Controlling Cache Utilization of HPC Applications

Swann Perarnau Marc Tchiboukdjian Guillaume Huard
INRIA MOAIS Team, CNRS LIG Lab, Grenoble University
{perarnau,tchiboukdjian,huard}@imag.fr

ABSTRACT

This paper discusses the use of software cache partitioning techniques to study and improve cache behavior of HPC applications. Cache partitioning is traditionally considered as an hardware/OS solution to shared caches issues, particularly to resource utilization fairness between multiple processes. We believe that, in the HPC context of a single application being studied/optimized on the system, with a single thread per core, cache partitioning can be used in new and interesting ways.

First, we propose an implementation of software cache partitioning using the well known page coloring technique. This implementation differs from existing work by giving control of the partitioning to the application programmer. Developed on the most popular OS in HPC (Linux), this cache control scheme has low overhead both in memory and CPU while being simple to use.

Second, we show how this user-controlled cache partitioning can lead to efficient measurements of the cache behavior of a parallel scientific visualization application. While existing works require expensive binary instrumentation of an application to obtain its working sets, our method only needs a few unmodified runs on the target platform.

Finally, we discuss the use of our scheme to optimize memory intensive applications by isolating each of their critical data structures into dedicated cache partitions. This isolation allows the analysis of each structure cache requirements and leads to new and significant optimization strategies. To the best of our knowledge, no other existing tool enables such tuning of HPC applications.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Measurement Techniques

General Terms

Experimentation, Performance

ICS 2011 June 1–4, Tucson, Arizona.

Keywords

page coloring, cache partitioning, working set

1. INTRODUCTION

The memory cache behavior of high performance computing (HPC) applications is a topic that has been the focus of numerous studies. Most of those studies analyze the cache usage ratio of a target application: how much cache is efficiently used. This usage ratio is closely related to *working sets* [2, 10].

From a general point of view, the working sets model the performance of a process relative to its resource utilization during a time interval. In particular, these working sets highlight specific ranges of values of the quantity of resources assigned to the process for which the process performance does not vary. Applied to the full execution period it outlines the resources an application requires to reach a given performance level. In [2, 10], this model has been applied successively to cache performance analysis.

Closely related to working sets, the reuse distance of an application, introduced by Beyls and D'Hollander in 2001 [1], plays a major role in its cache performance. The reuse distance is defined, for each memory access performed by the application, as the number of different memory accesses realized before the next access to the same location, if any. Under the assumption that the application runs on a machine equipped with a fully associative cache using the least recently used (LRU) policy, this metric expresses exactly the efficiency in cache of the application: measuring the reuse distance of each memory access will determine if a cache miss will be triggered by this access. Regarding set associative caches, where line eviction depends on memory accesses in the same set and where the LRU implementation usually presents slight modifications, several papers studied and confirmed the accuracy of the reuse distance [1, 24].

Formally, if the number of accesses having a reuse distance of d is $H(d)$, then the number of cache misses $Q(C)$ occurring on a cache of size C is: $Q(C) = \sum_{d=C+1}^{\infty} H(d)$ where a reuse distance of ∞ is associated to the initial access to each element. Thus, the working sets of an application are directly related to its reuse distance: if there exist a range $[i, j]$ for which H is null (*i.e.* $\forall d \in [i, j]\ H(d) = 0$), then $Q(C)$ will stay the same for values of C in this range $[i, j]$. This is obvious: if memory accesses of an application do not change from misses to hits when giving it slightly more cache, its performance will stay constant. More precisely, working sets can be deduced from reuse distances as the integral function of their distribution.

Unfortunately, despite its obvious usefulness, the evaluation of the reuse distances of a given application is tremendously difficult. Static analysis of the source code is quickly limited by its complexity and by missing runtime data. As an alternative, reuse distance are often determined by gathering the application's memory accesses using tools like Pin [20] or Valgrind [22]. They can also be used to measure working sets by feeding the memory access trace to a cache simulator (Valgrind even include a virtually indexed one). Nevertheless, the simulation of all the memory accesses of an application requires huge computational resources, limiting those experiments to short runs.

Our first contribution is a tool and a method for measuring working sets of an application. Our method do not suffer from the huge computational overhead induced by simulation methods. Indeed, determining one point of the working sets function just requires one regular run of the application. To achieve this result, we make use of well known page coloring techniques [16] to implement a cache control mechanism. Then, we use this cache control mechanism to assign to a given application a chosen fraction of the hardware cache. The resulting performance is a point of the working set function Q.

Our second contribution aim at improving the cache performance of a single HPC application. In particular, we show that our cache control method can be used to evaluate how memory accesses to each distinct data structure of the application contribute to the working set function. We deduce from this information an estimation of the cache requirements of each of these data structures. Combining this information with our cache control tool, we allocate to each data structure a well chosen fraction of the cache: memory accesses to this structure are then cached only to this fraction of the hardware cache. Finally we demonstrate that carefully choosing the partition size of each data structure can result in significant performance improvements.

The remainder of this paper is organized as follows. The next section presents a simple software cache partitioning mechanism based on page coloring. It differs from previously presented works as it gives control of partitions to users (application programmers) instead of the OS. Section 3 describes the implementation of our proposal on the Linux Operating System as well as its interfaces. We validate this implementation both as a page coloring facility and a cache controller in Section 4. This validation is based on working sets detection of a perfectly understood application, it makes sure that the working set changes accordingly to the cache partition in use.

This working set analysis is then applied in Section 5 to a parallel visualization application having more complex memory access patterns. This application can be configured to use different parallelization schemes. Thus, using our tool, we determined its working sets to select the most cache efficient parallelization scheme. To the best of our knowledge this paper is the first to focus on such working set analysis on actual application executions in the HPC context.

Section 6 presents various possible uses of our cache partitioning scheme to improve the cache performance of several parallel applications. First, we show how the noise of a data structure with close-to-none reuse distance can be suppressed, giving more cache to the remaining data structures of a visualization application. Then, we analyze in details an application (a multigrid stencil) to determine the working set of each of its data structures. Those working sets are evaluated by isolating each data structure inside its own partition and making the partition size vary. This analysis leads to a global partitioning of the application, dramatically improving its performance.

Finally we compare our tool to related works in Section 7 and summarize our results in Section 8.

2. CACHE CONTROL BY PAGE COLORING

Our cache control tool is based on a straightforward and lightweight method: page coloring. It has been designed for way-associative, physically indexed caches, but can also be applied to direct mapped ones. As most cache architectures are nowadays way-associative or direct mapped caches, our control scheme can be applyed on almost all recent systems. For clarity we define C as the cache size, A as its associativity (*i.e.* its number of ways), L as the cache line size and P as the page size. All sizes are in bytes.

As page coloring is critical to understand both our cache control mechanism and the experiments in the remainder of the paper, we recall its principle in the following subsection.

2.1 Page Coloring

Most modern architectures use physically indexed caches. In such systems, if the mapping of virtual pages to physical ones performed by the virtual memory (VM) subsystem is not properly chosen, unnecessary cache conflicts can be triggered during processes execution. Kessler *et al.* [16] showed that a VM subsystem choosing page mappings arbitrarily contributed up to 30% of total cache conflicts in an application execution. They proposed several *careful-mapping* algorithms to solve this problem, of which page coloring is the most popular in Operating System research. Several major OS implement it in their virtual memory subsystems (FreeBSD and Windows NT among others) and it has been praised as a key component of cache optimization of applications as well as a good performance stabilizer [8, 17].

Page coloring identifies by a color the group of physical pages that conflict (or overlap) in a cache. This definition arises from the inner working of physically indexed caches. It can be summarized as follows. Physical memory is cached line by line and each page is several lines long. As lines are mapped to associative sets in a round-robin fashion, consecutive lines of the same physical page are mapped to several, consecutive associative sets. The number of lines (and sets) in a cache being limited, many pages map to the same associative sets. A color identifies indiscriminately the group of pages overlapping in cache or the group of associative sets they map to. As a cache possesses C/AL associative sets and a page occupies P/L cache lines, the number of colors in a cache is C/AP. Figure 1 illustrates this page mapping and the corresponding colors on an hypothetical cache with 8 associative sets of 8 ways and physical pages being two lines long.

An OS virtual memory subsystem implementing page coloring tries to optimize cache utilization by giving different colors to consecutive virtual pages. As a page color never changes, page coloring is easily implemented in an efficient way. Of course, as the number of colors in a system is limited, it might still be necessary to give some pages of the same colors to a memory demanding process.

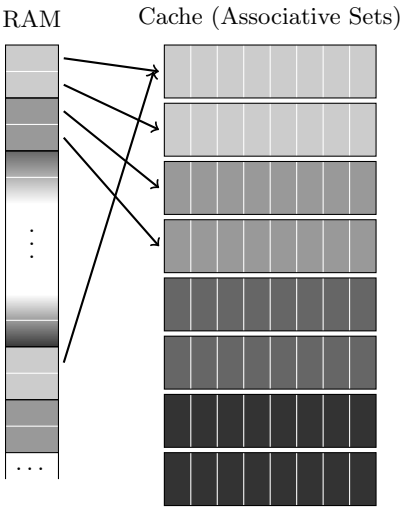

Figure 1: Page coloring in an hypothetical system with 2 lines per page and a 4 colored cache with 8 ways. Pages are placed in the cache in round robin, each line in a different associative set. Pages of the same color are placed in the same associative sets.

In a multiprogrammed environment this definition of page coloring does not suffice. To ensure fair resource sharing, page coloring is also tuned to give different colors to distinct processes. This way, applications competing for the same core will not trash each other's cache. This well studied issue [14, 18] is close to our work although we focus on cache sharing inside a single application. Indeed, our goal is to provide a cache partitioning interface based on page coloring directly to applications. Furthermore, in most HPC context, applications only have one thread per core. Thus, the precise issues we study relate to cache sharing (which is discussed later in this article), rather than resources contention.

2.2 Cache Control

In a sense, page coloring was one of the first cache partitioning algorithm. In this special case, each color represents a partition and giving processes different colors ensures they use different portions of the cache. Our cache control mechanism is a direct extension of this partitioning scheme, allowing a partition to span several colors. It works in two phase: first, the user sets a portion of physical memory aside for the page coloring scheme. Then he is provided with a specific memory allocation *device* which returns pages of a configurable set of colors in response to allocation requests. This control scheme allows an application programmer to select the colors allocated to some dynamic memory allocations (*i*.e data structures), creating custom cache partitions, usable in parallel. As further sections of this paper will show, letting applications control their cache partitioning can improve greatly their performance. Because the memory accesses of most HPC applications are well understood, our scheme should be easy to apply.

Our mechanism does not provide automatic page recoloring, a classical feature of cache partitioning schemes. This is a design choice: we consider the whole automatic recoloring mechanism as too intrusive in the context of HPC applications. Furthermore, a programmer knows better the key phases of its application and when to trigger recoloring. Indeed he can implement it by creating two different devices (with different colors) and copy data from one to the other.

We should also mention that partitioning cache induces a partition of the memory. This means that, as with any other cache partitioning scheme, the memory available to one partition is limited to the pages that can fall is that partition (in our case, the ones with the good color). In other words, a small cache partition will contain few colors, thus few pages to use. Since we provide those partitions to applications as memory mappable devices, a small partition will limit the size of the virtual memory mapped to it. This sounds like a constraining limitation, but our cache control scheme is only remapping available memory to specific parts of a process address space. Applications having enough memory without cache control should have enough with cache control enabled, except that it might be necessary to give cache partition larger than they need (regarding reuse distance) to data structures that occupy a large space in memory.

Most modern processors are composed of several cores and a cache hierarchy. Top level caches are shared by all cores and low level ones are private to a single core. In the remaining of this article we do not address the complex problem of cache partitioning across the whole hierarchy. We rather have chosen to enforce the partitioning according to the top level only, to benefit from the greater flexibility it provides and to avoid noise during experimental measurements. This choice already gives promising results and do not change our analysis of presented applications. Nevertheless, we plan to inspect the additional improvements that could result from a multilevel partitioning scheme in our future works.

3. IMPLEMENTATION

We implemented our cache control mechanism on Linux. We chose this OS for several reasons. First, contrary to some other operating systems, Linux does not implement page coloring. This makes our implementation easier as no existing mechanism needs to be bypassed. Second, according to the latest TOP500 [21], Linux (or its variants) is the most popular OS in HPC.

3.1 Linux Memory Subsystem

Before explaining how our cache control scheme has been implemented, some information on virtual memory management in Linux is necessary. Using the standard GNU C library, a process calling the memory allocation functions can trigger two events. If the requested allocation size is large (more than a page) the library will call the system function *mmap*. In the other case, the library will return a memory location coming from a pool of pages. In the latter case, the pools are managed dynamically, thus, along with new allocation requests, new pages will be asked eventually. Consequently any memory allocation will eventually ask the OS for more virtual pages.

This request is always handled in the same way: Linux creates or expands an *area*. Such area represents a region of the address space of a process that is managed by the same memory handler. Once new pages are made available to the process, the Linux kernel returns without having touched any of the pages (no page faults are triggered). For each area, a particular memory handler inside Linux is in charge of page faults handling. Thus, when a process accesses a virtual page for the first time, the kernel dispatch the page

fault to the fault handler of the area which faulted. Classically this means a physical page will be allocated to the process, but it could also triggers a DMA to fetch a file region on disk or instruct a particular device to send data on a network for example. Thus, in the Linux kernel, virtual memory management works in two steps: virtual pages are made available inside the virtual address space and physical pages are allocated when page faults occur.

Linux is a modular kernel: modules (code) can be loaded at runtime to enable additional functionalities. Such modularity also extends to virtual memory management: the new code can add to the system special (virtual) files having their dedicated page fault handler. Once a process will have those files mapped to its virtual memory, the first access to this *area* will trigger the dedicated page fault handler. Thus, kernel modules can add functionalities to the memory subsystem.

3.2 Cache Control as a Kernel Module

Cache control is implemented as a very simple module and a set of special devices. The kernel module is responsible for the management of a configurable number of contiguous physical page blocks (obtained from the kernel memory allocator). Once the color of each allocated page has been identified, users can ask for the creation of memory mappable virtual devices to the module by issuing `ioctls` on a special control file. Such commands contain the size and authorized colors of the new device.

When an application maps one of the device into its address space, every page fault in the corresponding area will trigger our module page fault handler, which will provide a page with an allowed color. Thus, the user can create cache partitions dynamically during the execution of its application.

When allocating memory, the module tries to reserve the same number of pages for each color. This design is motivated by the Linux physical memory allocator behavior: to obtain several pages of the same color, it is necessary to allocate all the memory between them. Such behavior makes allocating the same number of pages to each color much simpler than any other possibility (like asking the desired number of pages for each partition). As a result, the module allocates physical memory by contiguous blocks as large as the requested memory and partitions it according to colors.

Once the module is loaded, two interfaces to the control mechanism are available to users. The first one is a library providing simple functions to define a set of colors and a *zone*: a memory allocator working inside a specific device. This interface is close in design to systems like the Linux `hugepages`: a function creates a zone using a size (maximum number of bytes that can be allocated) and a color set, and memory allocations can then be made inside the zone. The zone corresponds exactly to a colored device. To ease development and to enable the use of our control scheme by existing applications, we provide a second interface which is a memory allocation hijack. Such interface enable any user to install a custom library intercepting any standard POSIX memory allocation call (`malloc,realloc,calloc,free`) on any C application. Upon loading, this custom library will create various partitions and map them to the process address space, creating various memory allocation pools inside the application. Upon an allocation request the library will determines the pool to use and return a part of this memory pool to the user process. It is, of course, possible to tune this library by changing the function determining for each allocation the pool to use.

```
#include<ccontrol.h>

char *t;
struct ccontrol_zone *z;
color_set c;

/* allocate region */
COLOR_ZERO(&c);
for(int i = 0; i < 32; i++)
COLOR_SET(i,&c);

z = ccontrol_new_zone();
i = ccontrol_create_zone(z,&c,size);
t = ccontrol_malloc(z,size);

do_stuff(t,size);

ccontrol_free(z,t);
ccontrol_destroy_zone(z);
ccontrol_free_zone(z);
```

Figure 2: Code snippet using our cache control library to create a cache partition of 32 colors and to allocate some character array into it.

Figure 2 gives a code snippet describing the few functions calls needed to create a cache partition of 32 colors and allocate some character array into it.

The kernel module is a lightweight process (both in CPU and memory). During the execution of the target application it is only triggered by page faults in the virtually mapped partitions. The Linux kernel calls our page fault handler with the page number (as an offset to the first virtual page of the area) to allocate. Thus our fault handler only retrieves one page pointer from its color arrays and returns it. The setup phase of the module, during which physical memory is requested from the kernel might seems more costly. It is not the case, even if a huge number of physical page allocations is made, the kernel is still able to respond quickly. For example, in the experimentation system used in the following sections, allocating 24 GB of physical memory to our module takes approximately 1 second. Our module also has low memory requirements as it only saves one pointer per physical page managed.

4. VALIDATION

We validated our cache control mechanism on two aspects: its capability to provide memory allocations with a good page coloring and its capability to partition the hardware cache.

4.1 Experimental Setup

All experiments were conducted on a Quad Intel Xeon E5530 System. Each CPU possesses 4 cores, with a L_1 Data cache size of 32 KB, a L_2 Unified cache, 8 ways associative of 256 KB and a L_3 Shared Unified cache, 16 ways associative of 8 MB. All caches have 64 B lines.

All our validation experiments use the same memory intensive application. This program consists only of a huge number of random accesses (reads) to a single memory region dynamically allocated. The size of this memory region can be configured. Given the fact that the number of

accesses performed on the memory region outnumbers the number of elements in it, this application will have performance depending on its ability to cache said memory region. If the region fit in L_1 cache then the first access to each element will cache it and further accesses will all be hits. As the region size grows, upper levels of the cache will be required until the top level does not suffice (then the application will touch frequently the physical memory).

Thus, if we measure the average access time per read on our setup, the application will exhibits 3 working sets: a first one when the region is smaller than L_2 cache, a second one when the region fits in L_3 and the third one when the region is bigger than L_3. A fourth working set could have appeared when the region fits the L_1 cache, but practically the performance drop is too small to be noticed on our system. Notice that a similar program was used dy Ulrich Drepper [10] to demonstrate this working set effect with Valgrind as a cache simulator.

Each data point in the following figures is the result of 100 executions of 5 000 000 reads on a memory region. The program was fixed on a single core, running on real-time scheduling policy (SCHED_FIFO) with max priority (ensuring no other program disturbs the measurements). Confidence intervals were too small to be included.

In the following, we refer to partition size as the number of colors used by a partition: it directly maps to the amount of cache made available.

4.2 Results

In the first experiment (Figure 3) we compare working sets of our application when the region is managed by the Linux kernel to a cache controlled region allocated by our tool and having access to the whole cache. In such setup, because of our pages allocation method, our cache control is only performing a classical page coloring on the memory region. Thus, as the Linux kernel does not implement page coloring the performance of the measured application should drop faster when the memory region size is close to the L_3 size. This experiment also validates our page fault handler: if it performs poorly the whole application performance will suffer from it.

As both physical memory allocators give the whole cache to the application, we can observe what we expected: 3 working sets corresponding to the size of each cache in the hierarchy. A small performance drop appears at region size 2^{17} which is the size of the L_2 and a big performance drop around 2^{23} (the size of the L_3 cache). Notice that under cache control the program still achieves good performance for a region of the same size as the L_3, whereas the imperfect page allocation of Linux make performance drop faster. This validates that our cache controller performs a proper page coloring. Those results were confirmed by measuring cache misses during the same experiment with hardware performance counters [3].

Next we validate the cache partitioning in itself: by making the available cache size vary, the application should exhibit working sets at different sizes. We compare several partition sizes given to the whole random access application. Each cache size should make the last working set of this program appear when the memory region get close to it. Figure 4 reports our measurements. The performance drop in access time follows closely the cache partition size. This assesses that our program behaves as if its cache was only the size of the partition it uses.

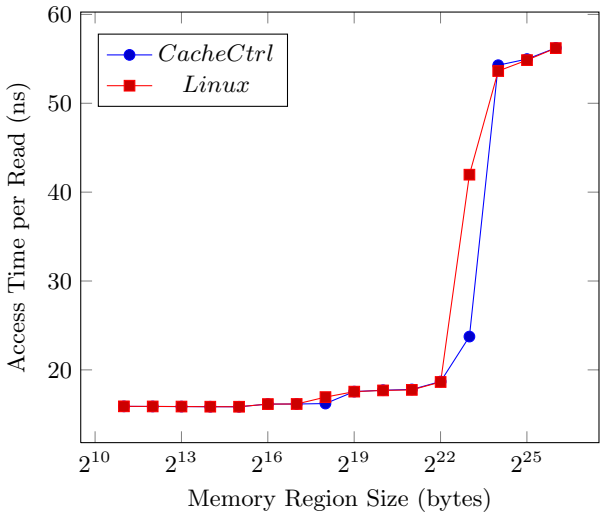

Figure 3: Random reads on a memory region using Linux page allocation and the cache controlled one.

5. WORKING SETS ANALYSIS FOR ALGORITHMIC CHOICES

This section presents an analysis of the cache performance of a parallel application from the scientific visualization domain. In particular it shows how the analysis of its working sets can lead to appropriate algorithmic choices for its parallelization.

5.1 Isosurface Extraction with Marching Tetrahedron (MT)

Isosurface extraction is one on the most classical filters of scientific visualization. It provides a way to understand the structure of a scalar field in a three dimensional mesh by visualizing surfaces having the same scalar value. Our application is based on the marching tetrahedrons (MT) algorithm, known for its good performance [15]. For each cell of a mesh, the MT algorithm reads the point coordinates and scalar values and computes a triangulation of the isosurface going through this cell. The triangulation consists of 0, 1 or 2 triangles according to how the isosurface intersects the tetrahedron.

The cache misses induced by MT can be analyzed as follows. The mesh data structure consists of two multidimensional arrays: an array storing for each point the coordinates and a scalar value and an array storing for each cell the indexes of its points (*cf.* Figure 5). Due to the mesh construction process, the order of points and cells has some locality: points and cells close to each other in the mesh space often have close indexes. Thus, processing cells in the order of their indexes induces fewer cache misses when accessing the point array due to an improved locality: successive cells often share common points or points located in the same cache line. This locality can even be optimized by reordering points and cells to obtain better cache performance [27].

5.2 Parallel MT for Shared Cache

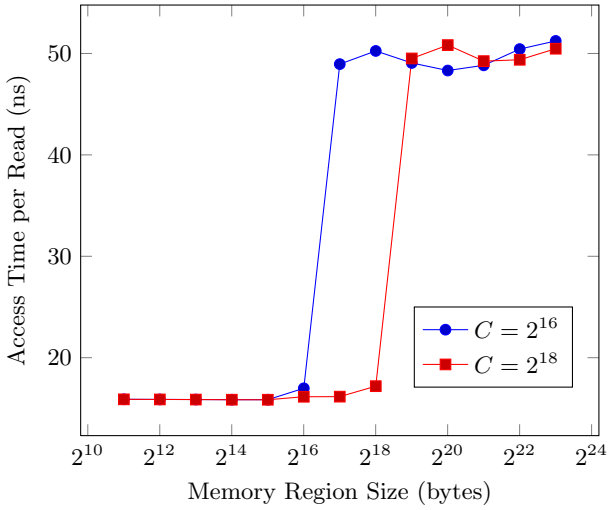

Figure 4: Random access: access time per element for varying memory size and various cache sizes.

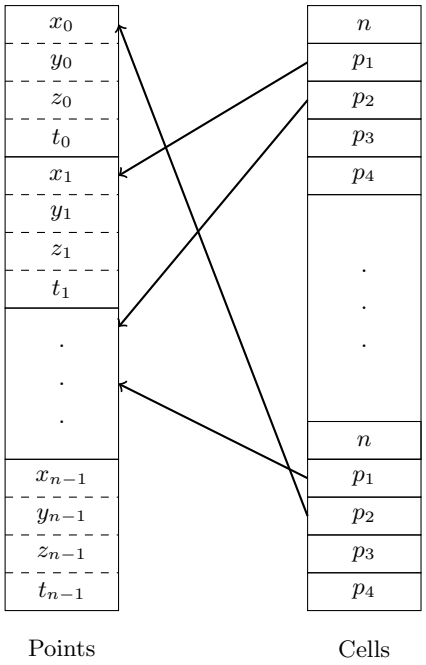

Figure 5: Mesh Data Structure. The point array stores coordinates and scalar values (t) and the cells array contains indexes to points defining each cell.

As each cell can be processed independently, it is relatively easy to parallelize the MT algorithm. One can logically divide the cell sequence into contiguous chunks and assign one chunk to each processor core. Since cell processing times differs according to the number of triangles generated, we used a work stealing scheduler to dynamically balance the load. When a core becomes idle, it selects another core at random and steal half of its remaining cells. This scheme efficiently uses the private caches of a multicore processor: each core processes contiguous cells and maximizes the reuse of points loaded in its private cache. However, cores operate on parts located far from each other in the cell sequence (and thus in the mesh space), reducing the chance that two cores use common points. Therefore, this parallel algorithm, denoted NoWindow, does not efficiently use the last level of cache of multicore processors which is shared amongst all cores.

To improve the reuse of data stored in the shared cache, a new parallel algorithm denoted SlidingWindow has been introduced in a previous work [26]. A fixed size window sliding on the cell sequence constrains cores to operate on cells close in mesh space. Threads still process chunks of contiguous cells for efficient private cache usage but these chunks are now smaller and closer to each other in the cell sequence thus improving shared cache usage. The SlidingWindow algorithm can be efficiently implemented using work stealing. The core operating at the beginning of the window has a specific status and is called the master. Steals to other cores are treated in the same way as the previous algorithm. However, when another core steals the master, it can only steal cells inside the window. The master is responsible for sliding the window on the sequence to enable new cells to be processed. This stealing mechanism guarantees that cores are operating inside the window at all time.

5.3 Shared Cache Misses Analysis with Reuse Distances

The SlidingWindow algorithm improves shared cache usage but increases synchronization overheads compared to the NoWindow algorithm as cores are stealing smaller amounts of work. We would like to use the SlidingWindow algorithm only when it significantly reduces the number of shared cache misses. We show in this section how we can predict the gain in shared cache misses of the SlidingWindow algorithm over the NoWindow algorithm using the working sets function Q of the sequential algorithm which processes the cell sequence in order.

Let $H(d)$ denote the number of memory references with a reuse distance d in the sequential algorithm. The number of cache misses on a fully associative cache of size C is given by $Q(C) = \sum_{d=C+1}^{\infty} H(d)$. We assume that the sequential algorithm has good temporal locality, *i.e.* cells far away from each other in the sequence use distinct points while cells having close indexes use common points. We first consider the NoWindow parallel algorithm on p cores sharing a cache of size C. In this case, as distinct cores do not operate on common points, the reuse distance is equal to the reuse distance of the sequential algorithm multiplied by p: each access performed by a core is followed by $p-1$ unrelated accesses performed by the other cores in parallel. Thus, $H_{\text{no-win}}(d) = H(\frac{d}{p})$ and the number of cache misses of the NoWindow algorithm is

$$Q_{\text{no-win}}(C) = \sum_{d=C+1}^{\infty} H\Big(\frac{d}{p}\Big) = \sum_{d=\frac{C}{p}+1}^{\infty} H(d) = Q\Big(\frac{C}{p}\Big).$$

The NoWindow algorithm induces as many cache misses as the sequential algorithm with a cache p times smaller.

We now consider the SlidingWindow algorithm where cores operate on elements at distance at most m in the cell sequence. Let $r(m)$ be the maximum number of distinct memory references when processing $m-1$ consecutive elements of the cell sequence. In the worst case, when processing the last element of the window, all other elements have

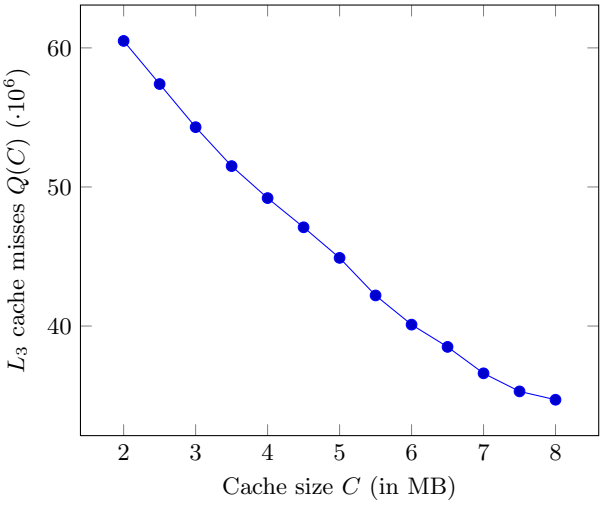

Figure 6: Number of shared cache misses of the sequential MT algorithm for varying cache sizes.

	L_3 cache misses	Time (ms)	Speedup
Sequential ($C = 2$MB)	$60.5 \cdot 10^6$	5015	0.66
Sequential ($C = 8$MB)	$34.7 \cdot 10^6$	3320	1.00
NoWindow	$55.3 \cdot 10^6$	1137	2.92
SlidingWindow	$38.4 \cdot 10^6$	964	3.44

Table 1: Performance of the two parallel MT algorithms NoWindow and SlidingWindow compared to the sequential algorithm.

been processed accessing at most $r(m)$ additional distinct elements compared to the sequential algorithm. Thus the reuse distance is increased by at most $r(m)$. The number of cache misses of the SlidingWindow algorithm is

$$Q_{\mathrm{w}}(C) \leq \sum_{d=C+1}^{\infty} H(d - r(m)) = Q(C) + \sum_{d=C+1-r(m)}^{C} H(d).$$

As we assumed the sequence has good temporal locality, $r(m)$ is small compared to m and $H(d)$ is small for large d. Therefore $\sum_{d=C+1-r(m)}^{C} H(d)$ is small and the SlidingWindow algorithm induces approximately the same number of shared cache misses as the sequential algorithm.

We experimentally verify this result by computing the number of shared cache misses for the sequential algorithm $Q(C)$ for cache sizes C varying from 2MB to 8MB on the Xeon E5530 (*cf.* Figure 6). We used a mesh composed of 150,000,000 cells. The number of cache misses for a cache of 2MB is much greater than when using an 8MB cache. Thus, we expect that using the SlidingWindow algorithm will result in a big gain in cache misses compared to the NoWindow algorithm on the 4 cores of the Xeon E5530. It is the case (*cf.* Table 1): the NoWindow exhibits almost the same number of misses as the sequential algorithm using four times less cache. Consequently its speedup is hindered by its poor use of this application locality. In contrary, as expected, the SlidingWindow version induces only a few more cache misses than the sequential algorithm and offers a better speedup.

6. DATA STRUCTURE(S) ISOLATION

	L_3 cache misses	Time (ms)
Linux	$37.1 \cdot 10^6$	4124
Cache Control	$34.7 \cdot 10^6$	3320
Optimized Cache Control	$23.7 \cdot 10^6$	3090

Table 2: Performance of the sequential MT algorithm with 3 different allocation policies.

Our objective, in this section, is to distribute as best as possible the cache available to the application. Our approach works in two steps: first we estimate from a derivative of the working sets analysis the cache requirements associated to each data structure of the application. Secondly, we use our cache controller to distribute the cache according to the first analysis. This method is especially effective in identifying the data structures (or memory regions) that would benefit the most from an increased cache size. We first use it to inhibit the negative effects of a streaming access pattern in the MT application. Then, we demonstrate better cache allocation strategies on a stencil algorithm similar to multigrid applications.

6.1 Avoiding Cache Pollution Due to Streaming Accesses in MT

When examining the access patterns of the MT algorithm, one can notice that only accesses to points exhibit data reuse. The other two access patterns, reading the cell sequence and writing the generated triangles, are pure streaming accesses and waste cache space. To avoid this negative effect, our implementation of the MT uses non temporal instructions to write the triangle sequence [7]. These instructions bypass the cache to avoid polluting it with useless data. We would like to do the same while reading the cell sequence. Unfortunately, non temporal read instructions do not exist. However, our cache controller can avoid the cache pollution by streaming-like access patterns by isolating them to a small portion of the cache. We have added this optimization on the cell array of the mesh in our MT implementation and compared the resulting 3 methods to allocate our data structures in Table 2.

In the first method, denoted Linux, we simply allocate them using `malloc`. In the second one, denoted Cache Control, we allocate them with our cache controller in a single region using all available colors thus realizing a perfect mapping of virtual to physical memory. In the last one, denoted Optimized Cache Control, we allocate the point array in a region composed of 100 colors and the cell and triangle arrays in the remaining 28 colors. As expected, due to a better color distribution in allocated pages, using a perfect mapping reduces the number of cache misses and improves the running time. Moreover, allocating more cache space to the array that exhibits reuse while confining the data structure accessed in streaming into a small cache region greatly reduces the number of cache misses (by 36%) and offers the best performance, a speedup of 1.33 over the unmodified application.

6.2 Multigrid Stencil

To demonstrate the full potential gains that can be obtained when using our cache partitioning scheme, we programmed a toy application derived from classical stencil filters. Our application makes a simultaneous use of three

different matrices that reside in memory to compute the elements of a result matrix. The input matrices form the multigrid structure, it is made of a large matrix ($Y \times X$ 64 bytes elements), a medium-sized matrix (one fourth of the large matrix size) and a small matrix (one sixteenth of the large matrix size). The resulting matrix has the same size as the large matrix. Each of its elements is a linear combination of nine points stencils taken from each input matrix at the same coordinates (interpolated for smaller matrices).

This application is interesting for two reasons: it is extremely memory intensive and it makes a simultaneous use of several working sets of different sizes. Our nine points stencil forms a cross (a center element, the two elements above it, the two elements on the right, and so on) and it is included in five lines of a matrix. Thus, in the ideal case, if five lines of each input matrix can remain in the cache during the computation, the stencil will be computed with a maximal reuse. This translates into a cache space of $X \times 64 \times 5$ bytes for the large matrix, half of this size for the medium one and one fourth of this size for the small one. Of course if these working sets are not mapped to disjoints colors sets, they will mutually trash each other cache when running the application.

Because the previous testing architecture (Intel Xeon) is known to prefetch memory too aggressively for the kind of access patterns our application contains, the following experiments were done on an Intel Core 2 Duo System. It contains 2 cores, with a L_1 Data cache size of 32 KB, a L_2 Unified cache, 16 ways associative of 4 MB. All caches have 64 B lines. From a coloring point of view, the L_2 cache contains 64 colors, each being 64 KB wide.

As we want to measure the cache requirements of each data structure independently, we modified our stencil so that it can use a different cache partition for each of the matrices. Then, for each application run, we isolated one of the matrices in a dedicated partition and the others in another one. The experiment consists in measuring the cache misses of the whole application while varying the size of the cache part associated to the isolated data structure. As the rest of the application is confined in a cache part of fixed size, the variation of cache misses can only be the result of the variation of the cache size given to the isolated data structure. As a result, we obtain the shape of the working sets of each data structure present in the application.

Figure 7 gives the resulting working sets for a partition size varying between 8 and 56 colors. The other data structures are contained in a fixed partition of 8 colors. This application was run with $X = 7168$ and $Y = 100$. Matrices are named from the smallest one M_1 ($X/4$ by $Y/4$) to the biggest M_3 (X by Y), the result matrix is named M_r.

Notice that, on such experiments, a difference in cache misses between structures for a given number of colors does not convey any significance: different structures are present in the small, static cache partition, inducing a different number of cache misses for the rest of the application. Thus, only the shape of the curve (the variation in cache misses for each structure) is of interest.

These working sets match the theoretical analysis of this application: each matrix needs to cache 5 rows at most to benefit optimally from cache, except for M_r which is only written to (needing no cache at all). As each matrix M_i is two times larger than M_{i+1} (in X), each matrix needs twice more cache than the previous. Given those working sets, we

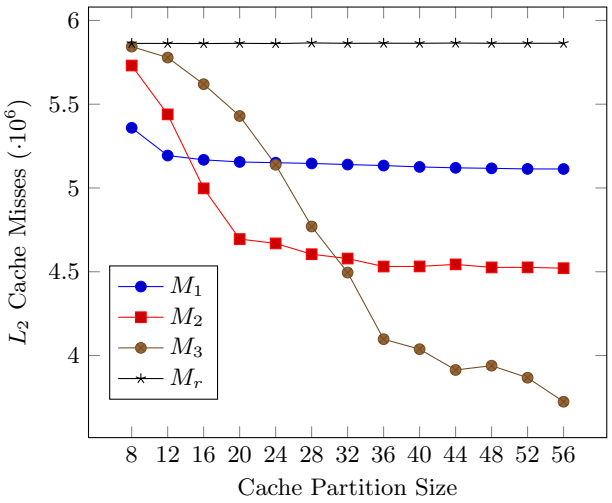

Figure 7: Multigrid Stencil: L_2 cache misses per partition size for each data structure.

	L_2 cache misses	Time (ms)
Linux	$3.6 \cdot 10^6$	139
Cache Control	$3.3 \cdot 10^6$	78
Optimized Cache Control	$2.2 \cdot 10^6$	57

Table 3: Performance of the stencil application with 3 different allocation policies.

choose an optimized cache partitioning with 9 colors for M_1, 18 for M_2, 35 for M_3 and 2 for M_r. Table 3 presents the resulting performance of our application compared to the unmodified (Linux) and the single partition (Cache control) versions.

Using our cache controller, compared to the unmodified application, we achieve a tremendous performance improvement of 38% in L_2 cache misses and a speedup of 2.4 regarding the execution time. This very good speedup can be explained easily: a cache miss cost a lot more in execution time than a cache hit. Thus, reducing the number of cache misses significantly improves the running time of our application.

Of course, such results are only an indication of the kind of performance improvements our cache control scheme makes available. This stencil application is specially tailored so that an unmanaged cache is not able to use the locality of each data structure while a good partitioning can fit in the whole cache. Nevertheless, we can safely assume that any application containing streaming access patterns or very different reuse distances per data structure should benefit from our scheme.

7. RELATED WORKS

Soft-OLP [19], a tool making cache partitions at the object level inside an application using page coloring, is the closest work related to our proposal. However Soft-OLP relies, like most previous works, on binary instrumentation techniques: to determine a good partitioning among objects, they measure reuse distances for each object on instrumented runs (with Pin). As the authors acknowledge, instrumented runs are 50 to 80 times slower than real executions. To cope with

longer execution times, they extrapolate the cache usage of each object in real word instances using measurements on small inputs (supposed representative). Our tool does not suffer from such limitations: measuring working sets is as fast as unmodified runs and thus can handle real sized instances. Instead of using Pin, we isolate each object in its own cache partition and measure misses with varying partition size.

A small part of the optimizations presented here were also discussed by Soares *et* al. in 2008 [25]. They proposed a OS scheme capable of detecting streaming access patterns using hardware performance counters. More precisely, they detected the virtual pages causing the most cache thrashing and isolated them to a small partition (using page coloring). Such technique can also be related to cache-bypassing instructions found on most modern architectures [7]. We believe our scheme goes further, by balancing cache usage between data structures (even if overall they do not fit in cache). We demonstrated its use both to reduce streaming access patterns influence on performance and to better distribute the cache among data structures of an application.

Several works have already resulted in the implementation of a Linux kernel module to provide a basic software cache partitioning mechanism using page coloring [5, 17, 18]. However, these works were focused on a different context: cache sharing issues among multiple running processes on the same core. Such issues require tedious OS optimization and complicated heuristics whereas our proposal deals with only one thread per core, focusing on providing good colors to the right memory region used by the thread. Moreover, not a single paper has provided its implementation to the community, limiting the reuse of these works.

Most existing implementations of page coloring also make a trade-off between protecting processes from each other and avoiding cache trashing inside an address space. Several works presented a cooperation between hardware mechanisms and compilers do cope with the latter [4, 23]. Unfortunately such methods are limited to data reorganization techniques inside compilers and QoS strategies of the OS, whereas our cache control allows any application programmer to fine tune the cache usage of each of its data structures.

Finally, control of the virtual memory subsystem by user programs has also been suggested in the domain of micro and exokernels [11, 12]. In those cases, an application could ask the OS to use other virtual memory managers than the default one, thus making it possible to rewrite a virtual memory manager fine tuned for a single memory access pattern. Several issues prevented those works to ever be made available in standard HPC configurations. First, all the virtual memory manager (not just the physical memory allocator) needed to be replaced. Such OS component is among the most complicated and it is considered too cumbersome to rewrite it for the improvement of a single application. Second, most of those specific operating systems are designed for single processor architectures and were never ported to HPC ones.

8. CONCLUSION

The Internation Exascale Software Project Roadmap [9] defines the support for explicit management of the memory hierarchy by runtime systems/user applications as one of the most critical aspect of future operating systems for HPC.

We believe our cache control tool to be a first step in the design of such mechanism. We presented a simple scheme, based on a well understood technique (page coloring) to expose to user applications control of the partitioning of the cache hierarchy. We validated this tool in Linux, one of the most popular OS in HPC. Our implementation is simple to use, efficient both in CPU and memory and does not necessarily require modifications of the target application. To the best of our knowledge, our tool is the first to allow measurements of the working sets of an application or of its data structures using only a small number of runs.

We also have presented a collection of experiments that highlight different usages of our proposal. We demonstrated how the analysis of application working sets can give hints about the proper parallelization scheme to use for them. We applied to two different applications our methodology for fine cache optimization. First, we used the analysis of the working sets of each data structure of a parallel scientific visualization application to remove cache thrashing due to small-reuse access patterns inside it. Then, this working sets analysis was used on a stencil application to design a dramatic optimization of its cache behavior. These experiments assessed that our methodology was sound, useful and might produce a reduction in caches misses of up to 38% in the most favorable cases and similar (or even better) improvements in execution time.

As future works, we plan to study the possible additional improvements that could be obtained when using a partitioning scheme sensitive to the cache hierarchy. Our cache control tool as well as all the code used for our experiments are available on `http://ccontrol.gforge.liglab.fr`.

9. REFERENCES

[1] K. Beyls and E. D'Hollander. Reuse distance as a metric for cache behavior. In *Proceedings of the IASTED Conference on Parallel and Distributed Computing and systems*, volume 14, pages 350–360, 2001.

[2] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: characterization and architectural implications. In *Proceedings of 17th International Conference on Parallel Architecture and Compilation Techniques*, pages 72–81, 2008.

[3] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *The International Journal of High Performance Computing Applications*, 14(3):189–204, 2000.

[4] E. Bugnion, J. Anderson, T. Mowry, M. Rosenblum, and M. Lam. Compiler-directed page coloring for multiprocessors. *ACM SIGOPS Operating Systems Review*, 30(5):255, 1996.

[5] S. Cho and L. Jin. Managing distributed, shared l2 caches through os-level page allocation. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 455–468, 2006.

[6] J. Corbet, A. Rubini, and G. Kroah-Hartman. *Linux Device Drivers*. O'Reilly Media, 3rd edition, 2005.

[7] I. Corporation. *Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference*.

[8] M. Dillon. Design elements of the FreeBSD VM system.

[9] J. Dongarra et al. The international exascale software project roadmap. *International Journal of High Performance Computer Applications*, 25(1), 2011.

[10] U. Drepper. What every programmer should know about memory, 2007.

[11] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, pages 251–266, 1995.

[12] K. Harty and D. R. Cheriton. Application-controlled physical memory using external page-cache management. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 187–197, 1992.

[13] J. L. Hennessy and D. A. Patterson. *Computer Architecture: a quantitative approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 1996.

[14] R. R. Iyer. CQoS: a framework for enabling QoS in shared caches of cmp platforms. In *Proceedings of the 18th International Conference on Supercomputing*, pages 257–266, 2004.

[15] C. Johnson and C. Hansen. *Visualization Handbook*. Academic Press, Inc., 2004.

[16] R. E. Kessler and M. D. Hill. Page placement algorithms for large real-indexed caches. *ACM Transactions on Computer Systems*, 10:338–359, 1992.

[17] S. Kim, D. Chandra, and Y. 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, 2004.

[18] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *Proceedings of the 14th International Conference on High-Performance Computer Architecture*, pages 367–378, 2008.

[19] Q. Lu, J. Lin, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Soft-olp: Improving hardware cache performance through software-controlled object-level partitioning. In *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 246–257, 2009.

[20] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, 2005.

[21] H. Meuer, E. Strohmaier, H. Simon, and J. Dongarra. 35th release of the TOP500 list of fastest supercomputers, 2010.

[22] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, pages 89–100, 2007.

[23] T. Sherwood, B. Calder, and J. S. Emer. Reducing cache misses using hardware and software page placement. In *Proceedings of the 13th International Conference on Supercomputing*, pages 155–164, 1999.

[24] M. Snir and J. Yu. On the theory of spatial and temporal locality. Technical report, 2005.

[25] L. Soares, D. K. Tam, and M. Stumm. Reducing the harmful effects of last-level cache polluters with an os-level, software-only pollute buffer. In *41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 258–269, 2008.

[26] M. Tchiboukdjian, V. Danjean, T. Gautier, F. Le Mentec, and B. Raffin. A work stealing scheduler for parallel loops on shared cache multicores. In *Proceedings of the 4th Workshop on Highly Parallel Processing on a Chip (HPPC 2010)*, 2010.

[27] M. Tchiboukdjian, V. Danjean, and B. Raffin. Binary mesh partitioning for cache-efficient visualization. *Transactions on Visualization and Computer Graphics*, 16(5):815 –828, sep. 2010.