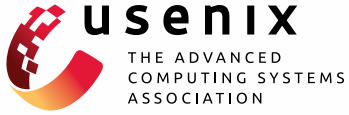


NightWatch: Integrating Lightweight and Transparent Cache Pollution Control into Dynamic Memory Allocation Systems

Rentong Guo, Xiaofei Liao, and Hai Jin, ***Huazhong University of Science and Technology;***
Jianhui Yue, ***Auburn University;*** **Guang Tan,** ***Chinese Academy of Sciences***

https://www.usenix.org/conference/atc15/technical-session/presentation/guo

This paper is included in the Proceedings of the
2015 USENIX Annual Technical Conference (USENIC ATC '15).

July 8–10, 2015 • Santa Clara, CA, USA

ISBN 978-1-931971-225

Open access to the Proceedings of the
2015 USENIX Annual Technical Conference
(USENIX ATC '15) is sponsored by USENIX.

NightWatch: Integrating Lightweight and Transparent Cache Pollution Control into Dynamic Memory Allocation Systems

Rentong Guo[†] Xiaofei Liao[†*] Hai Jin[†] Jianhui Yue[§] Guang Tan[‡]

[†] *Service Computing Technology and System Lab, Cluster and Grid Computing Lab,*
School of Computer Science and Technology, Huazhong University of Science and Technology, China

[§] *Auburn University, USA*

[‡] *SIAT, Chinese Academy of Sciences, China*

{*rtguo, xfliao, hjin*}*@hust.edu.cn jyue@auburn.edu guang.tan@siat.ac.cn*

Abstract

Cache pollution, by which weak-locality data unduly replaces strong-locality data, may notably degrade application performance in a shared-cache multicore machine. This paper presents NightWatch, a cache management subsystem that provides general, transparent and low-overhead pollution control to applications. NightWatch is based on the observation that data within the same memory chunk or chunks within the same allocation context often share similar locality property. NightWatch embodies this observation by online monitoring current cache locality to predict future behavior and restricting potential cache polluters proactively. We have integrated NightWatch into two popular allocators, *tcmalloc* and *ptmalloc2*. Experiments with SPEC CPU2006 show that NightWatch improves application performance by up to 45% (18% on average), with an average monitoring overhead of 0.57% (up to 3.02%).

1 Introduction

Modern multicore processors usually have a shared last level cache, where all cores place their data to improve cache utilization. This, however, creates a new challenge of cache management, due to cache pollution. One major problem is that data with weak locality may unduly evict other data with strong locality, if both are mapped into the same cache set [6, 12, 16, 18, 24, 30].

The major challenge to mitigate cache pollution is that the locality property of an application is implicitly determined by the runtime behavior. There has been much work [6, 16, 27, 28, 31] that has demonstrated the great potential of performance improvement via cache-aware memory allocation. However, they fall short in several aspects, such as requiring off-line analysis [3, 5, 16, 21, 23, 25, 26], special hardware support [7, 22, 27, 29, 31], or changing allocation interfaces [6, 28]. Hence, these techniques can hardly be used for general, unmodified applications.

This paper presents NightWatch, an online, transparent cache management subsystem for memory allocators. NightWatch dynamically characterizes the locality properties of allocated memory chunks, and provides hints to the memory allocator about the proper cache assignment for future allocation requests.

There are two main challenges to implement NightWatch efficiently. First, monitoring locality properties online is usually expensive and may easily cancel out the benefit from mitigated cache pollution. Second, it is hard to determine a priori whether a requested memory chunk will be a polluter before its actual use, due to the fact that the memory allocator has no knowledge about the application logic. NightWatch addresses the challenges by leveraging two new insights on the locality correlation among memory chunks[1]:

1. *Insight 1: Intra-chunk locality similarity*, where different pages in the same memory chunk tend to have similar locality properties;
2. *Insight 2: Inter-chunk locality similarity*, where different memory chunks in the same *allocation context*[2] tend to have similar locality properties.

Based on these insights, NigthWatch is built with mechanisms to monitor the access behavior of allocated memory chunks in a lightweight way, and to predict the behavior of new chunks with high reliability. Based on Insight 1, NightWatch infers the locality properties of a memory chunk by monitoring and sampling only a small part of each chunk. Based on Insight 2, NightWatch leverages the locality properties of previously allocated chunks to predict the locality properties of new chunks in the same allocation context.

We have integrated NightWatch into two popular memory allocators, *tcmalloc* [9] and *ptmalloc2* [10]. Specifically, NightWatch analyzes the historical locality

*Corresponding author

[1] A chunk is a memory block returned by malloc.

[2] The allocation context is identified as the call stack when the allocation function is called.

profile of allocated chunks and provides advices to the allocator on the proper cache assignment for the current allocation. In summary, we make three contributions:

- We demonstrate locality similarity within the same memory chunk, and between chunks in the same allocation context. These findings lay the foundation of practical online cache pollution control, which completely frees common programmers from cumbersome tasks of analyzing the program's cache demand.
- We present an open source implementation of NightWatch[3]. The cache management support is orthogonal to the traditional memory management techniques, and can be easily integrated into popular memory allocators [1, 8, 9, 10, 15].
- We have performed extensive evaluation of NightWatch with 27 programs from the SPEC CPU2006 benchmark suite. Compared with the popular allocator implementation of *tcmalloc*, NightWatch helps improve performance by up to 45%, with an average prediction accuracy of over 93%. NightWatch incurs very small extra overheads: the overhead is only 0.57% on average (up to 3.02%).

2 Motivation and Background

This section describes conventional ways of memory mapping, potential issues with cache pollution, and the concept of *restrictive mapping*.

2.1 Conventional Mapping

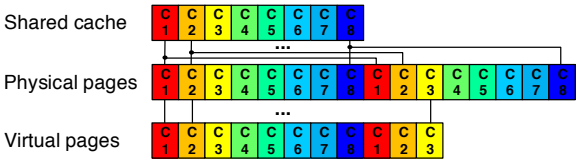
Since physical pages and cache sets are both physically indexed[4], the allocation of physical memory automatically determines the allocation of CPU caches. This relation is illustrated in Figure 1. A physical memory address is divided into two parts, i.e., page offset and physical page number. Several lower bits of the physical page number also serve as cache set index. The common bits divide cache sets and physical pages into different colors. Pages sharing a color are mapped to the same cache sets.

The memory allocator, as part of the runtime system, works at user level and is unaware of the mapping between cache and pages. The mapping is transparently handled by the *operating system* (OS). When the allocator acquires free memory, the OS returns memory with a unified mapping type. Conventional memory mapping techniques [20] attempt to maximize the number of cache sets assigned to consecutive virtual pages. As shown in Figure 2(a), physical pages in different colors are mapped to virtual pages in a round-robin manner. Such mapping allows even distribution of adjacent

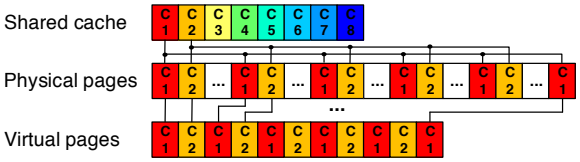
[3]https://github.com/grtoverflow/PC-Malloc

[4]CPU caches are commonly physically indexed [2, 11]. However, there are also some designs using virtual address as cache index. In this case, NightWatch's cache control mechanism will NOT work.

Figure 1: The relation between physical page number and cache set index



(a) Open mapping



(b) Restrictive mapping

Figure 2: Two types of memory mapping. Each bin in the shared cache represents a group of cache sets in the same color, and the bins in the following two rows are consecutive virtual pages and physical pages. The pages' labels distinguish the color.

data over the cache for the benefit of load balance between cache sets. The downside of this approach is that it allows weak-locality data to spread all over the cache, causing cache pollution to a maximum degree. Since there is no restriction on the pollution scope, we call this approach of memory mapping *open mapping*.

2.2 Issues with Conventional Mapping

The coexistence of both strong-locality and weak-locality data is very common. During execution, programs may have a large number of memory chunks with various locality properties.

Figure 3 shows the cache miss rate of the memory chunks of eight memory-intensive applications. It can be seen that the chunks differ in locality properties significantly. Take *dealII* as an example, 10% of its chunks have a miss rate below 10%, while 50% have a miss rate above 90%. In addition, the portion of the weak-locality chunks can be fairly large – two out of eight programs find a miss rate over 90% for more than half of their chunks. If not properly handled, the weak-locality chunks can overuse cache for little benefit, and leave little cache space to strong-locality chunks, thus degrading overall system performance.

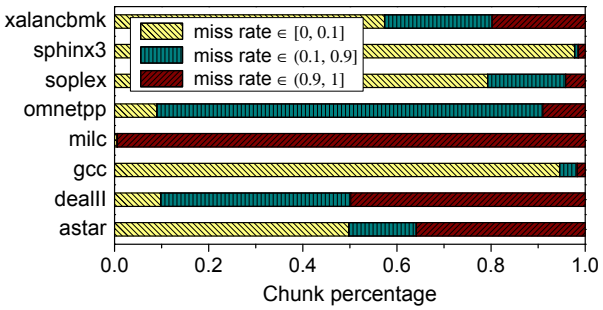


Figure 3: Chunk distributions of eight memory intensive programs. The cache size is 8MB.

2.3 Pollution Restrictive Mapping

A possible solution to the cache pollution issue would be to restrict the size of cache to which weak-locality chunks are mapped, and leaves more cache space to strong-locality ones. In this solution, page mapping is done in a restrictive way for polluters, and an open way for the rest of the chunks. Figure 2(b) depicts the principle of *restrictive mapping* for this solution. In the figure, the selection of physical pages are restricted in a limited *color region* (with two colors in this case) and mapped to consecutive virtual pages. Under such mapping, the weak-locality data are constrained and thus the pollution effect can be largely reduced.

While the idea of dual-mapping is conceptually straightforward, realizing it in an efficient way is non-trivial: the use of both open and restrictive mapping requires the memory allocator to be able to distinguish between polluter and normal chunks at runtime. As memory allocator is a core routine in most programs, such locality identification process needs to be performed in a lightweight manner. Otherwise, the monitoring overhead may easily nullify the benefit from improved cache locality.

3 Locality Similarity

Underlying our design are two observations of locality correlation between memory units. The first observation is concerned with locality similarity between pages within the same chunk, and the second on the locality similarity across different chunks within the same allocation context.

We use the SPEC CPU2006 benchmark suite to study the locality similarities. The benchmark consists of 27 programs[5]. We employ PIN [17] to collect the programs' memory allocation events and the full trace of data accesses. We then feed the data accesses to a cache simulator to track the cache miss rate of each memory chunk, and of each page within each chunk. The data

[5]Two programs, 434.zeusmp and 458.sjeng, are excluded, as they do not use dynamic memory allocation, and optimizing the cache allocation of stack and data segments are beyond the scope of this work.

```
img->mb_data
   = calloc(img->FrameSizeInMbs, sizeof(Macroblock));
......
/* encode a picture */
while (NumberOfCodedMBs < img->total_number_mb) {
   ......
   /* encode a macroblock in img->mb_data */
   encode_one_macroblock ();
   NumberOfCodedMBs++;
}
```

Figure 4: An example of intra-chunk access similarity

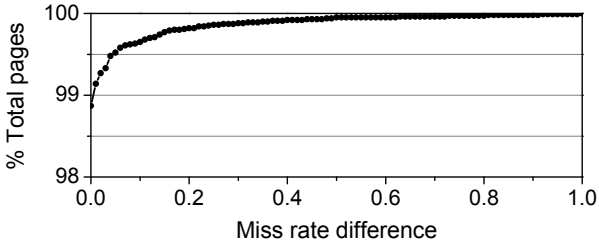


Figure 5: CDF of cache miss rate difference between pages and chunks

accesses are first filtered by a 256KB private cache, and then tracked in an 8MB set-associative shared cache. All the caches are configured with the LRU replacement policy, with 64B cache line size.

3.1 Intra-Chunk Locality Similarity

As discussed in Section 2, page is the basic unit for a memory allocator to manipulate a chunk's cache resource allocation. A chunk commonly consists of multiple pages, but managing cache allocation on a per-page basis is costly – that needs online monitoring for every page throughout the chunk's life cycle. Such an approach is necessary only if pages differ significantly in their locality properties.

Figure 4 shows an example of locality similarity within a single memory chunk, which is taken from *h264ref*, an implementation of the H.264/AVC (*Advanced Video Coding*) standard. The memory chunk, *img->mb_data*, is used to hold a whole frame of data during encoding. For an input video with 512x320 resolution, *img->mb_data* contains around 100 4KB pages. The frame is divided into macroblocks, each with 632B in size. In the main encoding iterations, each of the macroblocks is processed by *encode_one_macroblock ()*, with identical intra-frame and inter-frame compression algorithms. As a consequence, all the pages within *img->mb_data* will share similar locality properties. This implies that manipulating cache allocation for *img->mb_data* on a per page basis is unnecessary, since we can take the whole chunk as a basic cache allocation unit.

To verify the generality of the intra-chunk locality similarity, we examine all the chunks comprising more

```
for (img->number=0; img->number < input->no_frames;
      img->number++) {
   ......
   buf = malloc (xs * ys * symbol_size_in_bytes);
   /* read one frame */
   read(p_in, buf, bytes_y);
   /* convert file read buffer to source picture structure */
   buf2img(imgY_org_frm, buf, xs, ys, symbol_size_in_bytes);
   ......
   free (buf);
}
```

Figure 6: An example of inter-chunk access similarity

than one page in the 27 programs. We record the difference between each page's miss rate and its chunk's. The *cumulative distribution function* (CDF) of the difference is illustrated in Figure 5. The figure shows that most of the pages share a very similar miss rate with their chunks. Specifically, more than 98% of the pages have an identical miss rate as their chunks', and less than 0.5% of the pages have a miss rate difference from their chunks by 10%. This suggests that it is possible to manage cache allocation on a per chunk basis. More importantly, it is safe to reduce the number of monitored pages for lower overhead with little sacrifice of monitoring quality.

3.2 Inter-Chunk Locality Similarity

The locality monitor works only for allocated data. For a new memory request, the allocator has to perform cache mapping in a default way. If the monitoring results later on suggest that the default mapping does not fit the data's cache behavior, then a mapping switch, or remapping, is needed. This operation is prohibitively expensive: in the OS, data on the original pages needs to be copied to the new pages with proper physical indexing, followed by the update to the corresponding page table entries. In our experiments, for example, the time of remapping 1MB of data is as long as 1.8ms. Worse still, the effort may turn out not worthwhile, as many chunks are short-lived. For example, for the 27 programs we tested, over 90% of the chunks have a lifetime less than one second. After the monitoring and remapping phases, the cache allocation adjustment is too late to take effect, bringing very limited benefit compared with the cost.

To avoid cache remapping for a chunk, the allocator should make the initial mapping match the chunk's locality property, which calls for a reliable prediction of the chunk's access behavior. Fortunately, we find that the *allocation context*, defined as the call stack of an allocation request, provides important hints of the future behavior of to-be-allocated chunks, due to inter-chunk locality similarity.

Within an allocation context, a program commonly triggers memory allocation more than once. From Figure 7, we can see that for the 27 programs we tested, 99% of the chunk allocations fall in contexts that contain more

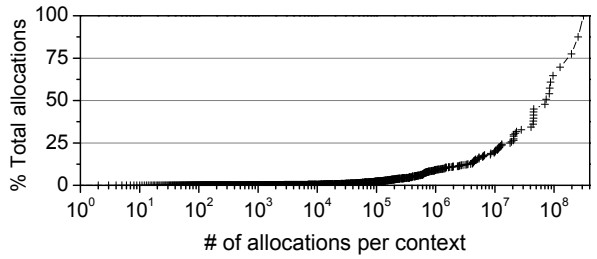


Figure 7: CDF of number of allocations

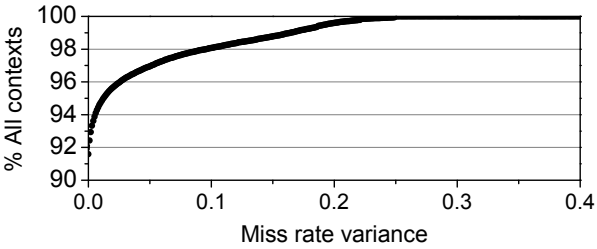


Figure 8: CDF of cache miss rate variance of the chunks sharing the same context

than 100 times of allocations. This indicates that most of the chunks are generated in "big" contexts, where they can find sufficient previously allocated chunks for locality prediction.

Figure 6 illustrates an example of chunk allocations in an allocation context, which is taken from *h264ref*. In the example, a group of memory chunks named *buf* are involved. Each of them lives in one encoding iteration: when a frame encoding iteration begins, *buf* is allocated to load one frame of data from the input file, then the data is converted to source picture format. After that, the memory chunk is freed back to the memory allocator. All the *buf* share the same allocation context, because the call stacks of *malloc()* in each encoding iteration are identical. Furthermore, these chunks also exhibit similar data access patterns – each of them serves one round of data installation and data conversion. As such, it is possible to use previously allocated chunks for locality prediction in later memory allocations.

In addition, the inter-chunk locality similarity also provides opportunities for reducing monitoring overhead. For contexts with good locality similarity, only a small part of the chunks need to be monitored to maintain a high prediction success rate.

To confirm the inter-chunk locality similarity in the same context, we calculate the variance of miss rate of chunks in each context. The CDF of miss rate variance is shown in Figure 8. Over 90% of the chunks share an identical miss rate with other chunks in the same context; less than 2% of the contexts have a miss rate variance greater than 0.1. As we will show later, this high level of locality similarity among the chunks leads to a high prediction success rate of 95.5% on average.

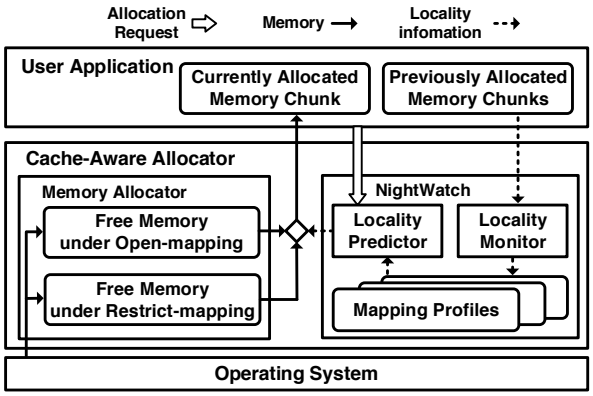


Figure 9: System overview of a NightWatch assisted allocator

4 Overview

NightWatch leverages the two observations of locality similarity, which inspire the design of efficient monitoring mechanisms, and help to make correct locality predictions. The system framework is shown in Figure 9.

The *locality monitor* collects locality information from previously allocated chunks. It periodically samples the references from the target chunks, and evaluates the chunk's locality property, which is sent to the *locality predictor*. Based on the historical locality information, the locality predictor determines the proper mapping for pending allocation requests. When a new request arrives, the predictor first checks its allocation context, and uses its predecessor chunks' locality profiles to predict the pending chunk's locality property. Then, the predictor notifies the memory allocator to perform the allocation.

The *memory allocator* is an extended allocator (i.e. extended from *tcmalloc*, *ptmalloc*, *ssmalloc,* etc). In most cases, a traditional allocator is unaware of cache pollution, and serves allocation requests with unique mapping (commonly the open mapping). To work with NightWatch, the allocator needs to be slightly modified to support both *open mapping* and *restrictive mapping*. The original allocator includes a set of data structures to maintain memory, for example, size classes used for containing small chunks [4, 9], and thread-local cache designed to manage per thread allocations [10, 15]. We extend the allocator to use two sets of these data structures, with each set maintaining memory under one of the two mapping types. The modified allocator uses an internal interface to serve allocation with both memory and cache demand, for instance, *malloc(mem_size, cache_map)*, where the *mem_size* is the user program's memory requirement, and the *cache_map* comes from the NightWatch's advice for cache mapping. The design of NigthWatch is general and portable so that it is quite convenient to extend the original allocator to support NightWatch. For example, it takes less than 700 lines of code modification to integrate NightWatch into *tcmalloc*, and 500+ lines of code modification for *ptmalloc2*.

The *operating system* manages the memory mappings. We modify the Linux kernel to support both open mapping and restrictive mapping, as discussed in Section 2. When the memory allocator runs out of certain type of memory, it acquires more memory via system calls. In our design, we extend the *mmap* system call as the interface to return free memory with multiple mappings.

5 Design and Implementation

In this section, we describe in detail the design and implementation of NightWatch's main components.

5.1 The Locality Monitor

Types of chunks. The locality monitor aims to evaluate a chunk's locality property by sampling its data accesses, and then classifies the chunk into two types: *polluter chunk* and *normal chunk*.

A polluter chunk is one with poor locality and caching it brings little performance gain, and therefore should be mapped to cache in a restrictive way. In practice, only a small fraction of the polluter chunks are completely non-temporally accessed: many of them still receive burst temporal accesses. As long as the burst accesses can fit in a cache region provided by the restrictive mapping, the chunks are treated as polluters. The remaining chunks are normal chunks. Data of normal chunks can get timely reuse before getting evicted from the cache. They are the normal users of the cache, and the potential victims of the polluter chunks. If normal chunks are identified, NightWatch will suggest to allocate them with open mapping.

Monitoring and mapping. The monitor samples a subset of the pages in each chunk periodically (i.e., every five seconds in our implementation). For each sampled page, the monitor records a hit or miss according to its access events and obtains a miss rate for the chunk. To obtain a stable result, multiple rounds of sampling are performed within each *sample collection* period, until the miss rate converges. Here, the condition of convergence is that the recent rate differs from the historical average rate by at most a given margin (0.1 in our case).

After the sample collection phase, the current miss rate of the chunk is generated. The miss rate is used to determine the *chunk type* and the proper mapping type, called the *target mapping type*. The current mapping type of the chunk may not match the target mapping type, and calls for a mapping switch. NightWatch does not notify the allocator to switch the mapping immediately, as many chunks are short-lived and may exhibit short-term locality variation; instead it waits until the next monitoring phase and checks the situation again. If the mismatch persists, it starts mapping switch.

In practice, a chunk's access pattern may change over time and remain unstable in the long term. In this case,

NightWatch does not keep switching between the mapping types, as remapping is expensive. There are two possible ways in which a chunk's role starts alternating: polluter→normal→ ... and normal→ ..... In the first case, NightWatch performs remapping only once, that is, from restrictive to open mapping, and stops monitoring the chunk afterwards. In the second case, NightWatch only performs the first open mapping and ignores future changes. The principle of mapping switch is that NightWatch would rather treat an unstable or primarily polluter chunk as a normal one, than restrict a normal one to its disadvantages. The consequence of this conservative mapping policy is that, for these particular cases, NightWatch degenerates the cache-aware allocator to a traditional memory allocator, and does not perform worse than a cache pollution unaware allocator. The conservative policy also has downsides. For example, for chunks with infrequent changing locality properties, the benefit from cache control may exceed the remapping overhead. One possible approach is to use methods similar to Branch History Table to detect phases of stable chunks and to make more aggressive cache control.

For small chunks below one page, NightWatch performs page alignment before monitoring. When a small chunk is selected as a sample, NightWatch will force the allocator to align the chunk to page size, so that the locality information monitored at page granularity can still represent the sampled chunk. The alignment will not cause much space waste, as NightWatch only needs to sample a very small percentage of the allocations.

Identification of access misses. To evaluate whether a sampled page encounters an access hit or miss, the monitor records a pair of successive references for the page, and estimates whether the second reference is timely enough to hit the CPU cache. There are two issues to be considered here. First, due to the nature of access locality, the reference events to a certain page tend to be clustered. As a result, a simple sampling procedure may find most of the collected reference pairs falling into the page's burst access interval. For mapping selection, such samples are meaningless, because they are frequent enough to hit the cache in any of the mapping types.

The second issue is the locality measurement. Access locality is commonly measured by *reuse distance*, which is defined as the number of distinct data accessed since the last access to the sampled data. Reuse distance is the same as LRU stack distance [19]. Although reuse distance provides a basis for precise cache miss prediction, it is very costly to be used directly for on-line monitoring – measuring reuse distance requires tracing the full data references. At present, no commodity hardware supports such measurement, and there is no efficient software approach either.

In NightWatch, we exploit the relation between cache events and reuse distance to address the two issues. Let Δn and Δm be the number of cache accesses and the number of cache misses between a reference pair, respectively. Also, let $C_{restrict}$ and C_{open} be the cache spaces (at cache line granularity) assigned by restrictive mapping and open mapping, respectively. Then we have

1. If $\Delta n < C_{restrict}$, the data reuse is frequent enough to fit either of the mapping types;
2. If $\Delta m > C_{open}$, the data will be evicted from cache before its reuse even with open mapping.

Note that the reuse distance measures distinct data accesses on the cache, and Δn is an access measurement without the *distinct* condition. Hence, Δn is an upper bound for the data's reuse distance. If Δn is less than $C_{restrict}$, the reuse distance will not exceed $C_{restrict}$. On the other hand, if Δm is larger than C_{open}, the distinct data installed on the cache is beyond C_{open}. Thus the reuse distance is already larger than C_{open} before the data reuse. So, even with open mapping, the data reuse will still trigger a cache miss.

The sampling process is as follows. In each monitoring cycle, a sampled page will be set with read/write protection for trapping page references. When the first reference arrives, the protection will be removed until the cache access volume reaches C_{open}, so that meaningless burst accesses can be skipped. Then, the monitor records the current number of cache misses, and waits for the second reference. When the second reference is trapped, the monitor checks the increment of cache misses Δm, and compares Δm and C_{open} to estimate whether the second page reference misses the cache or not.

Effectiveness of sampling. To evaluate the effectiveness of our sampling mechanism, we evaluate SPEC CPU 2006 benchmark suite. There are 1 million data access randomly sampled from each of the programs. For each of the sample, we compare our estimation of cache miss with the precise result given by off-line reuse distance analysis. Figure 10 shows an average false rate of

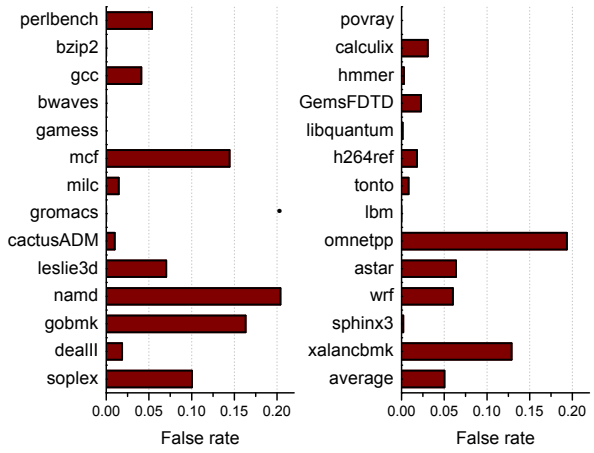


Figure 10: Cache miss estimation false rate

6.0%. Six out of the 27 programs have a false rate over 10% (*mcf, namd, gobmk, soplex, omnetpp, xalancbmk*). Compared with the reuse distance analysis, our method is much more lightweight and thus practical. As we will show later, it is accurate enough for the purpose of cache management. More importantly, since $\Delta m > C_{open}$ is a sufficient condition for cache miss events, our monitor can only take a miss event for a hit mistakenly, and may further regard a polluter chunk as a normal one, which is in line with our conservative mapping principle.

Minimizing monitoring overhead The optimization of monitoring overhead is carried out at two levels: 1) Page level. Due to high locality similarity between pages within a chunk, page sampling rate does not need to be high to allow accurate locality estimation. As we will show later in Section 6.3, for an estimation error below 5%, the required number of sampled pages is approximately $s^{0.65}$, where s is the total number of pages in a chunk. This sublinear relation means that the sampling method can scale to large chunks. 2) Chunk level. NightWatch tries to skip chunks in the request stream when it finds a chunk's prediction type to match the monitoring result. Specifically, upon each successful prediction, NightWatch doubles the sampling interval for reduced overhead. On the other hand, when a prediction is found to be false, the sampling interval falls back to zero. Based on inter-chunk locality similarity, this mechanism dramatically reduces chunk sampling rate, while guaranteeing a high prediction success rate.

5.2 The Locality Predictor

As we have described, once the locality monitor detects that a chunk's current mapping type does not match its actual locality pattern, the monitor will look for chances of mapping switch. The mapping switch mainly targets the first few chunks in their allocation context. For subsequent chunks in the context, their locality information can be predicted from the previously allocated chunks and thus mapping switch becomes much less necessary.

The prediction process is as follows. The predictor first analyzes the call stack of the allocation request to determine its allocation context. Then, it checks the mapping type of its two preceding chunks to make a prediction. If the current allocation request is the first one of its context, or the monitor has not yet determined the chunk type of its predecessors, the predictor assigns open mapping to the current request. If one of the predecessors is a normal chunk, the open mapping will also be applied. For other cases, restrictive mapping will be applied.

Notice the conservative policy of mapping that the predictor performs. Whenever there is inconsistency, the predictor chooses open mapping for the allocation request, for the same reason of the locality monitor's bias toward normal chunks in chunk type determination.

Accelerating locality prediction. NightWatch uses call stack as the identifier of an allocation context. We trace the 10-depth call stack of the current allocation function, and hash together all the program counters in every stack frame, so that the allocation context of a chunk can be determined by the hash value.

Some programs, especially those programmed with object-oriented languages, may extensively use small chunks. For those small chunks, we use a size-to-mapping table to provide a faster way to give the mapping type predictions. In our design, the table contains 64K entries, each corresponding to a set of chunks whose size equals the entry's index. The monitor's results are used to determine whether to invalidate a table entry or not. If the chunks in a table entry have the same mapping type, NightWatch directly returns a mapping type prediction for the coming allocation requests; otherwise, if the locality monitor detects inconsistency in this respect later on, the corresponding entry will be invalidated, in which case NightWatch will resort to the call stack tracing approach to find the allocation context and make prediction.

6 Evaluation

6.1 Experiment Setup

We conducted our experiments on a machine with four 2.13GHz quad-core Intel Xeon E7420 processors. The four cores in each of the processor share a set-associative 8MB cache, with 16,384 sets in total. Both stride prefetching and adjacent-line prefetching are enabled on the processors. The cache contains 256 colors; open mapping can use all the colors, while restrictive mapping can use only 32 colors. The server has 16GB memory, with eight 2GB fully buffered DIMMs. The operating system is CentOS 6.0, with Linux kernel 2.6.32-71.

We compare the NightWatch assisted *tcmalloc* (hereinafter *nw_tcmalloc* for short) with the original *tcmalloc* implementation. The *tcmalloc* used in our experiment is gperftools-2.4. The benchmark set consists of 27 programs of the SPEC CPU2006 benchmark suite. These programs are compiled with gcc 4.4.4.

6.2 Performance Improvement

We classify the 27 programs into three categories: polluters, victims, and neutral, based on two metrics: cache sensitivity and cache access rate. Cache sensitivity is defined as $T_{open}/T_{restrict}$, where T_{open} and $T_{restrict}$ are the program's execution times under open mapping only and restrictive mapping only, respectively. The cache access rate is the number of cache accesses per 1K cycles. The polluter programs have $cache\_sensitivity < 10\%$ and $cache\_access\_rate > 5\%$. Their data can hardly get timely reuse after installed into the cache. The second category is victim programs, with $cache\_sensitivity >$

Category	cache sensitivity	cache access rate (access per 1k cycle)	Benchmark
Polluter	$< 10\%$	> 5	410.bwaves 433.milc 459.GemsFDTD 462.libquantum 481.wrf
Victim	$> 20\%$	—	401.bzip2 403.gcc 429.mcf 447.dealII 450.soplex 470.lbm 471.omnetpp 473.astar 482.sphinx3 483.xalancbmk
Neutral	$[10\%, 20\%]$	< 5	400.perlbench 416.gamess 435.gromacs 436.cactusADM 437.leslie3d 444.namd 445.gobmk 453.povray 454.calculix 456.hmmer 464.h264ref 465.tonto

Table 1: Categories of benchmark programs. Cache sensitivity reflects the slowdown of program execution under pure restrictive mapping compared with under conventional (open) mapping.

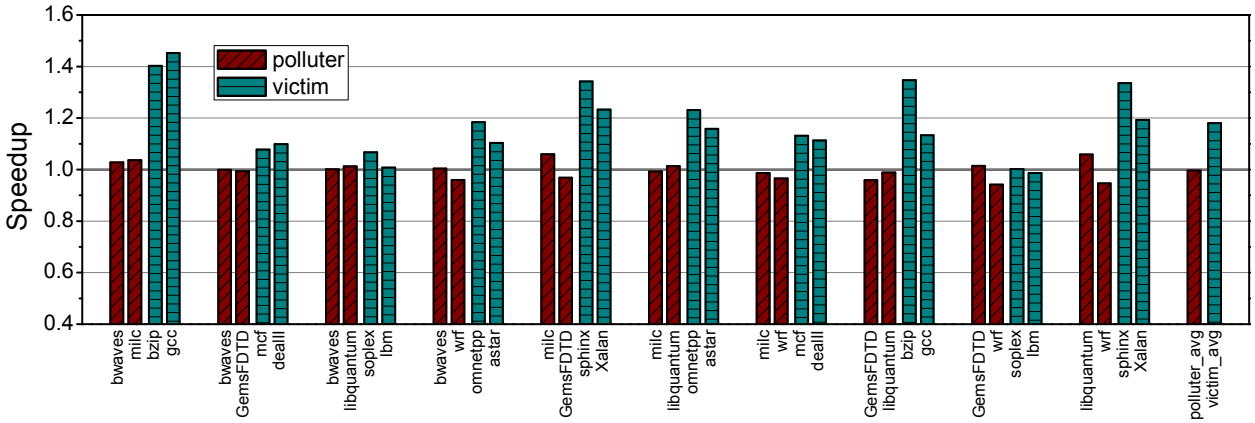


Figure 11: Performance speedup of polluter-victim combinations

20%, meaning that they have a high demand on cache, and are vulnerable to cache pollution. The remaining programs fall in the neutral category, which has limited demand on cache. They neither pollute cache, nor get polluted. The program classification is listed in Table 1.

Each workload used in our performance evaluation is a unique combination of four programs selected from two categories. Programs from different categories are combined only once to increase variety. There is one quad-core processor used in our experiments. We bind the programs to the four cores on this processor, so as to avoid the overhead of task migration. In addition, to make sure every program has three co-runners throughout the execution, we restart the early terminated programs until the longest one is completed, following the approach in previous work [14, 22].

In our experiment, we mainly evaluate the *polluter-victim* combination, in order to highlight the impact of polluters. The result is illustrated in Figure 11. As we can see, most of the victim programs can benefit substantially from NightWatch, with an average speedup of 1.18. In the combination $(bwaves, milc, bzip, gcc)$, for example, *gcc* achieves the highest speedup of 1.45. On the other hand, NightWatch has little impact on the polluter programs' performance in general. Compared with *tcmalloc* – which uses open mapping only – *nw_tcmalloc*'s dual-mapping scheme imposes little side effect on the polluters, due to two reasons. First, by leveraging intra- and inter-chunk locality similarities, *nw_tcmalloc* incurs very low overhead. As we will show later, for most of the programs, the overhead is less than 1%. The second reason is that *nw_tcmalloc* is able to distinguish between polluter and normal chunks and map them to cache in different ways. Thus, it does not harm the performance of the normal chunks used by the polluter programs.

It is worth noting that instead of staying unaffected or showing slight slowdown, several polluter programs, for example *bwaves* and *milc*, get noticeable speedup from *nw_tcmalloc*. This is somewhat counterintuitive, since for these programs, almost all the chunks are polluters. After applying restrictive mapping, the available cache space assigned to these programs is reduced from 8MB (under open mapping) to 1MB. We have run the two programs separately on two cores using *nw_tcmalloc* in comparison with *tcmalloc*, and observed speedups of 0.996 and 0.984 for *bwaves* and *milc*, respectively, which confirms the negative effect of reduced cache resource. Yet, when run with the victim programs *bzip* and *gcc*, the speedups surprisingly exceed one. A similar phenomenon was also observed by Lin et al. [14]. The reason behind this phenomenon is as follows: once the polluter chunks get restricted in cache usage, the normal chunks will get more cache space and thus their cache miss rate will be reduced. This results in a reduction of memory bandwidth pressure, and a smaller queuing delay at the memory controller. For example, for the first workload we tested, when *nw_tcmalloc* is applied, the overall cache miss rate is reduced by 4%. This reduction

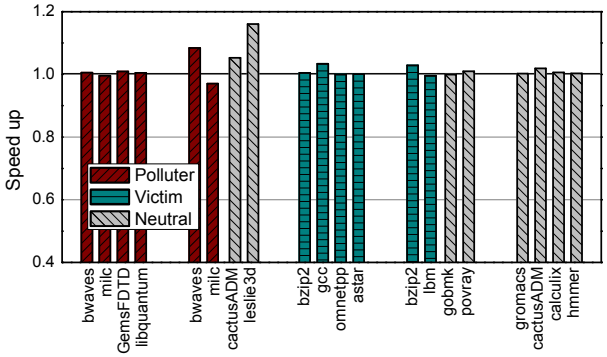


Figure 12: Performance speedup of five program combinations

turns out to be beneficial to the overall system performance, positively affecting *bwaves* and *milc* as well.

Figure 12 shows the performance of other combinations of benchmark programs, including all-polluter, all-victim, all-neutral, polluter-neutral, and victim-neutral. Overall, the programs experience very small changes in their performance, agreeing with the nature of these combinations, and suggesting that *nw_tcmalloc* retains system performance when it cannot bring improvement.

6.3 Efficiency of Locality Monitor

The locality monitor needs to collect access information of a random subset of pages in a chunk to determine the locality property of the chunk. The number of pages sampled, or sampling count, reflects a trade-off between sampling overhead and accuracy of locality estimation. Given a target upper bound of estimation error (e.g., 5%), we want to find the minimum sampling count, *MSC*, as a function of chunk size (in number of pages). We collect the page access statistics of all the programs' chunks, and for each chunk size, we experiment with increasing sampling count until the estimation error rate drops below 5%. For clarity purpose, we divide the chunk sizes into intervals, and calculate the average *MSC* for chunk sizes within each interval. Figure 13 shows the sampling count against chunk size in dot line. The measured curve can be roughly approximated by a straight line with slope 0.65, which translates to a sub-linear sampling count function $MSR(s) = s^{0.65}$, where s is the chunk size. This means the required sampling overhead grows much slower than a linear function, and thus can scale to large chunks while ensuring good accuracy of locality estimation.

6.4 Accuracy of Locality Predictor

For backward compatibility, *nw_tcmalloc* adopts the standard allocation interfaces. Since these interfaces only allow the programmer to specify the request's memory demand, *nw_tcmalloc* has to rely on NightWatch's locality predictor to infer the implicit cache demand. Therefore, a high prediction success rate is crucial to the

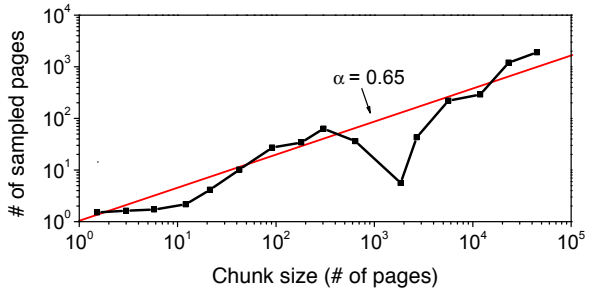


Figure 13: Number of sampled pages vs. chunk size

Benchmark	Succ. rate	Benchmark	Succ. rate
perlbench	99.8%	povray	99.7%
bzip2	95.8%	calculix	92.9%
gcc	96.2%	hmmer	99.5%
bwaves	99.3%	GemsFDTD	100.0%
gamess	98.6%	libquantum	77.8%
milc	99.7%	h264ref	92.2%
gromacs	100.0%	tonto	100.0%
cactusADM	100.0%	omnetpp	100.0%
leslie3d	100.0%	astar	97.5%
namd	58.1%	wrf	99.4%
gobmk	92.8%	sphinx3	100.0%
dealII	93.6%	xalancbmk	100.0%
soplex	94.3%	**Average**	**95.5%**

Table 2: Mapping type prediction success rate

efficiency of *nw_tcmalloc*.

If a chunk's predicted mapping type matches its actual locality property, the prediction is considered a success. Since NightWatch does not monitor every chunk, we collect prediction results and monitoring results in different runs in order to calculate prediction success rate. In the first run, the prediction results are collected in standard system configurations. In the second run, we force NightWatch to monitor every chunk throughout the benchmark's execution, and use the monitoring results to evaluate the predictor's accuracy. There are two programs (*mcf* and *lbm*) with no prediction information, because they do not provide any prediction opportunities. For example, *lbm* allocates two 214400KB chunks, each with an exclusive allocation context. Since there is no chunks previously allocated in the contexts, there is no clue for prediction.

Table 2 shows the prediction success rate of NightWatch. Due to inter-chunk locality similarity, a high prediction success rate is attained for most of the programs – 14 out of 23 programs have a prediction success rate over 99%. Note that the predictor falls short for *namd*, with a low success rate of 58%. This is due to the adaptive sampling method of the locality monitor, which doubles the sampling interval (i.e., number of skipped chunks in a row) upon every successful prediction. While helping to reduce monitoring overhead, such

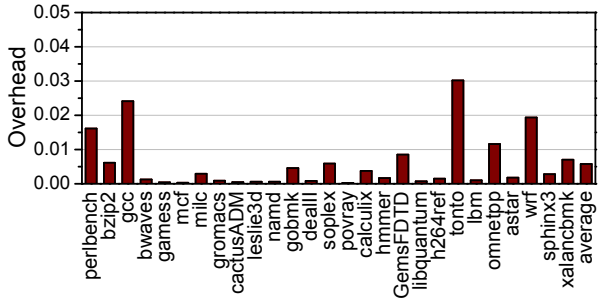


Figure 14: The overhead of *nw_tcmalloc*, defined as the fraction of a program's execution time spent on cache management

a design responds slowly to locality property change occurring across chunks within an allocation context. If mapping switching happens during a large sequence of skipped chunks, all follow-up chunk requests will receive a wrong prediction, until the next sampling point is reached. Nevertheless, such a case rarely happens for the 27 benchmarks; for most of the programs inter-chunk locality similarity remains valid, yielding an average prediction success rate of 95.5%.

6.5 Overhead Analysis

nw_tcmalloc's integrated cache management adds an extra time cost to a program's execution. The ratio between this time cost and a program's overall execution time defines *nw_tcmalloc*'s *overhead*. Specifically, *nw_tcmalloc*'s cost consists of three parts: the monitor's time cost T_{mon}, the predictor's time cost T_{pred}, and the time spent on mapping switching $T_{mswitch}$. Due to the high prediction success rate, $T_{mswitch}$ is negligible – compared with the overall execution times of the 27 benchmarks, which range from 250 to 1000 seconds, $T_{mswitch}$ is less than 1 second.

Figure 14 presents the overheads of *nw_tcmalloc*. On average, the overhead is only 0.57%, with a maximum 3.02%. Furthermore, for 22 of the 27 programs, *nw_tcmalloc*'s overhead is less than 1%. This indicates that *nw_tcmalloc* is highly efficient while offering performance benefits.

The monitor's cost, T_{mon}, is caused by locality evaluation for sampled pages and chunks, and thus depends on the total size of allocated chunks, denoted by $SIZE_{total}$. NightWatch implements a number of optimization strategies to reduce the cost. Figure 15 shows T_{mon} of the 27 benchmark programs sorted by $SIZE_{total}$. It can be seen that T_{mon}, which varies across two orders of magnitude, increases much more slowly than $SIZE_{total}$, which spans four orders of magnitude. This sublinear growing trend suggests that the monitoring cost scales very well with total allocation size. Notice that T_{mon} does not always increase with $SIZE_{total}$, because T_{mon} also depends on the size distribution of chunks, and how chunks are clustered

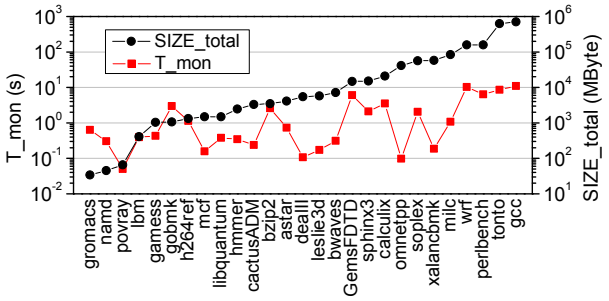


Figure 15: T_{mon} vs. $SIZE_{total}$ for 27 programs, sorted by $SIZE_{total}$

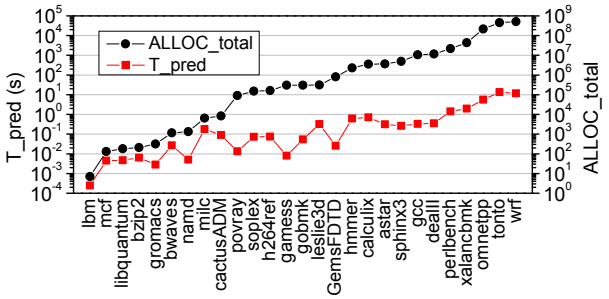


Figure 16: T_{pred} vs. $ALLOC_{total}$ for the programs, sorted by $ALLOC_{total}$

in contexts: small sized chunks and low clustering effect in contexts will reduce the benefit of locality similarities, thus causing a higher time cost.

The predictor's cost of a program, T_{pred}, is determined by the total number of allocations, denoted by $ALLOC_{total}$. Figure 16 presents T_{pred} of the 27 benchmark programs sorted by $ALLOC_{total}$. With the help of size-to-context lookup table, T_{pred} grows much more slowly than a linear function of $ALLOC_{total}$. On average, one second's cost allows NightWatch to make 34 million predictions, which make it suitable for programs that involve extremely frequent chunk allocation. For example, the *wrf* program takes 1000.4 seconds to complete, making a total of 500 million allocations. It takes NightWatch only 23 seconds to make all the predictions, accounting for 2.3% of the program's execution time.

6.6 Evaluation on Ptmalloc2

Due to space limits, we only briefly report on our evaluation on *ptmalloc2*. For the *pollter-victim* combinations, the NightWatch integrated allocator improves *victim*'s performance by up to 51% (18% on average), with an average overhead of 0.6% (up to 3.0%).

7 Related Work

There has been extensive research on improving cache efficiency for multi-programmed workloads. In this section we discuss the main technical approaches used and related work in the area. For clarity, we list the representative work in Table 3.

Cache-aware memory allocator: From the perspective of resource management, perhaps the closest systems to NightWatch assisted allocators are *ULCC* [6] and *ccontrol* [28]. Both designs attempt to incorporate cache control into the memory allocator in order to achieve higher cache efficiency. They enable cache control by providing special interfaces to programmers to modify the program's source code for improved resource utilization. Apparently this approach requires a deep understanding of the program's data access behavior. Adding to the complexity of this problem is the fact that many programs' data locality property varies at run time due to multiple factors (such as input, software configuration). A proper grasp of these complicated issues is thus beyond the capability of common programmers. Given the nonstandard interfaces, these designs also fail to provide transparent support for legacy programs.

Our work is the first to *transparently* integrate cache management into dynamic memory allocation. Compared with previous solutions, NightWatch assisted allocators hide the complexity of cache management and provides standard interfaces. This makes it easy to use and fully compatible with legacy software.

Cache bypassing: Commodity processors provide cache bypass instructions to bypass weak-locality data. Rus et al. [23] propose to develop automatical tools to help identify weak-locality string operations and transform them to cache bypass instructions. Sandberg et al. [25] employ off-line reuse distance analysis to characterize the access locality of the overall program, and replace non-temporal data accesses with bypass instructions. These approaches need off-line analysis and hence fail to adapt to a program's dynamic runtime behavior.

Software cache partitioning: Cache partitioning has been proven an effective approach to improve cache utilization [7, 13, 14, 22, 27, 32, 33]. Lin et al. [14] propose to partition the shared cache for co-running programs and isolate cache pollution from weak-locality data accesses. Their dynamic partitioning technique adjusts the cache partitions to accommodate locality changes of data. RapidMRC [29] guides cache partitioning to achieve optimal speedup, with the help of the *Miss Rate Curves* (MRC) of each program. The hot-page coloring method [31] enforces cache partitioning for hot pages to reduce the overhead of cache re-partitioning. ROCS [27] clusters weak-locality pages to a dedicated pollution buffer on cache. Soft-OLP [16] analyzes the reuse distance of each major data object, and performs proper cache allocation based on the object's locality properties and interactions. These techniques suffer from a number of limitations. First, some techniques need specialized hardware support from the processor. For example, [27, 29] use POWER processor's *Sampled Data Address Register* (SDAR) to evaluate MRC and pages' miss rate, which is not available in other processors. Second, it is hard to effectively obtain data reuse information [5, 16, 26]. It is reported in [21] that the collection of reuse information can degrade the performance by a factor of 13 to 50, even with accelerated instrumentation on 64 processors. Third, re-partitioning cache is expensive. Page recoloring for cache re-partitioning incurs significant overheads [31].

	Locality analysis	Dynamic memory allocation	Transparent to user	Specific hardware support
ULCC [6]	manually	yes	no	no
ccontrol [28]	manually	yes	no	no
Soft-OLP [16]	off-line	yes	no	no
Sandberg et al. [25]	off-line	no	yes	no
Rus et al. [23]	off-line	no	yes	no
RapidMRC [29]	on-line	no	yes	yes
hot-page color [31]	on-line	no	yes	yes
ROCS [27]	on-line	no	yes	yes
NightWatch	on-line	yes	yes	no

Table 3: Comparison of shared cache pollution management techniques

Compared with the above solutions, NightWatch provides the benefits without their limitations. It can efficiently obtain the data locality information on commodity processors and reduce the cache re-partitioning overhead with the help of data locality prediction.

8 Conclusion

In this paper we have presented NightWatch, an efficient cache management subsystem designed for memory allocators. The distinguishing feature is that NightWatch provides runtime support for pollution control, using only standard allocation interfaces and assuming no special hardware support. At the heart of the solution are two observations about locality correlation of data that are exploited to realize highly efficient locality monitoring and prediction. We have demonstrated the efficacy of NightWatch through extensive experiments, showing speedup of up to 1.5X for pollution victim programs at very low overheads. It should be noted that NightWatch is not limited to multi-program workloads; the multithreaded version is also open-sourced. In future, we plan to conduct a comprehensive evaluation on the effectiveness of the multithreaded version.

9 Acknowledgements

This paper is supported by China National Natural Science Foundation under grant No. 61322210, 61379135, 61433019, Doctoral Fund of Ministry of Education of China under grant No. 20130142110048, National Hightech Research and Development Program of China (863 Program) under grant No. 2015AA015303, US National Science Foundation grant ACI-award No. 1432892.

References

[1] AKRITIDIS, P. Cling: A memory allocator to mitigate dangling pointers. In *Proc. USENIX Security Symposium 2010*, pp. 177–192.

[2] AMDINC. Amd64 architecture programmers manual volume 2: System programming.

[3] BERG, E., AND HAGERSTEN, E. Fast data-locality profiling of native execution. In *Proc. SIGMETRICS 2005*, ACM, pp. 169–180.

[4] BERGER, E. D., MCKINLEY, K. S., BLUMOFE, R. D., AND WILSON, P. R. Hoard: A scalable memory allocator for multithreaded applications. *ACM Sigplan Notices 35*, 11 (2000), 117–128.

[5] DING, C., AND ZHONG, Y. Predicting whole-program locality through reuse distance analysis. In *Proc. PLDI 2003*, ACM, pp. 245–257.

[6] DING, X., WANG, K., AND ZHANG, X. Ulcc: A user-level facility for optimizing shared cache performance on multicores. In *Proc. PPoPP 2011*, ACM, pp. 103–112.

[7] DUONG, N., ZHAO, D., KIM, T., CAMMAROTA, R., VALERO, M., AND VEIDENBAUM, A. V. Improving cache management policies using dynamic reuse distances. In *Proc. MICRO 45* (2012), IEEE Computer Society, pp. 389–400.

[8] EVANS, J. Scalable memory allocation using jemalloc. `http://www.canonware.com/jemalloc/`, 2011.

[9] GHEMAWAT, S., AND MENAGE, P. Tcmalloc: thread caching malloc. google performance tools. `https://code.google.com/p/gperftools/`, 2004.

[10] GLOGER, W. Ptmalloc. `http://www.malloc.de/en/`, 2006.

[11] INTELINC. Intel® 64 and ia-32 architectures software developers manual.

[12] JALEEL, A., NAJAF-ABADI, H. H., SUBRAMANIAM, S., STEELY, S. C., AND EMER, J. Cruise: cache replacement and utility-aware scheduling. In *ACM SIGARCH Computer Architecture News* (2012), vol. 40, ACM, pp. 249–260.

[13] KIM, H., KANDHALU, A., AND RAJKUMAR, R. A coordinated approach for practical os-level cache management in multi-core real-time systems. In *Proc. ECRTS 2013*, IEEE, pp. 80–89.

[14] LIN, J., LU, Q., DING, X., ZHANG, Z., ZHANG, X., AND SADAYAPPAN, P. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *Proc. HPCA 2008*, pp. 367–378.

[15] LIU, R., AND CHEN, H. Ssmalloc: a low-latency, locality-conscious memory allocator with stable performance scalability. In *Proceedings of the Asia-Pacific Workshop on Systems* (2012), ACM, p. 15.

[16] LU, Q., LIN, J., DING, X., ZHANG, Z., ZHANG, X., AND SADAYAPPAN, P. Soft-olp: Improving hardware cache performance through software-controlled object-level partitioning. In *Proc. PACT 2009*, pp. 246–257.

[17] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proc. PLDI 2005*, ACM, pp. 190–200.

[18] MANIKANTAN, R., RAJAN, K., AND GOVINDARAJAN, R. Nucache: An efficient multicore cache organization based on next-use distance. In *Proc. HPCA 2011* (2011), IEEE, pp. 243–253.

[19] MATTSON, R., GECSEI, J., SLUTZ, D., AND TRAIGER, I. Evaluation techniques for storage hierarchies. *IBM Systems Journal 9*, 2 (1970), 78–117.

[20] MAUERER, W. *Professional Linux kernel architecture*. John Wiley & Sons, 2010.

[21] NIU, Q., DINAN, J., LU, Q., AND SADAYAPPAN, P. Parda: A fast parallel reuse distance analysis algorithm. In *Proc. IPDPS 2012*, pp. 1284–1294.

[22] QURESHI, M. K., AND PATT, Y. N. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proc. MICRO 39* (2006), IEEE Computer Society, pp. 423–432.

[23] RUS, S., ASHOK, R., AND LI, D. X. Automated locality optimization based on the reuse distance of string operations. In *Proc. CGO 2011*, IEEE Computer Society, pp. 181–190.

[24] SANCHEZ, D., AND KOZYRAKIS, C. Vantage: scalable and efficient fine-grain cache partitioning. In *ACM SIGARCH Computer Architecture News* (2011), vol. 39, ACM, pp. 57–68.

[25] SANDBERG, A., EKLÖV, D., AND HAGERSTEN, E. Reducing cache pollution through detection and elimination of non-temporal memory accesses. In *Proc. SC 2010*, IEEE Computer Society, pp. 1–11.

[26] SCHUFF, D. L., KULKARNI, M., AND PAI, V. S. Accelerating multicore reuse distance analysis with sampling and parallelization. In *Proc. PACT 2010*, ACM, pp. 53–64.

[27] SOARES, L., TAM, D., AND STUMM, M. Reducing the harmful effects of last-level cache polluters with an os-level, software-only pollute buffer. In *Proc. MICRO 41* (2008), IEEE Computer Society, pp. 258–269.

[28] SWANN PERARNAU, M. T., AND HUARD, G. Controlling cache utilization of hpc applications. In *Proc. ICS 2011*, ACM, pp. 295–304.

[29] TAM, D. K., AZIMI, R., SOARES, L. B., AND STUMM, M. Rapidmrc: Approximating l2 miss rate curves on commodity systems for online optimizations. In *Proc. ASPLOS 2009*, ACM, pp. 121–132.

[30] WU, C.-J., JALEEL, A., MARTONOSI, M., STEELY JR, S. C., AND EMER, J. Pacman: prefetch-aware cache management for high performance caching. In *Proc. MICRO 44* (2011), ACM, pp. 442–453.

[31] XIAO, Z., SANDHYA, D., AND KAI, S. Towards practical page coloring-based multicore cache management. In *Proc. EuroSys 2009*, ACM, pp. 89–102.

[32] XIE, Y., AND LOH, G. H. Pipp: promotion/insertion pseudo-partitioning of multi-core shared caches. In *ACM SIGARCH Computer Architecture News* (2009), vol. 37, ACM, pp. 174–183.

[33] YE, Y., WEST, R., CHENG, Z., AND LI, Y. Coloris: A dynamic cache partitioning system using page coloring. In *Proc. PACT 2014* (2014), ACM, pp. 381–392.