

Dynamic Classification of Program Memory Behaviors in CMPs

Yuejian Xie Gabriel H. Loh

Georgia Institute of Technology
College of Computing
{corvarx,loh}@cc.gatech.edu

Abstract

Multi-core processors with shared L2 caches can suffer from performance degradations when co-scheduled programs contend for cache resources in a destructive manner. In this work, we propose a new classification algorithm for determining the “personalities” of the programs with respect to their cache sharing behaviors. We first demonstrate that our scheme can more accurately predict when cache sharing problems may arise (and therefore when dynamic cache partitioning techniques are needed) than compared to other previously proposed approaches. This may be useful in the creation of better workloads for future multi-core shared-cache simulation studies. Furthermore, our proposed scheme can be implemented directly in hardware to provide dynamic, on-the-fly classification of program behaviors (other classification techniques require, for example, performance comparisons against solo-executions where a program uses the entire L2 cache which cannot be trivially derived in an online fashion). Using this dynamic classification ability, we propose a very simple dynamic cache partitioning scheme that performs slightly better than the Utility-based Cache Partitioning scheme while incurring a lower implementation cost.

1. Introduction

The simultaneous execution of multiple programs on a multi-core processor with a shared on-chip cache can result in cache interference that reduces system throughput and overall fairness and quality of service. Such situations may occur due to the presence of a program with a large memory footprint with frequent accesses, but low reuse characteristics which ends up evicting a large number of cache lines used by other cores. As a result, recent research efforts have reported on a large variety of static and dynamic cache management schemes to partition cache resources among the multiple cores [4, 7, 9].

Of particular interest is recent work by Moreto et al. that analyzes program characteristics to explain why some multi-programmed workloads benefit from cache partitioning while other workloads do not [6]. Using a few simple-to-describe metrics, they show that they can accurately classify benchmarks to predict when dynamic cache partitioning would be of value. Other studies have also proposed similar-in-spirit classification schemes for workload cre-

ation purposes [5, 7]. These classification techniques can be employed in a *post mortem* fashion to explain speedup phenomena *after the fact*. In our work, however, we present a new, simple classification technique that can be employed *in vivo* to dynamically monitor the instantaneous behavior of applications. On-the-fly classification of workload cache behaviors has many potential applications. For example, the classification information can be used to improve cache partitioning algorithms, or the information can be fed back to the operating system so that programs with conflicting/incompatible behaviors can be rescheduled to non-overlapping timeslots, or even to different chips (in a multi-socket system).

In the next two sections, we first review previously proposed classification techniques and describe why they cannot be employed for online dynamic workload monitoring. In Section 3, we then describe our proposed scheme. We present two case studies employing our technique. First in Section 4, we revisit the problem of explaining speedups in dynamic cache partitioning schemes and demonstrate that our approach can better predict when partitioning will be useful. Then in Section 5, we describe a very simple cache partitioning scheme that makes use of our online classification to perform a very simple form of dynamic partitioning that provides speedups with very little additional hardware overhead.

2. Existing Classification Schemes

As briefly mentioned in the introduction, several other works have presented a variety of approaches to classify programs’ cache behaviors in a multi-core context. This section presents a brief overview of some of these.

2.1. Lin et al.’s “Colors”

Lin et al.’s study aimed to duplicate past work on cache partitioning by using OS-level page coloring to directly allocate L2 cache resources between two programs on a real system (as opposed to in simulation) [5]. As part of this work, they classified programs into one of four classes to which they assigned colors. To perform this classification, they consider the performance degradation observed when running a program using only a 1MB L2 cache compared to the baseline configuration with 4MB. Any program with greater than 20% slowdown was classified as *Red* and

greater than 5% slowdown (but less than 20%) as *Yellow*. Out of the remaining programs with less than or equal to 5% slowdown, the program is classified as *Green* if the total number of L2 accesses is greater than or equal to 14 misses per thousand cycles, otherwise the program is in the *Black* category.

While Lin et al.’s color-based classification scheme may be useful for workload creation, it cannot be easily used for dynamic, on-the-fly classification of program behavior. In particular, computing the performance slowdown would require simultaneously running two copies of the program on two cores, each with their own dedicated L2 caches. One cache then further needs to be crippled to only use one-fourth of the cache capacity, and then finally performance between the two would be compared. To perform dynamic classification, the two copies would have to be synchronized to execute the exact same code, which means that the version with the full 4MB core will constantly be waiting for the slower 1MB version to catch up. If a separate core was available, then it would make more sense to simply schedule two programs on separate cores rather than co-scheduling them together and then use two additional cores to determine their respective stand-alone performances.

Overall, these constraints render Lin et al.’s classification approach completely impractical for *in vivo* cache-sharing behavior classification. To be fair, Lin et al.’s approach was not designed for dynamic scenarios; we make note that it is not trivial to simply “port” their approach, which motivates the need for a new scheme.

2.2. Moreto et al.’s Dual Metrics

Moreto et al. introduced two metrics to classify programs into three categories, which they use to explain when speedups can be obtained by cache partitioning [6]. The first metric, $w_{P\%}$, is the number of ways needed by a program to obtain at least $P\%$ of its maximum IPC (i.e., the IPC achieved if the program had sole usage of all n ways of the L2 cache). The second metric, $w_{LRU}(th_i)$, is the (average) number of ways used by each thread when running simultaneously (assuming LRU replacement). Moreto et al.’s results show that they can accurately predict when cache partitioning will be beneficial, but similar to Lin et al.’s scheme, the metrics cannot be easily monitored in an online fashion (i.e., $w_{P\%}$ requires comparison against the performance when using the full L2 cache, which in turn requires a complete redundant execution of the program).

2.3. Utility-based Classification

Qureshi and Patt studied the marginal utility of increasing set-associativity on programs [7]. In particular, they describe applications as belonging to one of three classes. *High-utility* programs exhibit continued performance improvements (or cache miss rate reductions) as the number of ways are increased. *Low-utility* programs do not gain

much benefit from allocating more cache ways. *Saturating-utility* programs show performance improvements with more cache ways, but only up to a point. Past this point, the allocation of additional ways does not significantly increase performance any further. Their categorization appears to be largely based on visual analysis of the per-program utility curves; they do not provide any formal, algorithmic means of classifying programs into these classes and as a result, it is not obvious how one would go about implementing a dynamic (in hardware) mechanism for classifying programs based on their criteria. Their application analysis also include CPI rates as the cache associativity is varied; if this information is required for classification, then it would be impractical to dynamically collect this information as it would require running the n concurrent copies of the program, each with a different number of ways allocated.

Qureshi and Patt also used a simple classification scheme for the purposes of generating interesting multi-core workloads [7]. In particular, for k concurrently running programs (sharing the L2 cache), the weighted speedup is equal to $\sum_{i=1}^k (IPC_i / SingleIPC_i)$, where IPC_i is the committed instructions-per-cycle (IPC) rate of program i when concurrently running with all other $k-1$ programs; $SingleIPC_i$ is the committed instructions-per-cycle rate when program i is running alone with full usage of the entire L2 cache. This is slightly different from Lin et al.’s classification as this groups the *workloads* rather than the individual benchmarks. Since this weighted-speedup-based classification requires comparing the performance of programs run separately and together, it is also nearly impossible to use as an online classification mechanism.

2.4. Chandra et al.’s Miss Models

Chandra et al. proposed three detailed models for predicting the impact of cache sharing among multiple threads [2]. The models provide an estimate of the number of additional L2 misses caused by sharing when compared to running a thread by itself. In particular, their Inductive Probability model predicts the number of such misses with an average error of only 4%. Unfortunately, the model is fairly involved; the large number of complex statistical computations would be very difficult to directly implement in hardware. While the model can in theory be used to classify programs’ behaviors in the context of L2 cache sharing, the granularity of the model output is much finer than is necessary for our purposes.

3. An “Animalistic” Taxonomy

In our proposed scheme, we classify benchmarks into intuitive “animal” personalities based on a few simple heuristic metrics. The main difference between our approach and the previously proposed schemes is that all of our metrics can be directly derived at runtime in hardware.

3.1. Our Four Animal Types

Turtles: There are some applications that simply do not make much use of the shared last-level (L2) cache. This may be because the program simply has very few memory instructions to begin with, or it may be that the program has a very small working set that completely fits within the level-one caches and therefore rarely accesses the L2 cache. We call these programs *turtles* as they are small and “slow moving” (with respect to frequency of L2 accesses).

Sheep: In a cache that employs dynamic cache partitioning, some applications simply are not sensitive to the number of ways allocated to them. These applications may actually exhibit a high rate of L2 accesses, but even with an allocation of only a few ways, these programs can achieve a low L2 miss rate. We call these programs *sheep* as they are gentle and tame, and not easily perturbed by other applications.

Rabbits: Some applications are very sensitive about the number of ways allocated to them. Such applications access the L2 cache fairly frequently, but if provided with a sufficient number of ways, the overall miss rate can be kept low. The performance can rapidly degrade if there is an insufficient cache allocation. We call these programs *rabbits* as they are “fast moving” (with respect to frequency of L2 accesses), delicate (in that they are more easily impacted by other programs) and tend to be happier with more space to run around in.

Tasmanian Devils: Our last class of applications simply do not “play well with others.” These applications access the L2 cache very frequently, but still have very high miss rates. As a result, such applications do not derive much benefit from occupying the cache (in terms of hit-rate reduction), and furthermore they tend to negatively impact other applications since the frequently missing accesses tend to kick out cache lines that are useful to other programs. We call such programs *Tasmanian Devils*¹, or just *devils* for short.

3.2. Dynamic Classification Algorithm

As discussed in Section 2, other previously proposed classification schemes require multi-configuration performance comparisons (e.g., speedup compared to the situation where the program runs alone with full ownership of the L2) and therefore are not amenable to dynamic, on-chip classification mechanisms. Our animal-based taxonomy has been designed to allow for on-the-fly monitoring of programs’ animal classes. We first review the Utility Monitor concept (which our classifier uses) and then describe the exact criteria used for determining a program’s animal type as well as implementation issues.

¹The Tasmanian Devil is an old Looney Tunes cartoon character with the primary traits of having a ravenous appetite and crazed behavior. Placed in almost any environment, he will devour and destroy everything.

3.2.1. The Utility Monitor

In the Utility-based Cache Partitioning work by Qureshi and Patt, they propose a simple mechanism for dynamically tracking the benefit or *utility* of allocating additional cache ways to a program, while keeping the overhead under control through set sampling [7]. This *Utility Monitor* (UMON) consists of $n+1$ counters per core for an n -way set associative cache. If a core actually had all n ways allocated for its sole use, a cache hit in the i^{th} position in the LRU stack causes the i^{th} UMON counter to be incremented, and the $n+1^{\text{st}}$ counter tracks misses. What each counter ends up representing is the number of additional hits that could be achieved if one more way were to be allocated to this program, and so the counters are sometimes called marginal gain counters. In concept, the UMON implementation requires that the L2 cache keep one set of shadow tags per core, but the set sampling technique greatly reduces this overhead by implementing shadow tags for only a subset of the cache sets and assuming that the behavior observed for these sampled sets are representative. Qureshi and Patt demonstrate that the difference between set sampling and implementing full sets of shadow tags is quite small. In the results presented in this paper, we simply use full shadow tags for implementation simplicity, with the understanding that set sampling can be employed to greatly reduce the monitoring overhead.

3.2.2. Our Animal Classification

We use a combination of dynamically collected information to determine the “animal” personalities of programs. We make use of the following metrics:

- *Accesses*: The total number of accesses to the L2 cache by the program. This includes instruction, data and prefetch requests.
- *Misses_{solo}*: The total number of L2 misses if the program had sole use of the entire n ways of the cache.
- *MissRate_{solo}*: The relative L2 miss rate if the program had sole use of the entire n ways of the cache ($Misses_{solo}/Accesses$).
- *WaysNeeded_{k%}*: The smallest number of ways needed to achieve a miss rate that is greater than or equal to $k\%$ of *MissRate_{solo}*.

The *Accesses* metric can be easily tracked by a single integer counter per core. The *Misses_{solo}* metric requires checking accesses against the shadow tags that track the simulated cache contents as if the program had sole access to the L2 cache. Apart from an integer counter to track the number of misses, this metric does not require any *additional* overhead since we are already paying the price of shadow tags for the utility monitor. At first glance, computing *MissRate_{solo}*

and $WaysNeeded_k$ seem to require expensive division operations that would make dynamic tracking of these metrics impractical. We will now describe the final classification criteria and then return to explain why division operations are not needed.

Based on the metrics defined above, we classify programs based on the following heuristic rules:

```

If ( $Accesses < 1,000$ )
  Animal := Turtle
Elsif ( $(MissRate_{solo} > 10\%)$  OR  $(Misses_{solo} > 4,000)$ )
  Animal := Devil
Elsif ( $WaysNeeded_{95\%} > \frac{n}{2}$ ) /*  $n = set\text{-}assoc.$  */
  Animal := Rabbit
Else
  Animal := Sheep

```

The intuition for these rules follows the qualitative descriptions of the “animals” given at the start of this section. If the application does not access the cache a lot, then it will be a turtle. If the application does access the cache a bit more frequently, then we examine its miss rate and its total misses. If the miss *rate* is high or the total (absolute) number of misses is extremely high, then we classify the program as a devil. The first criteria captures the situation where the program brings in a significant number of cache lines, but does not have a high reuse rate and so caching these lines is not very useful. The second criteria captures the situation where the program simply misses in the cache *very* frequently. In this case, regardless of whether the program has a high overall L2 hit rate, the sheer number of accesses will rapidly cause cachelines owned by other programs to get flushed out from the cache (i.e., they will rapidly be demoted in the global LRU stack and then evicted). In the remaining two cases, if the program needs a medium (or more) number of ways to attain a low number of misses (compared to $Misses_{solo}$), then we classify it as a rabbit. Finally, the sheep access the cache with some regularity, but even a few ways are sufficient to maintain decent hit rates.

The exact constants employed in our rules were chosen somewhat arbitrarily; we did experiment with a range of other values, but we found that the results and trends did not vary by any significant amount. Returning to the issue of implementation complexity, we first consider computing the condition $MissRate_{solo} > 10\%$. This is equivalent to computing $Misses_{solo}/Accesses > 10\%$, which can be rewritten as $Misses_{solo} > Accesses \times 0.1$. By choosing a slightly different threshold, say 12.5%, the multiplication by 0.1 can be replaced by a multiplication by 0.125 (same as division by eight) which can be trivially implemented as a right-shift by three positions. Next, for $WaysNeeded_{95\%}$, computing 95% of the hit rate would also involve an unattractive multiplication by 9 followed by division by 10.

By choosing a threshold such as 93.75%, which equals to $1 - \frac{1}{16}$, we can compute $WaysNeeded_{93.75\%}$ with a right-shift and a subtraction. For a practical implementation, several of the other constants could be replaced by the closest power-of-two, such as replacing 1000 by 1024 which simplifies testing for less-than to a NOR-operation of any bits of the counter left of the tenth least-significant bit. Since we found that the overall trends did not vary significantly with slightly different values of these parameters, we simply did not optimize them for implementability in this work. The main point of this discussion is to convince the reader that our classification scheme can be easily tweaked to make implementation easy.

The parameters were chosen for a sampling frequency of one million cycles. That is, we collect the metrics over an interval of one million cycles, and at the end of the interval, we make the decision of what type of an animal each program is acting like. We then reset all relevant counters and then re-evaluate program behavior after another one million cycles. If shorter or longer sampling intervals are desired, then the constants involved in the classification rules will need to be scaled appropriately. Similarly, with set sampling, the constants will also need adjustments proportional to the degree of sampling.

Note that during different phases, programs may exhibit different animalistic traits. We have observed that some programs act like devils during some phases, and then later may be more sheep-like. Similarly, programs (or program phases) may exhibit different degrees of animal traits. For example, one devilish program may satisfy the $Misses_{solo} > 4,000$ criteria by having 4001 such misses, which another program may have $Misses_{solo} = 200,000$. We informally refer to the former as a “small” devil, and the latter as a “large” devil. If a program consistently exhibits devilish behavior throughout its entire execution, then we also call it a “pure” devil. Similarly, due to the somewhat arbitrary criteria of $WaysNeeded_{95\%} > \frac{n}{2}$, there exists a continuum of behaviors between very “sheepish” (i.e., needing only a single way in the L2 cache) to very “rabbity” (i.e., needing all n ways to achieve a high hit rate).

4. Predicting Usefulness of Cache Partitioning

In our first case study, we demonstrate how to use our classification scheme to predict when cache partitioning will be useful. In particular, we use Lin et al.’s color-based classification as a point of comparison [5]. In particular, their results showed that cache partitioning provides the greatest benefits when a Red program is paired/co-scheduled with a Green program, and that the speedups are relatively small for all remaining color combinations. After first briefly explaining our simulation methodology, we will enumerate our benchmarks and show how they are classified under both Lin et al.’s and our own schemes. We will then

Fetch Width	16 bytes per cycle
Decode Width	4 insts (4-1-1-1 μ ops)
Issue Width	6 μ ops per cycle
Function Units	3 IALU, 1 IMul, 1 FAdd, 1 Div, 1 FMul, 1 Load, 1 STA, 1 STD
ROB Size	96 entries
RS Size	32 entries
LDQ/STQ Size	32/20 entries
Commit Width	4 μ ops per cycle
IL1/DL1	32KB, 8-way, 64-byte lines
Shared L2	4MB, 16-way, 64-byte lines
Main Memory	SDRAM, 800MHz bus (DDR), 9-9-9

Table 1. Baseline processor configuration.

show how Qureshi and Patt’s Utility-based Cache Partitioning (UCP) [7] performs for different workloads and demonstrate that our classification is more accurate at predicting when cache partitioning will be fruitful.

4.1. Simulation Methodology

For our simulations, we used the pre-release version of the SimpleScalar toolset for the x86 ISA [1], and we extended it to perform cycle-level modeling of a multi-core processor. The simulator accounts for the contention for cache/memory buses, finite MSHR capacity, a memory controller with request reordering [8], and DDR2 SDRAM timing. The simulated processor configuration is based on the Intel Core 2 Duo; the full details are listed in Table 1.

The current version of the x86 SimpleScalar toolset does not support multi-threaded programs, and as a result we simply make use of multi-programmed workloads. In particular, we use benchmarks from SPEC2006 from both the integer and floating point suites. We use reference inputs, and benchmarks with multiple inputs are distinguished by additional numerical suffixes (e.g., soplex.1, soplex.2). Due to incomplete system call emulation support in the x86-version of SimpleScalar, some applications currently cannot be simulated. Table 2 lists the applications and their baseline statistics. We use SimPoint 3.2 to select representative samples of each benchmark [3]. Prior to detailed cycle-level simulation, we warm the caches for 250 million instructions per application in a round-robin fashion. We then run the timing simulation for 500 million instructions per benchmark. We follow the methodology used by Qureshi and Patt where once a benchmark reaches its simulation instruction limit, we freeze the statistics for that application, but then continue simulating it so that it continues to compete for cache resources (otherwise, the other program would suddenly have access to the entire L2 cache) [7].

4.2. Classification

Table 2 lists each benchmark along with relevant statistics. The applications are sorted by Lin et al.’s color-based classification scheme. For the Red and Yellow classes, the programs are sorted by the performance slowdown (between

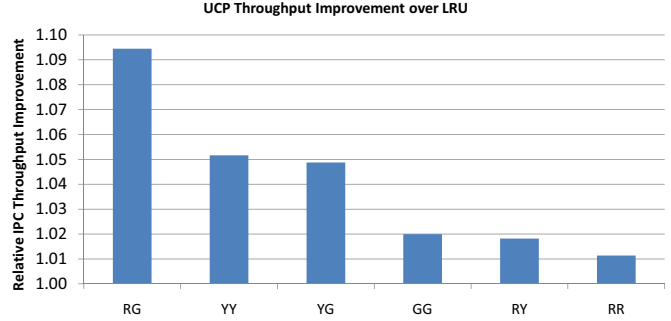


Figure 1. IPC throughput improvements for UCP compared to traditional LRU of different workload groups as determined by Lin et al.’s color-based classification.

running with 4MB and 1MB L2 caches), and the Green and Black applications are sorted by the L2 access rates (listed per 1000 instructions). These are the primary metrics used in their classification approach. For each benchmark, we also include a distribution of the time spent in each animal-class based on our proposed scheme. For example, bzip2.4 spends 23% of the time acting like a sheep, 65% of the time as a rabbit, and 12% of the time as a devil. The overall classification is based on which state the program spends the majority of its time in. There are no clear correlations between the two classification approaches; e.g., not all devils are in the red group and visa-versa.

4.3. Predicting Partitioning Speedups

We now compare the performance of Qureshi and Patt’s Utility-based Cache Partitioning (UCP) algorithm over a range of workloads.

4.3.1. UCP Grouped by Colors

We first present results for workloads formed based on Lin et al.’s color-based classification scheme. For each combination of colors (excluding Black which has so few L2 accesses to have any real impact on partitioning), we randomly formed several workloads (we evaluate many more workloads than, for example, Lin et al.’s study). In this work, we report IPC throughput (sum of the per-benchmark IPCs); other potential metrics include the weighted speedup and fair speedup, but we only stick to a single metric for brevity. Our results show that workloads combining Red with Green programs (RG) result in the highest performance improvements for UCP; this is in agreement with that reported by Lin et al.

Figure 2 shows the per-workload breakdown for UCP on the RG grouping. From these results, however, we see that the performance improvements are not consistent across the different application pairings. The Yellow-Yellow (YY) grouping also has a smaller overall performance improvement according to Figure 1, but the per-workload results in Figure 3 show that here, too, the improvements are in-

Benchmark Name	Base IPC	4MB/1MB Slowdown	L2 Access Rate (PKI)	% Time Spent as:				Overall Animal	Color Class
				Turtle	Sheep	Rabbit	Devil		
bzip2.4	1.35	48.8%	37.9	0.0%	23.6%	64.8%	11.7%	Rabbit	RED
soplex.1	0.34	34.1%	35.5	0.5%	50.1%	42.1%	7.3%	Sheep	RED
bzip2.1	1.62	31.9%	22.1	0.0%	39.8%	58.6%	1.6%	Rabbit	RED
mcf	0.16	29.8%	48.4	0.0%	32.5%	0.2%	67.3%	Devil	RED
omnetpp	0.54	24.1%	27.0	0.0%	0.8%	6.9%	92.4%	Devil	RED
bzip2.5	1.22	22.8%	16.9	0.5%	50.1%	42.1%	7.3%	Sheep	RED
bzip2.6	0.97	22.3%	19.4	1.2%	78.0%	20.4%	0.4%	Sheep	RED
bzip2.2	1.13	21.9%	16.9	0.0%	73.9%	25.9%	0.2%	Sheep	RED
bzip2.3	1.12	21.3%	20.3	0.0%	88.8%	11.2%	0.0%	Sheep	RED
hmmmer.1	1.02	20.4%	11.4	0.0%	37.6%	61.1%	1.2%	Rabbit	RED
h264ref.1	1.09	16.3%	10.7	2.2%	46.1%	51.7%	0.0%	Rabbit	YELLOW
perl.3	1.08	15.3%	12.9	0.0%	11.9%	48.7%	39.4%	Rabbit	YELLOW
astar.1	1.15	10.1%	16.4	0.0%	66.3%	30.0%	3.7%	Sheep	YELLOW
h264ref.2	0.81	10.0%	10.0	19.7%	79.3%	1.0%	0.0%	Sheep	YELLOW
hmmmer.2	1.01	8.7%	10.0	0.0%	99.4%	0.0%	0.6%	Sheep	YELLOW
astar.2	1.16	8.2%	9.8	0.0%	39.4%	60.6%	0.0%	Rabbit	YELLOW
h264ref.3	0.78	6.9%	14.5	18.5%	81.5%	0.0%	0.0%	Sheep	YELLOW
leslie3d	0.48	6.3%	27.8	0.0%	8.3%	0.0%	91.7%	Devil	YELLOW
cactusADM	0.81	5.7%	25.9	0.0%	52.7%	0.0%	52.7%	Sheep	YELLOW
soplex.2	0.31	5.1%	27.7	0.0%	0.0%	0.0%	100.0%	Devil	YELLOW
libquantum	0.42	0.0%	50.3	0.0%	0.0%	0.0%	100.0%	Devil	GREEN
go.1	0.78	2.8%	31.0	0.2%	99.4%	0.0%	0.5%	Sheep	GREEN
sphinx3	0.49	3.1%	30.0	0.0%	0.3%	0.0%	99.7%	Devil	GREEN
go.4	0.80	2.9%	28.2	0.0%	99.5%	0.3%	0.2%	Sheep	GREEN
gromacs	0.92	2.0%	27.7	0.0%	95.6%	0.4%	4.1%	Sheep	GREEN
perl.2	1.33	0.1%	27.5	0.0%	100.0%	0.0%	0.0%	Sheep	GREEN
go.2	0.83	2.7%	26.8	2.5%	96.8%	0.2%	0.5%	Sheep	GREEN
go.5	0.88	1.3%	25.3	0.9%	98.9%	0.2%	0.0%	Sheep	GREEN
milc	0.33	-0.2%	24.7	0.0%	0.0%	0.0%	100.0%	Devil	GREEN
go.3	0.69	2.0%	23.6	1.4%	80.2%	0.1%	18.3%	Sheep	GREEN
zeusmp	0.88	4.0%	21.1	0.0%	66.1%	0.0%	33.9%	Sheep	GREEN
lbm	0.22	0.0%	20.2	0.0%	0.0%	0.0%	100.0%	Devil	GREEN
gcc.2	0.63	0.7%	17.8	0.0%	5.3%	0.0%	94.7%	Devil	GREEN
gcc.3	1.27	0.1%	13.5	0.0%	100.0%	0.0%	0.0%	Sheep	BLACK
dealll	0.80	1.2%	11.0	46.1%	14.6%	1.1%	38.1%	Turtle	BLACK
perl.1	1.31	2.7%	9.1	46.6%	25.5%	2.4%	25.5%	Turtle	BLACK
gcc.1	1.52	-0.4%	8.7	0.0%	55.9%	0.0%	44.1%	Sheep	BLACK
sjeng	0.78	0.2%	3.9	0.0%	84.2%	0.0%	15.8%	Sheep	BLACK
namd	1.39	1.7%	1.5	88.6%	5.3%	0.0%	6.1%	Turtle	BLACK

Table 2. Benchmark classification.

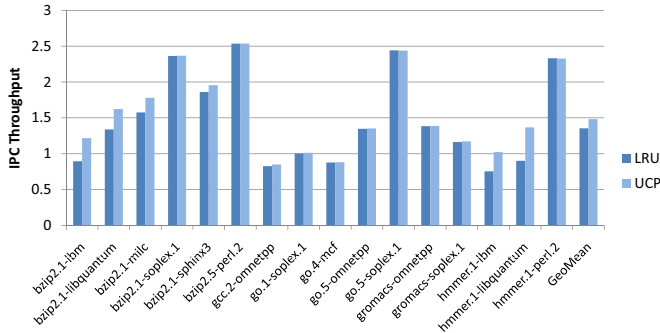


Figure 2. IPC throughput for the Red-Green workloads.

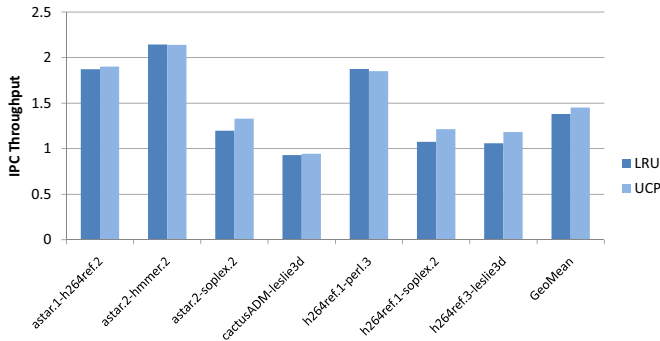


Figure 3. IPC throughput for the Yellow-Yellow workloads.

consistent. After presenting results based on our animal-classification scheme, we will return to these inconsistencies and explain them with our approach.

4.3.2. UCP Grouped by Animals

We then took all of these workloads, and regrouped them based on our animal classification scheme. Figure 4 shows the average throughput improvements across each set of animal pairings. The rabbit-devil combination yields the greatest improvements for UCP followed by devil-sheep. Figure 5 shows the individual throughput rates for each of the rabbit-devil workloads. So not only does the overall group average show strong performance gains, but the gains are quite consistent across all of the individual workloads as well. This makes sense, as the rabbits are all sensitive to how much effective cache capacity gets allocated to them, while the devils tend to thrash the cache. Applying dynamic cache partitioning in this scenario prevents the rabbit applications from getting hurt by the devils. The results for the devil-sheep workloads are similar (Figure 6), except that the magnitude of benefit is reduced. This is also consistent with the fact that sheep tend to need many fewer ways in the cache.

For the remaining workloads, UCP generally provides very little benefit. This makes perfect sense for any of the

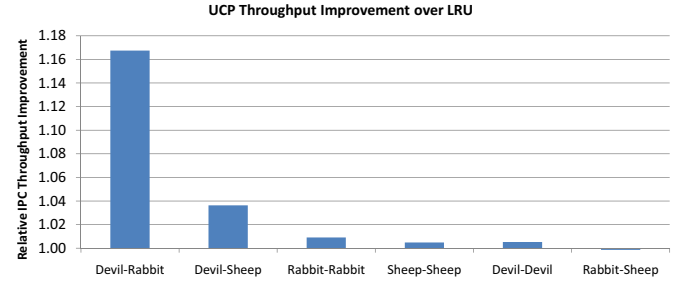


Figure 4. IPC throughput improvements for UCP compared to traditional LRU of different workload groups as determined by our animal-based classification.

workloads containing sheep (or turtles), as these applications are largely insensitive to how the other co-scheduled application makes use of the cache. For the devil-devil case, neither application makes good use of the cache, and so both will simply continue to miss in the L2. For the rabbit-rabbit, both programs need a significant number of ways in the cache to maintain relatively low miss rates. It is not to say that UCP does not do a good job at partitioning the cache resources between the applications; UCP by construction does partition the cache very well. The reason why the performance gains are not larger for these workloads is that the natural partitioning that occurs as a result of applying regular old LRU also works very well. This results from the applications having fairly-well matched access rates and miss rates.

4.3.3. Explaining the Exceptions

Returning to the anomalies observed in the color-based classification, we now use our scheme to explain these exceptions. For the red-green grouping (Figure 2), except for the bzip2.1-sphinx3 and gcc.2-omnetpp pairings, all other workloads that show low (or no) speedups involve at least one benchmark that we would classify as sheep. The sheep applications do not have much performance sensitivity to cache allocation, and therefore UCP does not have much impact. We actually classify bzip2.1-sphinx3 as a rabbit-devil pair, in which case we would expect a larger performance gain. The reason that the improvement is relatively modest (+5.1%) is that bzip2.1 is not a pure rabbit; it only exhibits rabbit behavior for 56% of the time. The last case of gcc.2-omnetpp is a devil-devil pair, for which partitioning also has very limited opportunities to make an impact. For all remaining workloads that did exhibit large performance improvements, our scheme successfully identified these as rabbit-devil pairs.

For the yellow-yellow groupings (Figure 3), the astar.2-soplex.2, h264ref.1-soplex.2 and h264ref.3-leslie3d workloads all show strong gains (+11%, +13% and +12%, respectively) despite being classified as YY. Our classification scheme groups the first two workloads as rabbit-devil

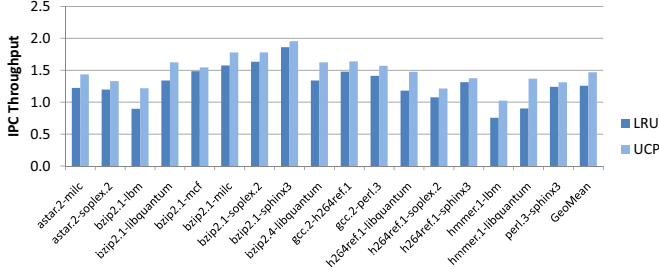


Figure 5. IPC throughput for the Rabbit-Devil workloads.

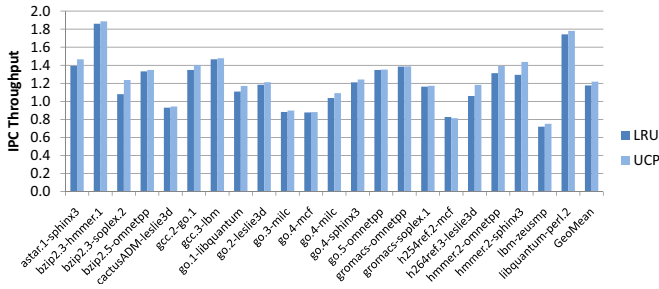


Figure 6. IPC throughput for the Sheep-Devil workloads.

pairings, for which we expect UCP to work well. The last workload is a sheep-devil pairing. For most sheep, an allocation of only 1-2 cache ways is sufficient to maintain high performance, while h264ref.3 is a “fatter” sheep that requires 3-4 ways.

From our results, we believe that our animal classification scheme provides a better means of determining the cache-sharing behaviors of workloads. At least for the workloads evaluated, we have shown that we can more accurately predict when cache partitioning will be beneficial. Our scheme can also be employed as a more meaningful and useful workload creation criteria for future multi-core shared cache studies. The fact that our scheme can be measured on-the-fly and provide phasic characterization can also help to better explain observed cache partitioning behavior. In the next section, we present another simple application of our dynamic classification for a very simple cache partitioning algorithm.

5. A Simple Replacement Policy to Guard Against Pathological Cache Sharing

From our animal classification scheme, we see that the main situations where dynamic cache partitioning is useful are when a devil-like application is present. Utility-based Cache Partitioning effectively does a good job at containing the devil application because UCP explicitly knows that allocating more ways to the devil provides very little benefit as measured by the marginal gains/utility. We observe that

it is sufficient to detect that an application is acting like a devil, and then to contain it. We now briefly describe our *Cache Partitioning by Catching Devils* (CPCD) policy for managing a shared cache in a multi-core processor.

For our CPCD approach, we only need to detect that an application is (currently acting like) a devil, and so our animal classification scheme can be simplified a little. In particular, the rules can be condensed to:

```

If ((Accesses ≥ 1,000) AND
    ((MissRatesolo > 10%) OR (Missessolo > 4,000)))
    Animal := Devil
Else
    Animal := Not-a-Devil

```

Since we no longer care about the exact criteria for recognizing rabbit behavior, we can also reduce the hardware requirements for our scheme. In particular, while we still need to maintain a set of shadow tags (with the same caveats as before regarding the use of set sampling to keep the overhead acceptable) to determine $Misses_{solo}$, we no longer need the per-core UMON marginal utility counters, as only the Rabbit classification makes use of the *WaysNeeded* metric. Given this online devil-detection, our cache management policy is simple. If no devils are present, simply use LRU. If a devil is present, limit it to only c ways; we say that the devil has been placed in a *cage* of size c . For our experiments, we used $c=4$, although we did not observe much difference in performance for cage sizes of three or five. Similar to our original classification scheme, we make the determination about whether a program is acting like a devil or not once every one million cycles, and then we simply assume it will continue acting in this way for the next million cycles. This is reasonable as program phases usually last for quite some time; this is consistent with Qureshi and Patt’s observations that UCP need not re-evaluate its partitioning decisions very often either [7].

Figure 7 shows the IPC throughput results for the baseline LRU, UCP, and our CPCD schemes. These results are for the rabbit-devil workloads. For almost all of the workloads, our CPCD scheme performs nearly as well, or even better than UCP. At first thought, it would seem that no partitioning scheme should do better than UCP since UCP explicitly computes the partition that results in the greatest reduction in L2 misses. Note, however, that our CPCD scheme is not a true partitioning approach. During the phases of execution where devil-like behavior does not exist, the cache management reverts to the original LRU policy where there are no set limits on how many ways a program can make use of. Overall, our results corroborate Qureshi and Patt’s earlier results that UCP does indeed work well for dynamic cache partitioning. From our CPCD results, we believe that when it comes to cache partitioning, paying attention to when programs are “misbehaving”

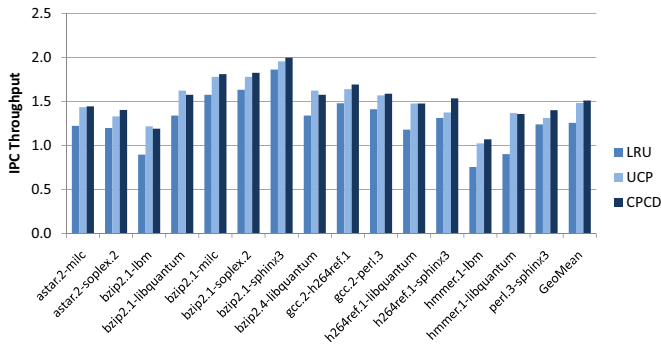


Figure 7. IPC throughput for LRU, UCP and CPCD on Rabbit-Devil workloads.

(i.e., are in devilish phases) can provide the majority of the most important information. This approach leads to a dynamic cache management scheme with less overhead than UCP while providing the effectively the same level of performance benefit.

6. Conclusions

The goal of this work was to introduce our new animal-based classification scheme and demonstrate its usefulness. In particular, we showed that we can use this approach to better identify cache-sharing behaviors and more accurately predict when cache partitioning will be useful. We also show how to use our classification scheme to build a multi-core shared-cache management policy that is simpler to implement than UCP while delivering the same (or slightly better) performance benefit. There may be other interesting applications of our animal-based classification scheme. For processors without some form of dynamic cache management/partitioning, one can provide feedback back to the operating system so that the process scheduler can attempt to avoid co-scheduling incompatible animal types (e.g., use a rabbit-sheep pairing followed by turtle-devil schedule rather than a rabbit-devil followed by turtle-sheep).

There are also many future avenues for research. It is not clear how well our animal classification scheme can predict the efficacy of cache partitioning for more than two cores. Similarly, we do not know how well the devil-catching (CPCD) approach works for systems with more cores and a greater variety of simultaneous animal types. Further work is also required to analyze the impact on fairness, as well as evaluating whether these ideas gracefully extend to handle multi-threaded applications. For multi-core systems with more cores, an effective cache partitioning scheme may need to explicitly take the programs' animal types into account. More research is required to refine our classification scheme. Our current algorithm uses a variety of thresholds/constants that have been empirically selected; more experiments are necessary to determine the robustness of the scheme.

Acknowledgments

Funding and equipment were provided by a grant from Intel Corporation. This material is based upon work supported by the National Science Foundation under Grant No. 0702275.

References

- [1] Todd Austin, Eric Larson, and Dan Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *IEEE Micro Magazine*, pages 59–67, February 2002.
- [2] Dhruba Chandra, Fei Guo, Seongbeom Kim, and Yan Solihin. Predicting Inter-Thread Cache Content on a Chip Multiprocessor Architecture. In *Proceedings of the 11th International Symposium on High Performance Computer Architecture*, pages 340–351, San Francisco, CA, USA, February 2005.
- [3] Greg Hamerly, Erez Perelman, Jeremy Lau, and Brad Calder. SimPoint 3.0: Faster and More Flexible Program Analysis. In *Proceedings of the Workshop on Modeling, Benchmarking and Simulation*, Madison, WI, USA, June 2005.
- [4] Seongbeom Kim, Dhruba Chandra, and Yan Solihin. Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 111–122, Antibes Juan-les-Pins, France, September 2004.
- [5] Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, and P. Sadayappan. Gaining Insights into Multicore Cache Partitioning: Bridging the Gap between Simulation and Real Systems. In *Proceedings of the 14th International Symposium on High Performance Computer Architecture*, pages 367–378, Salt Lake City, UT, USA, February 2008.
- [6] Miquel Moreto, Francisco J. Cazorla, Alex Ramirez, and Mateo Valero. Explaining Dynamic Cache Partitioning Speed Ups. *Computer Architecture Letters*, 6, 2007.
- [7] Moinuddin K. Qureshi and Yale N. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *Proceedings of the 39th International Symposium on Microarchitecture*, pages 423–432, Orlando, FL, December 2006.
- [8] Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter Mattson, and John D. Owens. Memory Access Scheduling. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 128–138, Vancouver, Canada, June 2000.
- [9] G. Edward Suh, Larry Rudolph, and Srinivas Devadas. Dynamic Partitioning of Shared Cache Memory. *Journal of Supercomputing*, 28(1):7–26, 2004.