Shared Cache Partitioning Based on Performance Gain Estimations

N Mahrom\(^1\) and M J Liebelt\(^2\)

\(^1\) School of Computer and Communication Engineering, Universiti Malaysia Perlis, Arau, 02600 Perlis, Malaysia
\(^2\) School of Electrical and Electronic Engineering, The University of Adelaide, South Australia, 5005 Australia

norfadila@unimap.edu.my

Abstract. In multiprocessor systems, dynamic cache distribution has been used to increase system performance by effectively partitioning the cache resources. However, different performance metrics used at runtime used to dynamically decide the partition sizes can give different impacts on performance, as well as varying impacts on the hardware cost of the system. In this paper, we propose an Adaptive CPI-based Cache Partitioning (ACCP) scheme to provide better utilisation of the shared cache resources among the competing applications in the system. ACCP uses performance gain estimations of the cache, without incurring significant hardware overhead. It aims to allow all applications in the system to run at approximately the same speed by accelerating the slowest application without significantly decelerating the others. We evaluated the ACCP on a quad-core system on which it achieved on average 23% reduction in miss rate, compared to an unpartitioned shared cache. ACCP also yields a similar IPC throughput improvement to a well-known UCP scheme, and better performance compared to the CPI by Muralidhara et al. Overall, the throughput of the system is improved at minimal complexity without yielding significant additional hardware cost. Hence, ACCP shows better overall performance in managing the hardware overhead compared to the UCP scheme.

1. Introduction

Different applications competing for shared memory resources in multiprocessor systems have different behaviours and requirements throughout the execution process. The performance of the applications can be characterised by many factors, such as the speed of the slowest application, or the maximum number of misses incurred by an application in the system. There are several trade-offs between these factors in optimising system performance. Consequently, various techniques have been proposed to enhance the performance of multiprocessor systems and to improve the overall shared cache throughput \([1][2][3][4]\). An Adaptive CPI-based Cache Partitioning (ACCP) scheme is proposed to modify the cache space allocation of applications in the system by up to four ways at the end of every execution interval. At the end of each interval, the shared cache space is partitioned based on the cycles per instruction (CPI) values in such a way that the application with the highest CPI (the slowest application) will receive extra ways from the application with the lowest CPI (the fastest application). However, to ensure that the slowest application will get benefit from the additional cache ways, its miss counter associated with the
most recently used (MRU) position is compared with the MRU miss counter of the fastest application. If the value of the counter belonging to the slowest application is less than the value in the miss counter of the fastest application, another application in the system will be selected as the slowest application (i.e., the application with the highest MRU miss count).

In a case having four applications running on different cores, the application with the next highest CPI value will not be selected. This is because there is a possibility that the MRU miss counts of this application is less than the MRU miss counts of the fastest application or the remaining applications in the system. The application with the highest MRU miss count is used to avoid an iterative search through applications with successively lower CPI values. The process of determining the number of cache ways to be given to the slowest application is described in Algorithm 1. The miss counters of the fastest and the slowest applications are read to estimate the performance gain. Note that the miss counts in the counters are correlated with the number of hits, had the respective application been given additional cache ways.

**Algorithm 1.** Adaptive CPI-based Cache Partitioning scheme.

- Start with equal partition for each competing application $i$, at the first interval.
  \[
  allocation[i] = \frac{\text{total cache ways}}{\text{number of cores}}
  \]
- At the end of each execution interval, do
  - Read instruction counts, $C_{\text{INS}}[i]$ and cycle counts, $C_{\text{CYC}}[i]$ for each application and calculate the CPI values.
  - Determine the slowest application, $\text{slowest\_app}$ and the fastest application, $\text{fastest\_app}$ based on the calculated CPI values.
    - Check the first miss counter, $C_{\text{MISS}}[1]$ of the $\text{slowest\_app}$ and the $\text{fastest\_app}$.
    - If $C_{\text{MISS}}_{\text{slowest\_app}}[1] < C_{\text{MISS}}_{\text{fastest\_app}}[1]$, then another application in the system that has the highest value of $C_{\text{MISS}}[1]$ will be assigned as the new $\text{slowest\_app}$.
  - Next, evaluate $j$ miss counters, $C_{\text{MISS}}[j]$ that belongs to the $\text{slowest\_app}$ and the $\text{fastest\_app}$ to determine the number of ways to modify.

function PARTITION THE SHARED CACHE

\[
\text{max\_gain} = 0 // \text{To check if adding one more way can provide further benefits}
\text{extra\_ways} = 0 // \text{Number of ways to modify}
\]

for each miss counter $j = 1$ to 4, do

- \[
\text{new\_max\_gain} = \text{max\_gain} + C_{\text{MISS}}_{\text{slowest\_app}}[j]
\]
- if (\text{new\_max\_gain} > \text{max\_gain}) and ($C_{\text{MISS}}_{\text{slowest\_app}}[j] > C_{\text{MISS}}_{\text{fastest\_app}}[j]$), then
  - $\text{max\_gain} = \text{new\_max\_gain}$
  - $\text{extra\_ways} += 1$
- else
  - break;
  - end if
- end for

if (allocation[fastest\_app] – extra\_ways > 0)

- allocation[fastest\_app] -= extra\_ways
- allocation[slowest\_app] += extra\_ways
- end if

return allocation

end function

In this paper, we do not estimate the performance loss that will cost the fastest application because several more counters would be required to record the number of hits incurred at each priority position.
of the lines in a set. The number recorded in these counters will represent the number of misses that will be received by an application if some ways are taken away from it. That is, the hit counts at the least recently used (LRU) position represents the number of additional misses that will be incurred by an application if a cache way is removed from it. If two cache ways are taken away from the application, the number of hits incurred at the second LRU position will represent the additional number of misses to be received by the application. Although the use of hit counters to estimate the performance loss of each application seems beneficial, it will cost additional hardware and processes at every execution interval. We chose to implement an algorithm that does not require this additional overhead.

To estimate performance gain from additional cache ways, the miss counters of the slowest application are compared with the counters of the fastest application, one at a time. The first counter to be checked is the one associated with the MRU position that represents the performance gain if the application were to be given an extra one way. In sequence, the counters are compared from the one associated with the MRU position to the one associated with the LRU position. Each time a counter is checked, if the miss count of the slowest application is greater than the miss count of the fastest application, an extra way is given to the slowest application. This process is repeated for all four counters, or until the value of the miss count belonging to the slowest application is less than the miss count of the fastest application. Note that this scheme guarantees that each application owns at least one cache way by making sure that the total number of ways given to the slowest application by the fastest application will leave at least one cache way to the fastest application. Otherwise, the cache will not repartition. All the miss counters of all applications are reset to zero after each partitioning interval.

2. Experimental Methodology

2.1. Configuration
Our baseline architecture is based on the Tensilica Xtensa toolset [5] and the Xtensa Xplorer is used to configure and customise the Xtensa LX4.0 processor cores. The Xtensa design tool provides a cycle accurate system simulator to perform software emulation of multiprocessor systems. We configured the Xtensa cores by using Xtensa Processor Generator (XPG) without local data and instruction caches. The Xtensa cores used in our work are configured with the Processor Interface (PIF) in which the read and write busses are of equal width, 32-bit. This is because PIF supports external communication for inter-core communications in an Xtensa multiprocessor system. Additionally, PIF does not limit the size of external peripherals that can be connected to the core. We created all memories (e.g. caches) in our system as custom memory-mapped devices using XTMP_device objects. In Xtensa Modeling Protocol (XTMP), a memory-mapped device refers to any component in the system that is accessed by load or store transactions from a simulated Xtensa core [6]. We used 4KB level 1 private cache and 512KB level 2 shared cache. The level 2 cache requires 15 cycles hit latency, while the main memory requires 300 cycles access latency.

2.2. Metrics
For all of the ACCP evaluations, the performance metrics used to quantify the performance of the system are the MPKI (misses per 1000 instructions), IPC throughput, weighted speedup, which indicates reduction in execution time, and the harmonic mean of weighted speedup, which accounts for both fairness and performance of the system [7][8].

Let \( IPC_i \) denote the IPC of the \( i \)th application, \( SingleIPC \), the stand alone IPC of the same application if it is executed in isolation, and \( N \) the number of processes the system executes concurrently. The formulae for the aforementioned metrics are:

\[
IPC\ Throughput = \sum_{i=0}^{N} IPC_i
\]

\[
Weighted\ Speedup = \sum_{i=0}^{N} IPC_i/SingleIPC_i
\]
\[ Harmonic\ Mean = \frac{N}{\sum_{i=0}^{N} (SingleIPC_i/IPC_C)} \] (3)

2.3. Benchmarks

The performance evaluation of different cache replacement schemes has been undertaken using sets of applications that were selected from the SPEC CPU2000 benchmark suite. As the Tensilica Xtensa development tool includes the Xtensa C/C++ compiler (XCC), which supports compiling C and C++ applications, only applications that are written in C were selected to be used for evaluation purposes. The applications were compiled with Xtensa’s optimising compiler xt-xcc at the -O3 optimisation level. The chosen applications are the combinations of three different applications categories, which are classified based on the demand for L2 cache space. They are named high utility, low utility, and saturating utility applications [9].

Low utility applications are benchmarks that do not benefit from the allocation of any additional cache space. Applications named mcf, bzip2, and equake are under this category. The argument could be that these applications fit in the L1 cache and suffer frequent compulsory misses in the L2 cache. High utility applications improve their performance whenever more L2 cache space is allocated. They are twolf, vpr, parser, art, and ammp. However, if increasing the available cache space can improve the performance of applications up to a point at which their performance saturates, these applications were classified as saturating utility. The applications are gzip, mgrid, and mesa. All the applications from different categories were combined to represent eleven workload mixes as shown in Table 1.

| Workload | Applications | Utility |
|----------|--------------|---------|
| Mix-1    | art, bzip2, parser, vpr | HLHH    |
| Mix-2    | vpr, parser, mcf, bzip2  | HHLL    |
| Mix-3    | ammp, twolf, vpr, art    | HHHH    |
| Mix-4    | equake, bzip2, twolf, vpr| LLHH    |
| Mix-5    | bzip2, ammp, vpr, art    | LHHH    |
| Mix-6    | mcf, vpr, twolf, gzip    | LHHS    |
| Mix-7    | parser, mesa, vpr, art   | HSHH    |
| Mix-8    | twolf, art, bzip2, mgrid | HHLH    |
| Mix-9    | art, vpr, twolf, equake  | HHHL    |
| Mix-10   | equake, bzip2, art, vpr  | LLHH    |
| Mix-11   | ammp, vpr, twolf, bzip2  | HHHL    |

3. Simulation Results

We evaluated the ACCP scheme on a quad-core system sharing a 512KB, 32-way associative L2 cache. The shared cache was warmed up for 10 million instructions and the partitioning algorithm was invoked after every execution interval of 10 million instructions. Note that the size of the execution intervals of utility cache partitioning (UCP) scheme [10] and CPI cache partitioning scheme [11] are 5 million and 15 million, respectively. Therefore, we decided to use 10 million instructions for comparability with the previously published schemes. We analysed and compared the results of ACCP against two existing cache partitioning schemes, namely the UCP scheme [10] and CPI [11]. We forced the entire system including all the cores to shut down when any one of the cores reached the end of its execution. For a system employing UCP, the traditional LRU policy is modified so that it works in a thread-wise manner, in which the victim selection must be among the cache lines belonging to the miss-causing application.
[10]. However, in the case where the number of allocated cache lines is less than the number of cache lines currently belonging to the miss-causing application, a cache line belonging to another application is replaced.

For all of the performance comparisons, the conventional unmanaged/unpartitioned shared cache using the LRU policy is used as the baseline. Figure 1 shows the miss rate (in misses per 1000 instructions, MPKI) of ACCP, alongside the performance of UCP and CPI. While ACCP was proposed to simplify the complex partitioning decision logic in UCP and to improve the partitioning conditions in CPI, ACCP on average has yielded a performance rate similar to both schemes. However, ACCP in a few cases is able to provide miss rate improvements over UCP and CPI by up to 5% and 9%, respectively. Even though for the majority of the workloads ACCP incurred approximately similar miss counts to UCP and CPI-based, the results show that our approach is able to achieve cache miss reductions as much as UCP, with less hardware overhead. Furthermore, comparison with CPI shows the ability of ACCP to improve the miss rates by considering both CPI values and miss rate.

![Figure 1. Miss rate improvements relative to an unmanaged traditional LRU cache.](image1)

Our findings in Figure 1 are supported by the IPC throughput observed across the simulated workloads, shown in Figure 2. The IPC throughput of Mix3 when using ACCP was 4.3% better than the IPC throughput of CPI, although it was observed that ACCP has incurred a 5% additional miss rate. A similar pattern is demonstrated by Mix11, for which ACCP yields a 2% improvement in IPC over CPI, whereas the MPKI figures recorded by both schemes show a negligible difference. ACCP performs better than CPI because while the use of IPC values can give good predictions in partitioning the shared cache, the estimation of performance gain due to the increasing of cache resources can result in better partitioning decisions of the slowest application and further improve the system performance. Hence, ACCP has demonstrated that for the majority of the workloads, the combination of CPI values and performance gain estimation could be used effectively to partition the shared resources.

![Figure 2. Performance comparison of UCP, CPI and ACCP relative to an unmanaged LRU L2 shared cache.](image2)
Additionally, we found that a simpler partitioning algorithm and a lower implementation overhead do not limit the ability of ACCP to achieve similar performance to UCP. ACCP also has an advantage over UCP, in that it can be employed with any type of cache replacement policy. To investigate the effect of ACCP on each application within a workload, Figure 3 shows speedups achieved by ACCP compared to UCP and CPI.

We observed that although the patterns for both weighted speedup and harmonic means of the IPC are similar to the achieved IPC throughput reported in Figure 2, there are further improvements exhibited by ACCP. In addition, ACCP performs similarly to or slightly better than UCP for the fairness performance of several mixes as illustrated in Figure 3 (a). ACCP compares more favourably when performance is measured with the harmonic mean, as in Figure 3 (b) which shows that ACCP in some cases outperforms UCP, and has a positive impact on the applications’ throughput of the workload mixes. Comparison with CPI shows that the fairness speedup of ACCP is better than its weighted speedup by up to 2%. Thus, we show that ACCP is able to assist the fairness of the workload mixes, and hence meets the objective of the scheme which is to reduce the speed difference of all running applications.

Figure 3. Performance comparison using the (a) weighted speedup, and (b) harmonic mean of weighted IPC metrics.

4. Performance Analysis on Cache Scaling and Sensitivity
We further consider the performance of ACCP by analysing its effect on 1MB and 4MB L2 shared caches. Figure 4 shows the performance of ACCP is largely unaffected by changes in cache sizes. This means that ACCP retains similar performance trends, and consistently achieved similar performance to UCP. Also, ACCP consistently outperforms the CPI for the majority of the simulated workloads, with the highest improvement recorded in a 1MB cache at 1.5% over UCP and 3.7% over the CPI, for Mix11. Meanwhile, with a cache size of 4MB ACCP for Mix10 outperforms the CPI by the maximum value of 5.6% and UCP by a maximum of 6.9%. This better performance could be due to the total benefits of the
larger cache size received by all high utility applications in the mix that contribute to the significant overall performance compared to other workload mixes. Hence, ACCP has provided sufficient cache space to the high utility applications in Mix10 to achieve higher overall performance improvement compared to other schemes.

![Figure 4. Performance comparison of ACCP, UCP and CPI on different sizes of L2 shared cache, relative to baseline LRU.](image)

Next, the effect of different cache sizes is investigated in terms of the MPKI metric. Figure 5 depicts the MPKI of ACCP normalised to the MPKI of the UCP and CPI. An interesting point to observe in the figure is that in the majority of cases, ACCP performs better over CPI than it does over UCP. For the 1MB cache, the reduction over CPI is between 5% and 10% greater than the reduction relative to UCP. However, the reduction of ACCP recorded over CPI for the 4MB cache is more than 10% better than the reduction over UCP. The most significant reduction exhibited by ACCP in the 4MB cache size occurs for Mix10, in which the recorded MPKI is less than 20% that of UCP. This is correlated with the performance value recorded in Figure 4, in which ACCP achieved the highest improvement over UCP in the case of Mix10. ACCP shows a reduction of less than 5% on average in the 1MB cache and about 5% to 10% in the 4MB cache over the UCP and CPI, respectively. However, we observed in Figure 5 that ACCP performed poorly in Mix7 (for the 1MB cache), in which it incurred around 20% more misses compared to the UCP and CPI. For the 4MB cache, ACCP achieved the highest improvement in the miss rate in Mix10 compared to UCP and CPI.

![Figure 5. Miss rate reduction of ACCP over UCP and CPI.](image)

To identify the contributing factor to the significant performance variation of ACCP in the two mixes, the distribution of the miss counts among the cores in Mix7 (for 1MB cache) and Mix10 (for 4MB cache) is shown in Figure 6. For Mix10 in the 4MB cache, the ACCP has significantly reduced the number of misses in core C2, which executed a high utility application. This application could have
received most of the 4MB cache space from ACCP, causing the significant reduction of the miss counts in this core to achieve the highest overall miss reduction of Mix10, compared to the UCP and CPI.

For the 1MB cache, we found that core C1 of Mix7 incurred the highest number of misses compared to other cores in the system. Even though core C1 also executed a high utility application, the fact that the core did not get much benefit from the 1MB cache could be due to competition with other high utility applications executed by cores C3 and C4. Hence, the significant number of misses incurred by core C1 has contributed to the worse performance of Mix7 compared to the UCP and CPI. Note that an application could perform better or worse due to competition with other cores in the system, since the application could be treated differently from its type. This means that a high utility application could be treated as a streaming process in one mix, but could get many advantages from the available cache space in another workload mix. Therefore, this could account for the different results produced by an application in different mixes, leading to variation of performance improvements among mixes.

5. Complexity Management and Hardware Implementation

In Algorithm 1, the fastest and the slowest applications are chosen based on the CPI values, then the four miss counters of the two applications are read and compared. The values of the counters indicate how much the applications could benefit from the allocation of another 1, 2, 3 or 4 ways. Primarily, the partitioning algorithm of ACCP requires the implementation of addition, subtraction, and comparisons. However, ACCP does not employ any complex control logic due to the use of few if statements, in which the if statements can be directly implemented in hardware by using a compare-and-multiplex operation. Furthermore, the number of iterations that the miss counters need in order to decide the number of ways to reallocate can be determined in advance. Hence, comparisons between each miss counter can be done in parallel. Additionally, there is no control-flow dependence on the input values of the iteration counts, except the dependence on the slowest application identifier. This scenario therefore permits the implementation and parallelisation of simple control logic for the algorithm to make the partitioning decisions.

The strategy of examining each application’s performance gain by increasing up to four cache ways has reduced the hardware cost of monitoring and gathering information on all applications in the system. Comparison with a well-established scheme, namely UCP, shows that the ACCP scheme uses considerably less storage and fewer counters. In order to monitor the behaviour of each competing application, had the applications received four additional cache ways, ACCP needs only four tag registers per set and four miss counters assigned to each application. The counters used in this work were assumed to be 32 bits long, but from our observation, the size can be reduced in practice for the following reasons. Firstly, the monitoring period was reset every ten million instructions. Hence, 24-bit counters are sufficient to be used for the instruction counters. Secondly, the miss counts of the miss counters recorded in the simulation results showed that the numbers of misses could be contained well within the 24-bit counters.

![Figure 6. Distribution of L2 miss rate among UCP, CPI and ACCP.](image-url)
Now consider a 4MB, 32-way associative cache assuming a 32-bit physical address space. The storage overhead that will be incurred for a quad-core system if all sets are monitored and evaluated by ACCP is presented in Table 2. Note that, as ACCP requires each cache line to have a core ID, the length for the identification is \( \log_2 N \) bits. Thus, the overhead due to the identification fields is \( 8192 \times 16 \times 2 \text{bits} = 32KB \), an increase of 0.66\%. In total, the storage overhead for a quad-core system is \( 4 \text{cores} \times 68.02\text{KB} + 32\text{KB} = 304.08\text{KB} \), an increment of 6.3\% over the L2 cache area. Moreover, the aforementioned adders and comparators are needed to update the counters and to perform partitioning decisions in the ACCP. This will increase the total hardware overhead slightly over the indicated value even though the additional overhead is relatively small. However, by using DSS to reduce the hardware overhead, the actual hardware cost for monitoring 256 sets by ACCP is reduced. To achieve such a low hardware overhead, ACCP will cost each core 2.14KB of additional hardware due to the shadow tag arrays and counters to record the CPI values, which is an increment of 0.04\% in the L2 cache area. As a result, implementation of ACCP will add an extra 0.2\% area overhead towards the total shared cache area of a quad-core system.

### Table 2. Hardware overhead per processor.

| Type of Overhead | Value   |
|------------------|---------|
| Shadow tag array entry \((15 \text{ bits tag} + 2 \text{ bits LRU}) \times 4 \text{ positions}\) | 8.5B |
| Total shadow tag array overhead \((8192 \text{ sets} \times 8.5\text{B})\) | 69,632B |
| Miss counters overhead \((4 \text{ counters} \times 3\text{B})\) | 12B |
| CPI counters overhead \((2 \text{ counters} \times 3\text{B})\) | 6B |
| Total overhead | 68.02KB |
| Area of baseline L2 cache \((736\text{KB} \text{ tags} + 4\text{MB})\) | 4832KB |
| % increase in L2 cache area due to a shadow tag array | 1.4\% |

### 6. Conclusion
ACCP attempts to improve the utilisation of a shared cache among multiple cores and to manage the progress of running applications in the system at approximately the same speed, by accelerating the slowest application without significantly slowing any others, while maintaining low hardware overhead. From the simulations that have been conducted, ACCP has achieved similar or better performance than existing cache partitioning schemes. Comparison with the complex UCP scheme, which requires higher hardware overhead than ACCP, has shown that the ACCP is capable of achieving similar performance, even though ACCP adjusts the cache allocation of only two cores in the system. The performance gain estimation used in ACCP also has outperformed the CPI in implementing effective partitioning decisions. While it was proven by the CPI that CPI values could be used as the sole metric to determine better partition sizes, it is found from our investigations that the performance gain estimation can produce significantly more effective partitioning decisions. This is because the performance gain estimation reflects how much an application would benefit from additional cache ways.

Overall, ACCP shows that referring solely to CPI values to determine the cache partition sizes is less effective than also using information provided by historical memory accesses made by each application in the system. Evaluation on the lines that have been evicted from the cache in order to make better partitioning decisions can give more flexibility to the partitioning scheme and the system to gain further benefits from the employment of both an optimal partitioning scheme and an optimal cache replacement policy. This is because the applications’ behaviours and their cache activities will not have any impact on the estimation of performance gain. Finally, the analysis made on the hardware cost has demonstrated that the new ACCP scheme incurs a very low hardware overhead. Hence, the proposed scheme can be
effective in improving CMP system performance without incurring the significant hardware overhead of UCP.

References
[1] Sato M, Nishimura S, Egawa R, Takizawa H and Kobayashi H, (2016). “A cache partitioning mechanism to protect shared data for CMPs” in IEEE Symposium in Low-Power and High-Speed Chips (COOL CHIPS XIX), 2016, p 1-2.
[2] Shen F, He Y, Zhang J, Li Q, Li J and Xu C (2019). Reuse locality aware cache partitioning for last-level cache. Computers & Electrical Engineering, 74, p 319-330.
[3] Gupta S and Zhou H (2015) “Spatial locality-aware cache partitioning for effective cache sharing” in 44th International Conference on Parallel Processing p 150-159.
[4] Mittal S (2017). A survey of techniques for cache partitioning in multicore processors. ACM Computing Surveys (CSUR), 50(2), 27.
[5] Cadence Design Systems, Inc. (2014). Tensilica Software Development Toolkit (SDK), 2014 Release. Cadence Design Systems, Inc., Santa Clara California, USA.
[6] Tensilica Inc. (2010). Xtensa Modeling Protocol (XTMP) User’s Guide. Tensilica Inc., Santa Clara California, USA.
[7] Mahrom N, Liebelt M, Raoof R A A, Daud S and Ghazali N H (2018). “An Adaptive Insertion and Promotion Policy for Partitioned Shared Caches” in IOP Conference Series: Materials Science and Engineering 318(1), p 012048.
[8] Selfa V, Sahuquillo J, Gómez C and Gómez M. E (2015). “Methodologies and performance metrics to evaluate multiprogram workloads” in 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, p 150-154.
[9] Jain P and Surve S K (2019) “Resource Centric Characterization and Classification of Applications Using KMeans for Multicores” in 2019 International Conference on Information Networking (ICOIN) p 25-30.
[10] Qureshi M K and Patt Y N (2006) “Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches,” in Proc. of the 39th Annual IEEE/ACM International Symposium on Microarchitecture, p 423-432.
[11] Muralidhara S P, Kandemir M and Raghavan P (2010) “Intra-application cache partitioning,” in IEEE International Symposium on Parallel & Distributed Processing (IPDPS), p 1-12.