Xu et al. proved that contention can rise even when memory bandwidth requirements are below the peak bandwidth due to irregular access patterns. They proposed to distribute the memory accesses over the execution time of a workload to minimize contention by means of scheduling strategies.

Regarding LLC contention, Tang et al. [4] studied the impact of sharing memory resources on datacenter applications and found that improperly sharing LLC resources can cause potential degradation. To tackle this issue, authors presented two approaches that enhance the t2c assignments in the datacenter. In [5], Zhuravlev et al. proposed a scheduling algorithm that among other resources, addresses contention for LLC space. Knauerhase et al. [7] presented a scheduler that observes task execution properties using hardware counters to provide co-schedules that reduce cache interference. Fedorova et al. [8] proposed a *cache-fair* scheduling algorithm that gives more execution time to those processes whose performance is affected by unequal cache sharing. Most of these works base the co-schedules in measuring the number of cache misses during sampling periods. In contrast, Jiang et al. [9] presented cache-contention aware proactive scheduling (CAPS), which assigns processes according to cache reuse signatures, avoiding some of the constraints of sampling-based techniques.

More recent scheduling strategies take into account several levels of the memory hierarchy. Feliu et al. [6] addressed bandwidth contention along the memory hierarchy of chip multiprocessors (CMP), while Eyerman and Eeckhout [10] count the number of misses along the cache hierarchy in a simultaneous multithreading (SMT) processor and use the number of misses in each cache to estimate job symbiosis in a probabilistic way.

The current predominant approach to processor design combines multicore and multithreading in a single chip. In this type of processors, thread allocation plays a key role in improving overall performance due to the multiple and heterogeneous levels of resource sharing. In [11], Čakarević et al. characterized different types of resource sharing in a UltraSPARC T2 processor and presented a case study where they improve the execution of a multithreaded network application with a resource sharing aware scheduler. Acosta et al. [12] showed that processor throughput has a high dependence on thread allocation and proposed a t2c allocation policy that, in general, combines computation-bound and memory-bound processes in each core. A similar strategy is followed by Weng and Liu in [13].

This work is related with resource partitioning in CMPs with SMT cores. There is also a significant amount of research exploring this issue, but no especial attention has been devoted to L1 bandwidth utilization. Some SMT resource partitioning policies like DCRA [14] account L1 misses in order to classify threads as *slow* or *fast*, while recent techniques like Hill-Climbing [15] or ARPA [16] concentrate on partitioning internal pipeline resources by taking into account performance feedback and core utilization metrics, respectively. Regarding cache partitioning, the works by Moreto et al. [17] [18] focused on partitioning the LLC of CMPs to increase memory level parallelism and reduce workload imbalance. On the other hand, recently proposed cache partitioning algorithms SHARP [19] and PriSM [20] manage LLC cache sharing in CMPs using formal control and probability theories, respectively. Finally, Chen and John [21] coordinate pipeline and L2 management to optimize performance in CMPs. This problem is also tackled by Bitirgen [22] et al. using artificial neural networks.

III. Experimental Platform

Experiments have been performed in a shared-memory SMT Intel Xeon E5645 processor, with six dual-thread cores. Each core includes two levels of private caches, a 32KB L1 and a 256KB L2. A third-level cache of 12 MB is shared among the L2 private caches. The system is equipped with 12 GB of DDR3 RAM and runs at 2.4 GHz.

The system has installed a Fedora Core 10 Linux distribution with kernel 3.3.0. The library *libpfm* 4.3.0 is used to manage hardware performance counters [23]. Events *perf_count_hw_cache_l1d:access* and *perf_count_hw_cache_l1d:miss* are used to gather the L1 requests, while events *unhalted_core_cycles* and *instructions_retired* are used to collect executed cycles and instructions, respectively. The events are gathered at runtime to provide online values for L1 bandwidth and IPC during the execution of benchmarks. In addition, the proposed Static and Dynamic thread allocation policies are based on runtime L1 bandwidth measures obtained from performance counter values.

SPEC CPU2006 benchmark suite with reference inputs has been used in all the experiments. For evaluation purposes, the execution time of the benchmarks is fixed to 200 seconds in standalone execution. Benchmarks with shorter or longer execution time are relaunched or killed, respectively, to run exactly during 200 seconds. The number of executed instructions required by each benchmark to achieve this execution time is recorded offline and used as target number of instructions for further executions.

IV. Effects of L1 bandwidth on performance of SMT processors

Current microprocessors deploy a cache hierarchy organized in two or three levels of caches. The first-level cache, the closest to the processor, is the most frequently accessed while low level caches, are accessed in case the looked data is not found in the higher level caches. Consequently, L1 caches are critical for performance and thus, they are designed to provide fast access and high bandwidth.

This section analyzes the relation between L1 bandwidth consumption and processor performance (i.e., IPC). First, the dynamic behavior in stand-alone execution is analyzed. Then, we study how co-runners (i.e., two threads running simultaneously in the same core) interact each other on their respective performances and L1 bandwidth consumptions.

A. Stand-alone execution

As a first step to investigate the possible relation between the bandwidth utilization of the L1 cache and the overall processor performance, we measured the average L1 transaction rate (i.e., TR_{L1}) and the IPC achieved by each process. To avoid interferences of other applications each benchmark was run alone.

Figure 1(a) and Figure 1(b) depict both average TR_{L1} and IPC of the SPEC CPU2006 benchmarks. At first glance, a certain correlation can be observed between both metrics since most benchmarks with high IPC also present high TR_{L1}, and conversely, benchmarks with low IPC also experience low TR_{L1}. However, benchmarks with similar IPCs can widely differ in their L1 transaction rates (e.g., *gobmk* and *hmmer*), and vice versa, benchmarks with close TR_{L1} can diverge in the achieved IPC (e.g., *dealII* and *GemsFDTD*). Thus, although certain similarities appear among both performance indicators, there is no clear evidence about the connection between them.

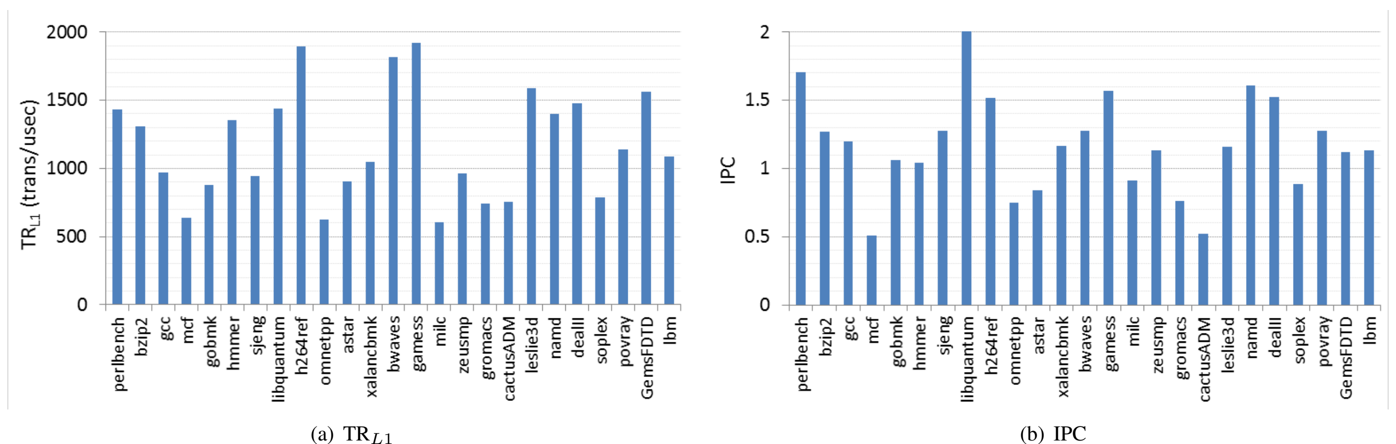
Nevertheless, it is well known that the benchmark behavior can widely vary over the execution time. Thus although some divergences can appear on the average values, one should look for further insights in the dynamic values of both metrics at run-time.

Figure 2 depicts the results at each OS execution quantum for a subset of benchmarks. Each plot presents the IPC and L1 bandwidth for the same benchmark to ease the analysis. In addition, both Y axis (IPC and TR_{L1}) are scaled in a 100x factor. The plots clearly illustrate the strong connection between both metrics. As observed, L1 bandwidth utilization and performance show an almost identical shape during the entire execution time for all the benchmarks. Both metrics follow the same trend (rises and drops) in a synchronized way and similar magnitude. The trend in both performance indicators is so close that even small peaks can be observed in both IPC and TR_{L1} curves (e.g., by time equal to 40 seconds in Figure 2(f)).

The finding that both IPC and TR_{L1} for a process follow a so synchronized and correlated trend has important connotations. It implies that when a process shows high performance (i.e., high IPC) during a running period, it will certainly show high L1 bandwidth consumption. And vice versa, if a process is consuming a small amount of L1 bandwidth then its IPC is expected to be low. Therefore, to allow processes to achieve their best performance they must be run so that they can get the highest bandwidth consumption; thus, these scenarios should be promoted. Since some benchmarks present phases with widely differenced L1 bandwidth requirements, changes in the t2c allocation should be allowed dynamically at run-time to favor such scenarios.

B. Analyzing interferences between co-runners

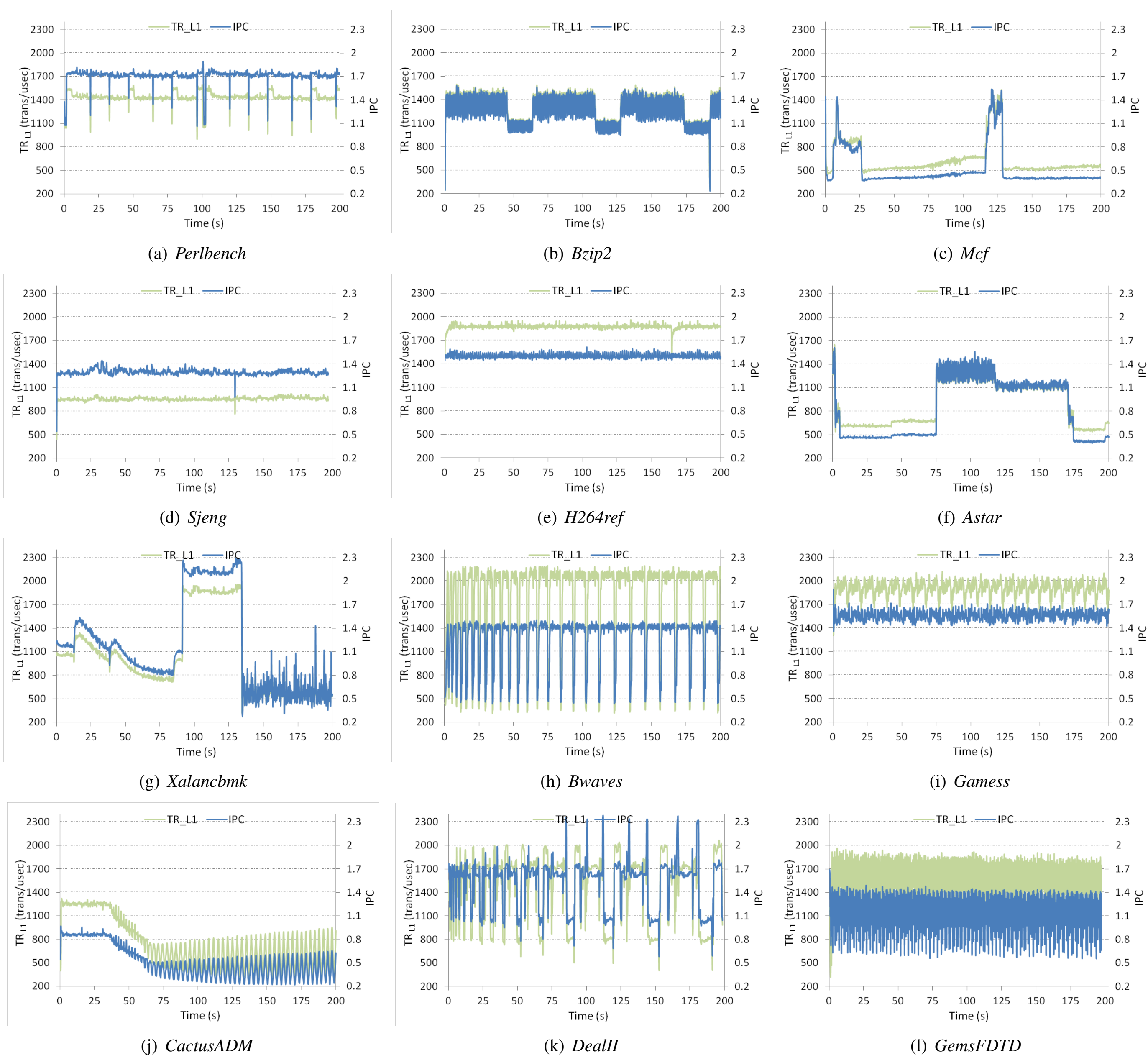
While current microprocessors implement LLC caches, which are shared by a subset or all cores, L1 caches are designed private to each core. In case of single-threaded cores, all available L1 bandwidth is devoted to a single process. In such a system, processes do not compete for L1 bandwidth.



(a) TR_{L1}

(b) IPC

Figure 1. Average TR_{L1} and IPC for SPEC CPU 2006 benchmarks



(a) *Perlbench* (b) *Bzip2* (c) *Mcf*

(d) *Sjeng* (e) *H264ref* (f) *Astar*

(g) *Xalancbmk* (h) *Bwaves* (i) *Gamess*

(j) *CactusADM* (k) *DealII* (l) *GemsFDTD*

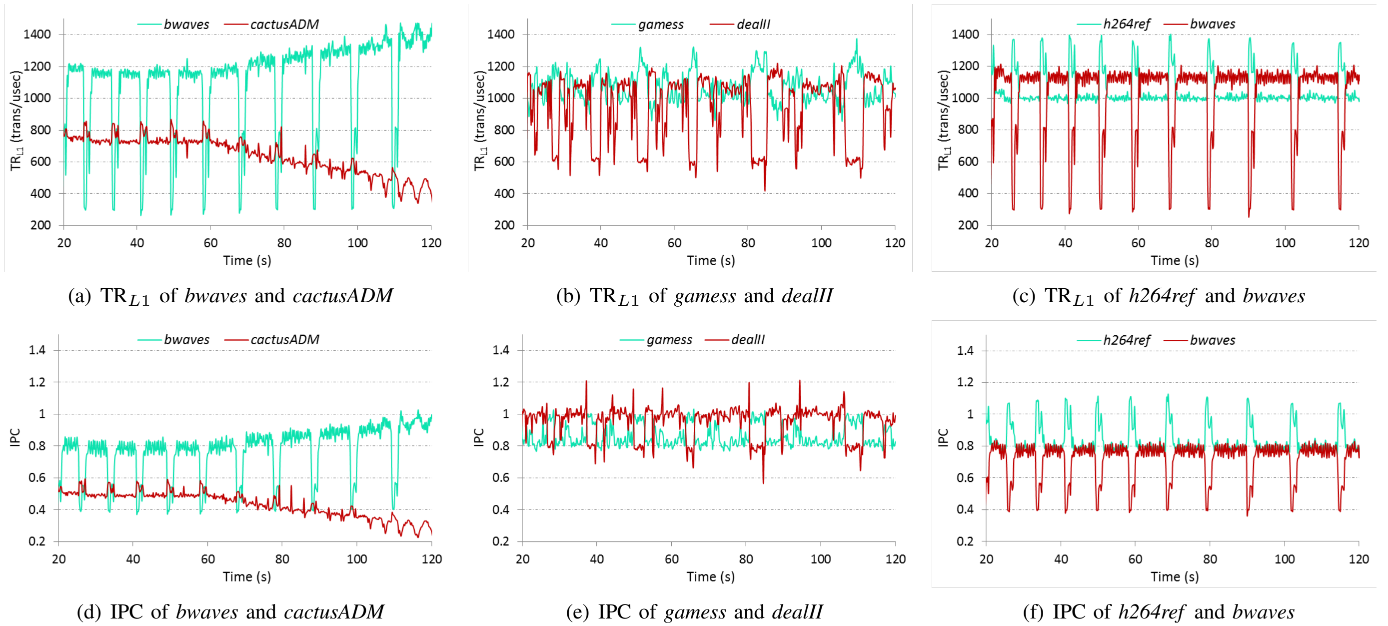
Figure 2. TR$_{L1}$ and IPC evolution with time for a set of SPEC CPU 2006 benchmarks

In contrast, in current SMT cores, those threads running concurrently share the L1 cache. Since, as shown above, the performance of the processes depends on the L1 bandwidth they utilize, the performance will suffer when several threads run in the same SMT core because of they compete for L1 bandwidth.

This section analyzes how sharing the L1 bandwidth can limit the thread performance. To this end, multiple experiments running two different benchmarks (co-runners) on a single dual-thread core were performed. Results show that whatever the pair of benchmarks launched to run concurrently, IPC and L1 bandwidth values are significantly lower for both co-runners than those obtained in stand-alone execution. These performance drops are caused, among others, by the L1 bandwidth constraints.

To clearly show the impact of limited bandwidth on performance, the pair of behavior of the benchmarks selected to run concurrently must fulfill two key characteristics. First, each pair of threads must include at least one benchmark with high L1 bandwidth requirements. Notice that if the pair of co-runners does not consume significant L1 bandwidth, the impact of contention on performance will be less accentuated. Second, at least one of the co-runners must present a non-uniform shape. Otherwise, that is, if its bandwidth consumption is uniform (does not rise and fall), no significant insights will be appreciated on the resultant plot.

Figure 3 illustrates the results for three pairs of bench-



(a) TR$_{L1}$ of *bwaves* and *cactusADM*
(b) TR$_{L1}$ of *gamess* and *dealII*
(c) TR$_{L1}$ of *h264ref* and *bwaves*
(d) IPC of *bwaves* and *cactusADM*
(e) IPC of *gamess* and *dealII*
(f) IPC of *h264ref* and *bwaves*

Figure 3. TR$_{L1}$ and IPC dynamic evolution with time when running both benchmarks on a single SMT core

marks. For each pair, two plots are shown in the same column; the individual TR$_{L1}$ of both co-runners is shown in the upper side and individual IPC in the lower side. The presented results depict the dynamic evolution of both performance indicators during a fragment, the comprised from seconds 20 to 120, of the execution time, to ease the identification of trends. Different observations can be appreciated in this figure that can serve as a guide for designing thread allocation and scheduling policies.

The first observation is that when a pair of benchmarks runs concurrently on the same core, the individual L1 bandwidth utilization of each benchmark can significantly drop with respect to that achieved on stand-alone execution. Although such drop is expected, it is not clear how strong it will be. Notice that, in some cases, the TR$_{L1}$ drop is below 40%, what shows the importance of adequately sharing this resource. The second observation is that the individual L1 bandwidth consumption of a benchmark is strongly related with that of its co-runner. More precisely, when the use of L1 bandwidth of a benchmark drops, a large amount of bandwidth is available for the co-runner, so a positive side effect occurs which results in an increase of the co-runner's L1 bandwidth consumption.

As example, lets analyze this behavior in Figure 3(a) with *bwaves* and *cactusADM* as co-runners. The most interesting effect is the caused by *cactusADM* on the *bwaves*'s behavior. The TR$_{L1}$ of *bwaves* in stand-alone execution is regular although with important drops. However, the decreasing trend in the *cactusADM*'s L1 bandwidth requirements leaves more L1 bandwidth available to *bwaves*, which turns into an increase in its TR$_{L1}$. It can be also observed that when the TR$_{L1}$ of *bwaves* drops below 400 transactions per second, the bandwidth consumption of *h264ref* slightly rises. This behavior, even with a more accentuated impact, can be observed in the two other pairs of benchmarks. In summary, rises, drops and decreasing or increasing trends in the L1 bandwidth consumption of a benchmark trigger the opposite behavior in the co-runner.

Lets focus the analysis now on IPC values, which are shown in Figure 3 (lower row of plots). As stated in Section IV-A, the IPC achieved by a benchmark in stand-alone execution is strongly correlated with its consumed L1 bandwidth. An important finding is that this property is preserved even if a thread is sharing the L1 cache with a co-runner. Therefore, the previous analysis of the impact of the L1 bandwidth requirements on performance in stand-alone execution can be extended for two co-runners.

Putting together the previous observations and findings, we claim that sharp drops on the TR$_{L1}$ of a benchmark cause sharp loses on the benchmark performance (IPC), and trigger an opposite behavior (both in L1 bandwidth and IPC) in its co-runner. This pattern is also exhibited when benchmarks present slightly rising or dropping trends.

In summary, although multiple microprocessor components are shared in a SMT processor, L1 bandwidth contention can strongly drop the performance further than half with respect to stand-alone execution, showing that in such cases, L1 bandwidth contention becomes the major performance bottleneck. To reduce such bottleneck, this paper focuses on L1 bandwidth-aware thread allocation policies.

V. L1 BANDWIDTH-AWARE THREAD ALLOCATION POLICIES

The previous analysis illustrates the usefulness of designing L1 bandwidth-aware thread allocation policies. The aim of these policies is to properly balance L1 bandwidth requirements among cores in a timely manner in order to improve performance. Performance benefits vary depending on how far is the bandwidth required by the co-runners from that available in the shared L1 cache.

This section presents the devised Static (St2c) and Dynamic (Dt2c) thread to core allocation policies. Both policies rely on the L1 bandwidth demand of the running processes to guide the t2c allocation, but they differ on the way L1 bandwidth demand is estimated. The policies could be also considered as a part of a global scheduler that, before allocating threads to cores, selects the proper jobs to run the following quantum among all available processes. Below, these polices are described.

A. Static thread allocation policy

The Static t2c policy allocates threads to cores based on their average L1 transaction rate when they run alone in the system. The policy is referred to as static because it uses the average L1 bandwidth requirements of the threads, without taking into account dynamic deviations from this value. To be able to run a given thread using this policy, its average L1 bandwidth should be provided to the scheduler, which is an approach similar to that used in several bandwidth aware schedulers tackling main memory and LLC bandwidth [3] [6].

As stated before, the goal of the policy is to properly distribute the amount of accesses that all running threads perform among the L1 caches in the system. Since the experimental platform supports simultaneous execution for only two threads in each core, balancing L1 requests among cores can be easily done. For instance, threads can be ordered according to their L1 bandwidth requirements. Then, the threads with highest and lowest L1 bandwidth requirements can be selected to be run in the same core. This rule can be iteratively applied to form the remaining pairs of co-runners. If the SMT processor supports the execution of three or more threads it is possible to balance L1 requirements by calculating the cumulative TR_{L1} of all the threads and dividing this value by the number of cores. Then, threads can be properly allocated to the cores in order to minimize TR_{L1} differences among caches. Given that the allocation is guided by static measures, the t2c mapping only needs to be recalculated on a change in the running threads.

The advantage of using the L1 bandwidth metric to guide the allocation is that L1 bandwidth requirements are quite uniform over the execution time in a noticeable group of benchmarks (11 of the 25 analyzed), and thus it is a good approximation of the real requirements of the processes. Moreover, since these values are obtained while threads are running alone in the system any possible interference from other threads is avoided.

B. Dynamic thread allocation policy

The discussed St2c allocation policy presents two main shortcomings. First, it requires the processes to be run alone in order to estimate their average L1 bandwidth requirement before applying the thread allocation policy, which is not always possible. Second, the average L1 bandwidth requirements of benchmarks does not capture well the L1 bandwidth requirements of processes with non-uniform shapes like *astar*, *xalancbmk* or *mcf*. More precisely, when running such benchmarks, the St2c allocation policy does not discern among execution periods with high L1 bandwidth requirements from those with scarce requirements, which may cause suboptimal t2c mappings, that is, execution periods where L1 bandwidth requirements are not properly balanced among L1 caches.

The proposed Dt2c allocation policy tackles both of the aforementioned shortcomings. This policy uses the L1 bandwidth requirements that processes experience during their concurrent execution to guide the t2c mapping. Unlike the static policy, L1 bandwidth requirements of the running threads are dynamically obtained at run-time at the granularity of quantum. The L1 bandwidth requirements for the next quantum are assumed equal to the L1 bandwidth consumed during the last quantum, which is obtained using performance counters as stated in Section III. Since such values are gathered dynamically at run-time and at a smaller granularity, the policy should be able to provide better L1 bandwidth estimations for those threads presenting non-uniform bandwidth demands.

Balancing L1 requests among all the L1 caches can be performed as explained for the St2c allocation policy, to either dual-thread cores or cores with higher number of supported threads, but using the dynamically measured L1 bandwidth requirements of each thread, instead of the average value. Since bandwidth requirements are updated at the granularity of a scheduler quantum, the thread allocation process needs to be performed at the same granularity to provide L1 bandwidth balancing for each quantum.

As mentioned above, the main advantage of using dynamic L1 bandwidth measures is that the t2c mapping is adapted when a thread changes its L1 bandwidth requirements, and thus L1 bandwidth balancing is improved. For instance, L1 bandwidth requirements of benchmarks like *astar* or *mcf* can be properly addressed. That is, the t2c allocation can assign to the same core *astar* together with a benchmark with high L1 bandwidth requirements while *astar*'s L1 bandwidth requirements are low, and change its co-runner to another with lower bandwidth requirements as soon as the L1 bandwidth consumed by *astar* grows. In this way, L1 bandwidth distribution among cores is enhanced when compared with the St2c policy, where a given thread is launched to be run with the same co-runner during its complete execution.

VI. Evaluation Methodology

A. Methodology

To evaluate the effectiveness of the proposed policies, their performance is compared against the Linux OS scheduler and a *naive* thread allocation strategy. The four thread allocation policies have been implemented in a user-level Linux scheduler, sharing the main part of the code and only differing in the allocation algorithm. In this way, any possible overhead is shared among the studied policies and thus they can be fairly evaluated. The proposed policies are implemented following the explained algorithms. Linux OS allocation strategy is implemented by leaving to Linux the final decision about the t2c mappings. Finally, the naive t2c allocation policy dynamically allocates threads to cores such as the threads with higher L1 bandwidth requirements are allowed to run simultaneously in the same core. Quantum length for the schedulers is set to 200 microseconds, which is the granularity at which performance counters are accessed and the t2c mapping for the following quantum is obtained.

To avoid performance differences caused by early finalization of the execution of some benchmarks, which means that part of the execution in some cores will be performed by a

Classification	Benchmarks
Extreme L1 bandwidth	h264ref, bwaves, gamess
High L1 bandwidth	perlbench, bzip2, hmmer, libquantum, leslie3d, namd, dealII, gemsFDTD
Medium L1 bandwidth	gcc, gobmk, sjeng, astar, xalancbmk, zeusMP, povray, lbm
Low L1 bandwidth	mcf, omnetpp, milc, gromacs, cactusADM, soplex

Table I. BENCHMARK CLASSIFICATION ACCORDING TO THE L1 BANDWIDTH REQUIREMENTS

single co-runner and thus, without L1 bandwidth contention, we keep all benchmarks of the mix in execution until the last one executes its target number of instructions. This implies that some benchmarks will execute more instructions than the targeted number. For comparison purposes, we consider in these benchmarks only the performance metrics obtained while the target number of instructions is executed.

Thread allocation strategies are evaluated using two different metrics: average IPC and harmonic mean of weighted IPC. Average IPC of the threads composing a workload is the plain metric to measure throughput improvement between different runs of a workload. When evaluating schedulers, this metric can provide greater benefits to unfair scheduling strategies [24]. For example, at least for a while, it would be possible to improve the average IPC running the processes with highest IPCs. However, such scenarios are not possible with the proposed experimental methodology, since all the benchmarks are running until the complete execution of the mix. Thus, under the umbrella of our experimental methodology, average IPC is a good metric to quantify throughput improvement. To quantify fairness in addition to performance, the harmonic mean of weighted IPC [25] is typically used. This metric is interesting because most metrics quantify either performance or fairness independently, while this one encapsulates both of them. Fairness is captured by using the harmonic mean, which tends to be lower if any thread presents lower speedup than the remaining co-runners.

B. *Mix design*

According to the average L1 bandwidth requirements of the benchmarks in standalone execution, we classify them in four groups, presented in Table I. Benchmarks with higher L1 bandwidth utilization can potentially induce higher degradation in the co-runner and at the same time, they can suffer a strong degradation due to L1 bandwidth constraints. Thus, it is critical to allocate them sharing the core with the appropriate co-runners to enhance performance. Otherwise, significant performance losses will appear.

Based on the benchmark classification, mixes are classified according to the number of *extreme* benchmarks they have. The *balanced* mixes are formed with half the benchmarks belonging to the extreme L1 bandwidth category. These workloads have potential to offer greater benefits with a good t2c allocation since each benchmark with extreme L1 bandwidth demand can be allocated to a different core to run with a benchmark with lower L1 bandwidth requirements. The non-balanced mixes are formed with less *extreme* benchmarks than the number of cores. Since more threads can present intermediate bandwidth requirements, lower differences between distinct t2c mappings are expected.

We designed a wide variety of mixes consisting of up to twelve threads. In order to force that all the cores run two threads simultaneously, each mix is run on half the number of cores that threads contains the mix.

VII. THREAD ALLOCATION POLICIES EVALUATION

Performance of the proposed St2c and Dt2c policies is evaluated and compared against that of the Linux OS scheduler. A wide set of mixes has been evaluated for a different number of cores, ranging from two cores (four threads) to 6 cores (twelve threads). For each number of threads, we used mixes with different L1 bandwidth demands. We mingled *balanced* mixes with mixes presenting a lower number of benchmarks with *extreme* L1 bandwidth than cores.

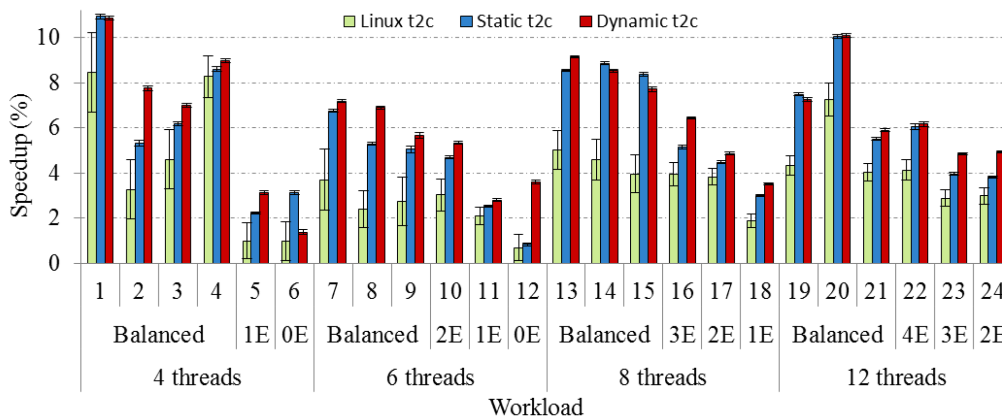


Figure 4. Speedup of the average IPC relative to the naive thread allocation strategy with 95% confidence intervals

Figure 4 presents the speedup of the average IPC achieved by the proposed policies and Linux for each mix over the naive t2c allocation policy, which has been used as baseline. Average values with 95% confidence intervals are represented. With XE we refer to a mix with X *extreme* benchmarks; e.g., 1E means only one *extreme* benchmark, that will be executed in one of the cores, while the remaining cores will not have any benchmark belonging to this category.

Compared to the Linux scheduler, the proposed policies achieve better performance across all the twenty-four evaluated mixes. While Dt2c and St2c policies provide speedups higher than 5% in seventeen and fifteen mixes, respectively, Linux scheduler only surpasses this value in four mixes. On the contrary, the speedup of Linux scheduler falls around or below 2% in six mixes, while St2c and Dt2c policies only do that in one mix.

As observed, the Dt2c allocation policy performs better, on average, than the St2c allocation policy. Significant differences can be appreciated in some mixes like 2, 3, 8, 12, 16 and 24. The major differences appear when the mix includes benchmarks showing a non-uniform shape in their L1 bandwidth requirements. For instance, mix 2 includes *bwaves* and *cactusADM*, which present a non-uniform shape. On the contrary, mix 1 shows minor differences since all benchmarks present an almost uniform shape in their L1 bandwidth consumption. The only exception in which St2c provides significant benefits over the Dt2c policy is in mix 6. The reason is that includes the *GemsFDTD* benchmark, whose L1 bandwidth utilization varies so fast (see Figure 2(l)) that the Dt2c policy is not able to accurately predict the bandwidth requirement for the next quantum.

As expected, the policies offer higher performance when running balanced workloads. As the number of *extreme* threads drops in the mix, the achieved speedup is on average smaller since the L1 bandwidth contention is reduced. Nonetheless, performance differences among mixes also come from the characteristics of the non-extreme benchmarks. For example, mix 20 has one and five benchmarks with medium and low L1 bandwidth demand, respectively; while mix 21 includes one, three and two benchmarks with high, medium and low L1 bandwidth consumption. Since bandwidth differences among possible pairs can be higher in mix 20 than in mix 21, one should expect major performance benefits from appropriate t2c mappings in such mix. Thus, even in non-balanced workloads (e.g. 12, 16, 22, 23 and 24), noticeable performance benefits can be achieved.

Notice too that confidence intervals of the Linux t2c policy are considerably larger than those of the St2c and Dt2c allocation policies. This is due to the fact that Linux does not consider L1 bandwidth to perform the allocation. Therefore, its thread to core mappings greatly vary along different instances of the experiment, and consequently, their corresponding performance. On the other hand, the confidence intervals for the devised policies are usually below 0.1%, ensuring that the achieved speedups are stable among executions.

Finally, the performance of the proposed policies scale well with the number of threads. Nevertheless, the number of accesses to main memory is expected to grow with the number of threads. Thus, it may happen that LLC and main memory contention grow so creating a new contention point in such memory structures. On such a case, the proposed t2c policies could be combined with main memory and LLC bandwidth-aware schedulers to tackle such contention points.

Looking at Figure 5, which shows the speedups using the harmonic mean of weighted IPC, the same conclusions can be drawn. The speedup values are slightly reduced, however, differences between the performance of the Dt2c policy and the St2c policy are wider (e.g., mixes 3, 7, 17 and 24). Thus, one can conclude that the Dt2c allocation policy is the best one since this metric evaluates both performance and fairness.

Average values do not reflect what is happening over time. To provide insights and a sound understanding about how the different policies work with time, lets analyze the behavior of mix 2. In this mix, the St2c policy significantly improves Linux performance, and at the same time, the Dt2c policy considerably improves the performance of the St2c policy. Figure 6 shows the dynamic TR_{L1} of each benchmark during the complete execution of the mix under the studied t2c allocation policies.

Notice that the Linux and St2c policy plots are quite similar during the first 250 seconds. According to the TR_{L1}

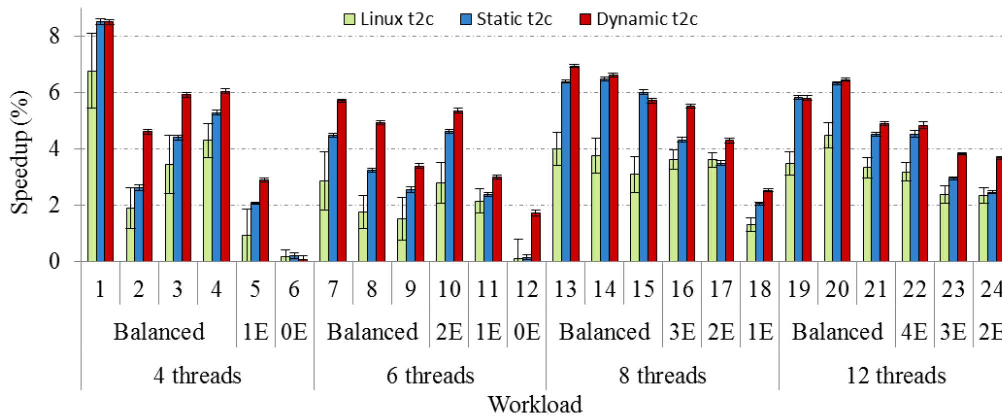
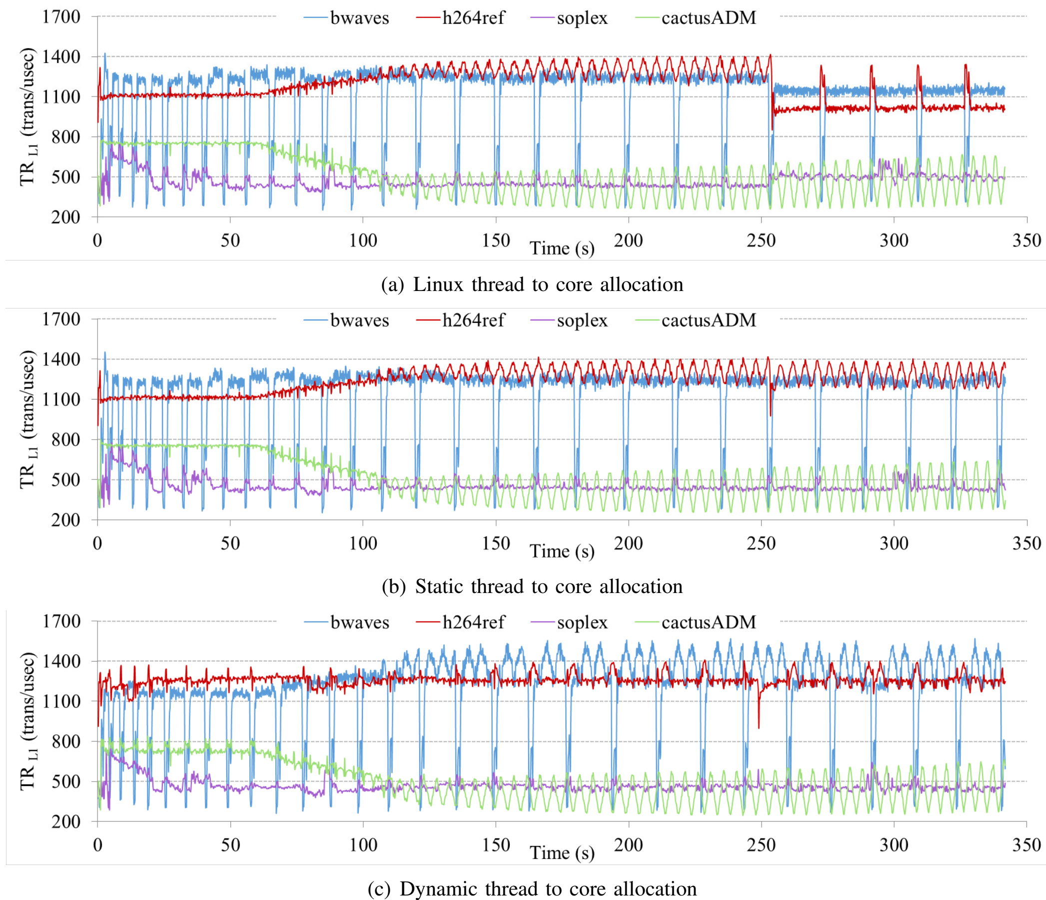


Figure 5. Speedup of the harmonic mean of weighted IPC relative to the naive thread allocation strategy with 95% confidence intervals



(a) Linux thread to core allocation

(b) Static thread to core allocation

(c) Dynamic thread to core allocation

Figure 6. TR$_{L1}$ of benchmark in mix 2 varying the tread to core allocation policy

curves, one can deduce that *h264ref* and *cactusADM* were running on one core and *bwaves* and *soplex* on the other one. Around second 250, Linux changes the thread to core mapping and starts running together *h264ref* and *bwaves*. This can be deduced because the rises in the TR$_{L1}$ curve of *h264ref* are synchronized with the drops of *bwaves*. However, notice that in spite of this thread to core mapping yields to lower performance, Linux keeps it until the end of the execution.

Unlike the previous policies, the Dt2c policy usually selects as co-runners *bwaves* and *cactusADM*, which according to the observed TR$_{L1}$ is the best choice. As observed, *Bwaves* obtains regular peaks around 1500 trans/usec, while the maximum TR$_{L1}$ does not surpass 1400 trans/usec in the other two t2c policies. Finally, when *bwaves* experiences sharp drops in its TR$_{L1}$ curve, the Dt2c policy benefits the *h264ref* benchmark, which at that point, is the more consuming L1 bandwidth benchmark of the remaining ones. Consequently, the L1 bandwidth of *h264ref* rises occurred during drops int the curve of *bwaves* are higher than those obtained by *soplex* in the St2c policy, thus, enhancing the performance.

VIII. Conclusions

This work has addressed the L1 bandwidth contention in current multithreaded CMPs and has proven that by addressing the L1 bandwidth distribution in SMT multicores, performance enhancements can be achieved.

The relation between IPC and TR$_{L1}$ of the benchmarks in standalone execution has been analyzed, showing that both metrics are strongly connected and follow the same shape over their execution time. When two threads run on a dual-thread SMT core, they share the available L1 bandwidth, which many times is not enough to satisfy their requirements. Results have shown that trends, rises and drops in the curve of the L1 bandwidth consumption of a given thread trigger the opposite behavior in the co-runner. Moreover, we found a strong connection between TR$_{L1}$ and IPC of a given thread in stand alone execution, which is preserved when various threads are executed concurrently in the SMT core.

According to the previous findings, if the L1 requests are properly balanced among the processor cores, then the L1 bandwidth contention should be reduced, so increasing the L1 bandwidth that threads can consume and consequently improving their performance. To exploit this idea, we have proposed two t2c allocation policies with the aim of improving the L1 bandwidth balancing. The St2c policy uses the average L1 bandwidth requirements of the threads to obtain the t2c mapping, while the D2tc policy dynamically accesses performance counters to update the L1 bandwidth requirements of the thread at runtime and adapt the t2c mappings.

Experimental evaluation on a Xeon E5645 have shown that both policies significantly improve the performance with respect to the Linux OS scheduler, which in many cases is unable

to improve the performance of a naive policy further than 1%. In contrast, the proposed Dt2c policy achieves speedups as high as 10% over the naive scheduler and doubles the speedups obtained by the Linux OS scheduler in most of the evaluated mixes. Finally, the proposed thread allocation policies can be combined with memory bandwidth-aware schedulers proposed for CMPs and sharing resource strategies for SMTs in order to improve the overall system performance.

ACKNOWLEDGMENTS

This work was supported by the Spanish Ministerio de Economía y Competitividad (MINECO) and by FEDER funds under Grant TIN2012-38341-C04-01; and by Programa de Apoyo a la Investigación y Desarrollo (PAID-05-12) of the Universitat Politècnica de València under Grant SP20120748.

REFERENCES

[1] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: maximizing on-chip parallelism," *SIGARCH Comput. Archit. News*, vol. 23, no. 2, pp. 392–403, May 1995.

[2] C. D. Antonopoulos, D. S. Nikolopoulos, and T. S. Papatheodorou, "Realistic workload scheduling policies for taming the memory bandwidth bottleneck of smps," in *High Performance Computing (HiPC)*, 2004, pp. 286–296.

[3] D. Xu, C. Wu, and P.-C. Yew, "On mitigating memory bandwidth contention through bandwidth-aware scheduling," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2010, pp. 237–248.

[4] L. Tang, J. Mars, N. Vachharajani, R. Hundt, and M.-L. Soffa, "The impact of memory subsystem resource sharing on datacenter applications," in *International Symposium on Computer Architecture (ISCA)*, 2011, pp. 283–294.

[5] S. Zhuravlev, S. Blagodurov, and A. Fedorova, "Addressing shared resource contention in multicore processors via scheduling," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010, pp. 129–142.

[6] J. Feliu, J. Sahuquillo, S. Petit, and J. Duato, "Understanding Cache Hierarchy Contention in CMPs to Improve Job Scheduling," in *International Parallel Distributed Processing Symposium (IPDPS)*, 2012, pp. 508 –519.

[7] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn, "Using os observations to improve performance in multicore systems," *IEEE Micro*, vol. 28, no. 3, pp. 54–66, may 2008.

[8] A. Fedorova, M. Seltzer, and M. D. Smith, "Improving performance isolation on chip multiprocessors via an operating system scheduler," in *International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2007, pp. 25–38.

[9] Y. Jiang, K. Tian, and X. Shen, "Combining locality analysis with online proactive job co-scheduling in chip multiprocessors," in *International Conference on High Performance Embedded Architectures and Compilers, (HiPEAC)*, 2010, pp. 201–215.

[10] S. Eyerman and L. Eeckhout, "Probabilistic job symbiosis modeling for smt processor scheduling," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010, pp. 91–102.

[11] V. Čakarević, P. Radojković, J. Verdú, A. Pajuelo, F. J. Cazorla, M. Nemirovsky, and M. Valero, "Characterizing the resource-sharing levels in the ultrasparc t2 processor," in *International Symposium on Microarchitecture (MICRO)*, 2009, pp. 481–492.

[12] C. Acosta, F. J. Cazorla, A. Ramirez, and M. Valero, "Thread to core assignment in smt on-chip multiprocessors," in *International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2009, pp. 67–74.

[13] L. Weng and C. Liu, "On better performance from scheduling threads according to resource demands in mmmp," in *International Conference on Parallel Processing Workshops (ICPPW)*, 2010, pp. 339–345.

[14] F. J. Cazorla, A. Ramirez, M. Valero, and E. Fernandez, "Dynamically controlled resource allocation in smt processors," in *International Symposium on Microarchitecture (MICRO)*, 2004, pp. 171–182.

[15] S. Choi and D. Yeung, "Learning-based smt processor resource distribution via hill-climbing," in *International Symposium on Computer Architecture (ISCA)*, 2006, pp. 239–251.

[16] H. Wang, I. Koren, and C. M. Krishna, "An adaptive resource partitioning algorithm for smt processors," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2008, pp. 230–239.

[17] M. Moreto, F. J. Cazorla, A. Ramirez, and M. Valero, in *Transactions on High-Performance Embedded Architectures and Compilers III*, 2011, ch. Dynamic cache partitioning based on the MLP of cache misses, pp. 3–23.

[18] M. Moreto, F. J. Cazorla, R. Sakellariou, and M. Valero, "Load balancing using dynamic cache allocation," in *International Conference on Computing Frontiers (CF)*, 2010, pp. 153–164.

[19] S. Srikantaiah, M. Kandemir, and Q. Wang, "Sharp control: controlled shared cache management in chip multiprocessors," in *International Symposium on Microarchitecture (MICRO)*, 2009, pp. 517–528.

[20] R. Manikantan, K. Rajan, and R. Govindarajan, "Probabilistic shared cache management (prism)," in *International Symposium on Computer Architecture (ISCA)*, 2012, pp. 428–439.

[21] J. Chen and L. K. John, "Predictive coordination of multiple on-chip resources for chip multiprocessors," in *International Conference on Supercomputing (ICS)*, 2011, pp. 192–201.

[22] R. Bitirgen, E. Ipek, and J. F. Martinez, "Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach," in *International Symposium on Microarchitecture (MICRO)*, 2008, pp. 318–329.

[23] S. Eranian, "What can performance counters do for memory subsystem analysis?" in *Proceedings of the ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, 2008, pp. 26–30.

[24] A. Snavely and D. M. Tullsen, "Symbiotic jobscheduling for a simultaneous mutlithreading processor," *SIGPLAN Not.*, vol. 35, no. 11, Nov. 2000.

[25] K. Luo, J. Gummaraju, and M. Franklin, "Balancing Thoughput and Fairness in SMT Processors," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2001, pp. 164–171.