# Model Driven Cache-Aware Scheduling of Object Oriented Software for Chip Multiprocessors

Tolga Ovatman and Feza Buzluca

Istanbul Technical University,
Department of Computer Engineering,
34469 Maslak, Istanbul, Turkey
ovatman,buzluca@itu.edu.tr

**Abstract.** Leveraging utilization of the shared caches of multicore processors is one of the heavily studied topics of today's chip multiprocessing community. Providing a scheduling mechanism that maximizes throughput by reducing miss-rates of shared caches and preserves the fairness of processor usage is in the center of this problem. Proposed scheduling algorithms in this field usually take advantage of thread level properties of software providing modifications at operating system level. In our study we choose to approach the problem from a different perspective and use software models to guide operating system to effectively map software's objects onto processor cores. In an object oriented software objects collaborate on fulfilling jobs and they may operate on common data. Our scheduling method takes class dependencies into account and tries to schedule objects of coupled classes onto cores that share the common cache. This paper presents case studies on implementations of three software design patterns(Strategy, Visitor and Observer) and an image filtering software implementation. During our experiments we use our cache-aware scheduler in guiding Linux's completely fair scheduler (CFS) to perform more cache-aware schedules and increase performance. Our results promise that guiding/restricting operating system's scheduler using class-relational information present in the object oriented software model can be fruitful in increasing software performance on multicore processors.

## 1 Introduction

For the last decade, mainstream in processor technology is chip multiprocessors, also named as multicore processors, which involve multiple processing cores in a processor die. By their nature, multicore processors utilize parallel running software where cores are assigned to each thread produced by parallel decomposition of software. This assignment operation is done by operating system schedulers, which put emphasis on fairness and load-balancing problems rather than utilization of shared data among threads. As multicore architectures get more complicated, cache memories not only serve as buffers for accelerating memory

access of threads but also provide a rapid communication medium for shared data among threads. Recent multicore processor architectures contain relatively smaller caches for each distinct core and larger shared caches for the cores that reside on the same chip. It can be expected to encounter more complicated cache hierarchies as the number of cores increase.

Aside from this situation, current operating system schedulers do not provide an effective way to deal with cache utilization of processors yet. Instead, their primary concern is more fair time-slicing of processing elements to provide user balanced running time of applications [1] [2] [3] [4]. This is quite natural since operating system scheduler is expected to run on a wide range of processor architectures and application software. Leveraging different concerns in such a heterogeneous environment is a serious challenge, that becomes more important as multicore processors continue evolving towards manycore processors.

Improving operating system schedulers to take cache utilization into account is being heavily studied by the community. In most of the studies, a single centralized solution to replace the scheduler is proposed using data gathered from runtime profile of software [5] [6, 7] [8] [9] [10] [11] [12] [13]. Since proposed improvements are at operating system level, software analysis are carried on lower level software structures like loops or thread groups.

Apart from approaches based on modification of operating system's scheduler, another idea is guiding the scheduler using classes as higher level software components. In our study we try to experiment if extracting such guidelines from object oriented software design can improve Linux's completely fair scheduler (CFS). We apply our approach on design pattern implementations and gain performance improvement when the scheduler is guided regarding coupled classes of software. Coupled classes access methods of each other frequently, raising the probability of shared data between their objects at runtime. We use design patterns (which can be found frequently in object oriented software) to reason about possible object tuples that frequently share data at runtime.

At the end of our experiments it can be seen that extracting information from the software model and placing tightly coupled objects into neighboring cores (cores that share the same cache) improves operating system's scheduler performance. Our approach does not need to change the whole scheduling mechanism of the system. Instead, we analyze the dependency relation among classes in the class diagram of software and provide a set of candidate cores for the classes that have the potential to communicate frequently at runtime. Placing those classes' objects at neighboring cores decrease cache miss rates by taking advantage of shared data between software classes.

Rest of the paper is organized as follows. Section 2 summarizes the studies on scheduler improvement studies regarding cache utilization. Section 3 explains the concept of cache aware scheduling and important factors that effects the process of exploring shared data for parallelization. In Section 4 minor examples are given to demonstrate the effects of cache-aware scheduling of threads followed by more complicated design pattern examples in Section 5. Three different design pattern implementations (strategy,visitor and observer) are used

to demonstrate the performance improvement gained by scheduling data sharing objects at neighboring cores. Experiments on a real-world image filtering software is presented at Section 6. Last section concludes the paper and briefly presents future studies.

## 2 Related Work

Previous work on cache-aware scheduling on multicore systems generally takes advantage of dynamic information of software provided by runtime analysis [5] [6, 7] [8] [9] [10]. This type of scheduling can be supported with the information obtained by static analysis of software models and shared data between them. Wickzier et al. provide annotations for the programmer to explicitly guide their $O^2$ scheduler called CoreTime in managing shared data among multiple threads [11]. Xue et al. also proposed a method claiming that static scheduling can be made locality aware by ensuring that the set of iterations assigned to a processor exhibit data reuse [12]. Our approach takes one step further and evaluates the impact of inter-class relationships of software's object oriented model to guide its scheduling.

Another interesting point in Xue's study is the usage of loops as recurring software components in scheduling decisions. Loops are heavily used in software parallelization/cache-utilization studies before. Tam et al. utilize threads as disjoint components of parallel/concurrent software and schedule them based on sharing patterns they pose at runtime [7]. In other words, they basically find coupled threads at runtime and schedule them to share L2 caches. Federova et al. identify coupled threads as co-runner threads and try to reduce performance variability caused by cache-unfair scheduling of them [13]. We focus on coupled software components at object oriented level and use the data sharing classes' objects (which are already specified at software model/code) to guide the operating system's scheduler.

Using static software models is another rarely used subject in cache-aware scheduling studies. One of those studies that explicitly uses models and software abstractions in maximizing cache reuse in multicore scheduling is done by Kumar and Delgrande [14]. They try to solve optimal multicore scheduling problem by using a graph theoretic formulation and answer set programming in their study. In our study we specifically use object oriented software models to reason about data sharing among software's classes.

In our previous study, we examined if it was possible to capture parallelization behavior together with frequently used design patterns [15]. In addition to the parallelization recipe for the design patterns, in this study we also provide scheduling guidance based on characteristic data sharing between patterns' classes. Our study makes three contributions on the topic of scheduling for multicore systems; 1) investigating the usability of higher level static information, 2) use of object oriented approach and design patterns, 3) applying model based analysis to discover shared data usage.

## 3 Cache-Aware Scheduling

In the context of this paper, we use the term *Cache-Aware Scheduling* to indicate the operation of guiding operating system's scheduler with the information of shared data between software classes. Shared data can be detected dynamically via runtime environment or an external dynamic analysis tool. However partial or full development of the software at hand is needed to perform this kind of analysis. In our study we have chosen to use software models and static class diagrams to reason about parallelism at an early stage of software development.

Using software models to guide scheduling provides two important advantages. Firstly, we can obtain parallelization information before the actual software runs or even before it is implemented. This helps us to design more competing software for multicore systems and to produce parallel code that performs better on different multicore architectures. Secondly, we provide the ability to guide the operating system's scheduler without replacing it during the scheduling process. The analysis of software model at hand can be performed semi-automatically by a programmer or an automated tool to detect data sharing software components of software. According to this information the operating system's scheduler tries to assign objects, which operate on common data to proper cores so that shared data can be placed into shared caches.

During the analysis of the software three different factors in parallelization should be considered.

- Parallelization : The number of distinct parts in software that can run independently. They should be scheduled to different cores.
- Data sharing : Object tuples that share a significant amount of data regarding shared/non-shared caches. They should be scheduled to neighboring (or same) cores.
- Resource utilization : The ratio of cores/caches to the number of objects that run on the system.

Resource utilization is heavily influenced by parallelization and data sharing since these two factors have an orthogonal effect on system performance. Decomposing software too much for the favor of parallelization causes objects to write on different caches frequently and increase cache misses. On the other hand scheduling objects strictly on neighboring cores to utilize cache reuse may cause parallelized objects to wait for the same core even though there are some other idle cores present. This situation decreases the parallelization performance when there exists fewer cores in the die than the objects to be scheduled. During our experiments we try to explore how these factors effect each other to extract more meaningful information from the model. We use practical real-world examples based on design pattern implementations, small enough to successfully observe the effect of each factor during the scheduling.

In our studies we use Gang of Four (GoF) software design patterns [16] to analyze data sharing classes of the pattern. Software design patterns are frequently used in today's object oriented software designs to solve common problems. Additionally, a large number of studies exist in the literature about

detecting software design patterns [17] [18] [19] [20] [21] [22], making it possible to automatically inject our cache-aware scheduling directives inside software.

We apply our cache-aware scheduling technique on design patterns to show that even for smaller parts of the software we can provide better scheduling using data sharing information between components. This approach can be applied to larger software where many different instances of many design patterns can be found and analyzed for data sharing. In this paper we focused on the applicability of model based scheduler guidance by analyzing data usage of recurring themes in software designs. Our approach is not limited to specific software design patterns but rather offers to use parallelization strategies together with patterns that emerge in software designs.

## 4 Effects of Cache-Aware Scheduling

### 4.1 Case Studies on Software Design Patterns

Our experiments are performed in a system with 4 double cored Intel Xeon processors and an operating system of Linux kernel 2.6 running on it. We use Java as the main programming language to develop the design pattern case studies. Since Java lacks an API to explicitly set a thread's processor affinity, we used C++ to implement *pthread*'s [23] thread affinity setting functions [24] and JNI to call our C++ thread affinity setter implementations from Java programs. *pthread* library allows thread distribution via `sched_setaffinity` and `CPUSET` functions which can be used to explicitly define thread-to-processor distribution schemes for the objects in the patterns. For the majority of the experiments, objects of the patterns are programmed as separate threads, and assigned to processors either explicitly under control of the programmer or automatically by the system scheduler. In our experiments we repeat program runs for a sufficient number of times to let the running time average converge.

Figure 1 presents the processing element architecture used in our experiments which consists of four different processors each having two cores with a shared L3 cache of 4096KB in size. In our scheduling schemes we use the term "neighboring cores" to indicate the cores that reside in the same physical processor and share the same cache (e.g. 1-7, 2-5, 3-6, 4-8). We call our proposed scheduling approach as CAWS (Cache-Aware Scheduling) where the threads that share data are placed onto neighboring cores as much as possible. Linux's CFS scheduler actually does not take caches into account and migrate the threads often, resulting threads to share caches in a non-determined way.

For our case studies, we implement three different design patterns: Strategy, Visitor and Observer. All these patterns commonly consist of some master (service requester)-worker (service provider) classes. UML diagrams of the mentioned patterns can be found below.

For strategy (Figure 2), each strategy object (worker) provides a service of applying a different algorithm on the client (master/service requester) object. Data is shared between strategy and client objects for this pattern. At runtime

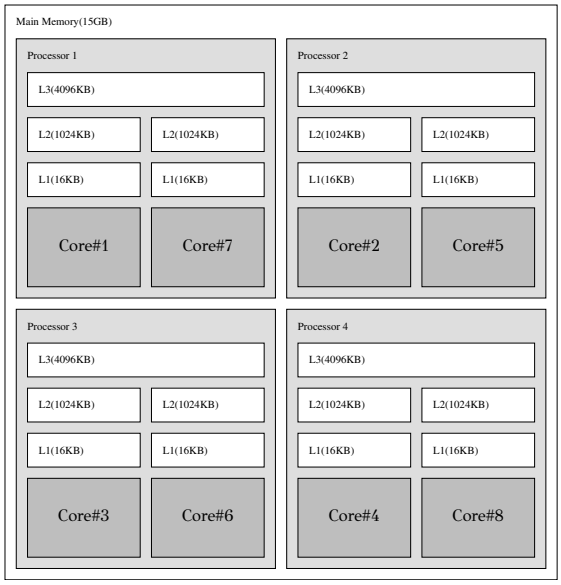


**Fig. 1.** Central Processing Unit Architecture

there may be many clients (service requester) running in parallel using a specific strategy object in common.

In visitor (Figure 3), each visitor object provides its service when it is called explicitly by the master (service requester) object. At runtime there may be many elements requesting services from a set of visitor objects arbitrarily. Some visitor objects may be used in common during these service requests as well. Objects that implement the Visitor interface and Element objects that are visited by Visitors are data sharing components for Visitor pattern.

In observer (Figure 4), a subject object presents the update notification service of its states to a set of observer objects. At runtime some observer objects may register to different common subjects. A Subject object and its observers commonly use the state of the Subject in this design pattern.

Similar examples can be implemented for other patterns as well, we choose our examples in this paper illustrate different data sharing (read-only, read/write) and thread creation schemes. Our implementations are explained in detail in Section 5 but before initiating more complicated experiment scenarios it can be useful to illustrate the effect of cache reuse in scheduling design patterns on basic experimental configuration.

### 4.2 Effects of Cache-Aware Scheduling on Basic Examples

To show that sharing common caches makes a notable performance difference at runtime we need to provide a basic set of isolated examples showing the

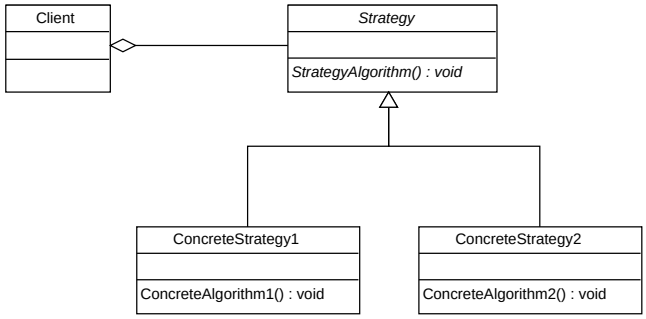


**Fig. 2.** Strategy Design Pattern

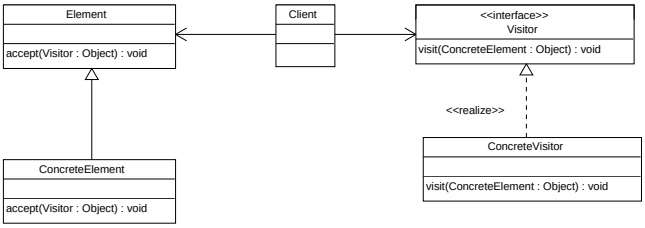


**Fig. 3.** Visitor Design Pattern

difference of cache-aware scheduling with respect to its counterparts. For this purpose we configure our implementations consisting of only one master-worker object couple for each design pattern. In each of the examples below there only exist two objects at runtime sharing a fixed amount of data that is proportional to the size of common caches in the processor.

For each example we used the worst-case running time to normalize running times between 0 and 1 (worst performance). The results we obtained for each of the examples are as follows.

- **Strategy :** In Table 1 we can see that for a large quantity of shared data, placing two objects at neighboring cores(CAWS) outperforms the CFS. When the amount of data being shared gets smaller cache sharing effect loses its significance.

**Table 1.** Normalized running times for basic strategy implementation

| Shared Data: | 1MB | 8KB | None |
|---|---|---|---|
| CFS | 0.95 | 0.99970 | 0.99965 |
| CAWS | 0.87 | 0.99965 | 1.00000 |

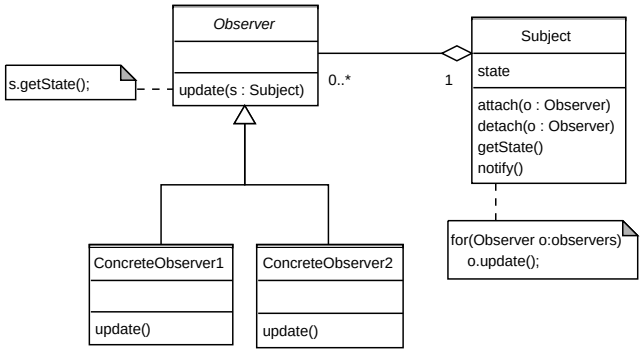


**Fig. 4.** Observer Design Pattern

- **Visitor :** In Table 2 we can see similar results with Table 1. When the amount of shared data gets closer to shared cache sizes using a cache-aware scheduling starts to perform better.

**Table 2.** Normalized running times for basic visitor implementation

| Shared Data: | 1MB | 8KB | None |
|---|---|---|---|
| CFS | 0.81 | 0.96 | 0.9998 |
| CAWS | 0.77 | 0.96 | 1.0000 |

- **Observer :** Finally in Table 3 we can see similar results but this time an additional scheduling scheme has also been added(referred to as SACS(Same Core Scheduling)). Since one observer and one subject cannot run parallel at all we can place them at the same core at runtime. When placed at same core, with an amount of data small enough to fit the private cache, the system had a superior performance.

**Table 3.** Normalized running times for basic observer implementation

| Shared Data: | 1MB | 8KB | None |
|---|---|---|---|
| CFS | 0.99 | 1.00 | 1.00 |
| CAWS | 0.87 | 1.00 | 1.00 |
| SACS | 0.87 | 0.29 | 0.99 |

As it can be seen from the running times above, scheduling the data sharing objects in a way that allows them to use the same processor cache outperforms

the Linux's CFS. We can also see that for the objects that have sequential behavior and use shared data, scheduling them at the very same core provides superior performance since it allows storing shared data at private cache of the core.

From our basic examples above we see that migrating shared data among processors and re-fetching large amounts of data inside the memory hierarchy are time consuming operations that degrade software performance. By running experiments on multi-object examples we can comment about cache-aware scheduling on more realistic cases.

## 5 Applying Cache-Aware Scheduling

We shall continue our experiments with more complicated configurations on design patterns to show the difference between cache-aware scheduling and current scheduler of Linux. In this section we let many objects inside the design patterns interact during runtime using different parallelization approaches. For all the patterns below, we instantiate different number of objects for each different type of class that the pattern contains. We implement each object as a separate thread, hence two terms (object and thread) are used interchangeably in this Section. We briefly describe the parallelization approaches we apply for the following patterns and discuss the results for different configurations.

### 5.1 Strategy

For strategy pattern, we construct a constant number of strategy objects, each representing a different strategy for a specific number of client objects. Each client object is affiliated with a strategy object at runtime working on a predetermined amount of shared data that is smaller than the size of the shared cache. For the sake of simplicity, the data of the client is always read (never written) by the strategy for this case.

In Figure 5, we can see Normalized Running Times(NRT) of 32 client objects under different scheduling policies using different number of strategies. When the number of parallelized parts (strategies) are less than number of distinct processing cores in the system, we can see a performance gain which is caused by reduced missing rate during the data access of threads. If the number of parallelizable parts exceed the number of cores (8 in our system), scheduler starts to preempt threads and change cache content thus the effect of cache-aware scheduling vanishes for number of strategies more than 8. A speedup of nearly 10% compared to CFS, is present when cache aware scheduling is used.

### 5.2 Visitor

In this case, desired number of visitors are constructed independently before element objects run. When an element needs a visitor one is taken from the pool and assigned to the element object. Since waiting times can vary for each element

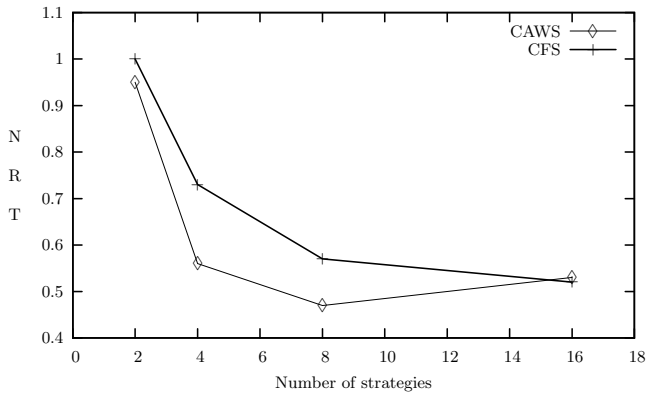


**Fig. 5.** Scheduling strategies with different policies

and each visitor, each class holds a queue of the next object to provide/request service. Visitors hold a queue of elements to start serving the next object in line after the ongoing work finishes. A similar situation is present for elements as well, they hold a queue of visitors to ask for a service. For this case a more complicated structure is used where any visitors may visit any elements during runtime; unlike strategy no predetermined element-visitor bindings are applied before system run.

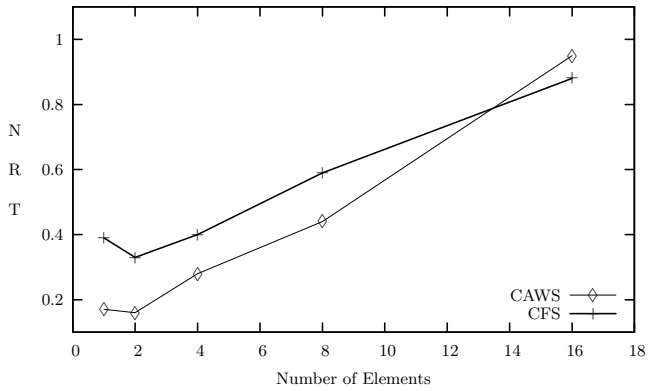


**Fig. 6.** Scheduling 8 Visitors with different policies

In Figure 6 we can see that cache-aware scheduling outperformed others until the number of parallelized objects reach the number of cores. Additionally, even when we schedule visitors on distinct cores from elements but in the same cores

with other visitors, CAWS still outperform CFS. This time we use cache read and writes so we don't experience a cache utilization as much as strategy case.

### 5.3 Observer

Implementation of observer adopts a different object construction approach than the previous cases. This time, observer objects are constructed inside subject objects. This enforces each observer thread to be started and joined inside a different object, providing larger number of object constructions during runtime. Additionally, subject-observer groups run more isolated in this case thus need less synchronization effort. Moreover instead of enforcing objects to be scheduled on static cores, a set of candidate cores are provided to operating system for each object. Hence a hybrid CAWS-CFS approach is used versus CFS this time.

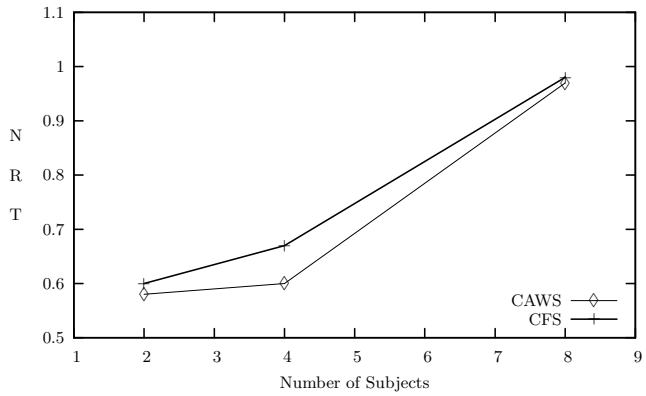


**Fig. 7.** Scheduling 2 observers with different policies

In Figure 7, we can see running times for 2 observers observing different number of subjects. Although observer objects are created and destroyed continuously for each subject, degrading the amount of data reuse during runtime, scheduling the system using a cache-aware policy still provided performance upgrade when compared to CFS.

Finally in Figure 8, the number of objects in the system varies as a whole consisting of different number of subjects and observers. Again using CAWS policy results in a better performance than the default CFS scheduling. We can see that for both examples mixing CFS with CAWS still provided better results than using only CFS. Albeit gaining relatively smaller performance improvements in some of the cases above, it is important to consider that CAWS operates on application level while CFS operates directly on the kernel level. Guiding operating system scheduler based on model driven analysis may also allow us to start tuning an application for a specific processor architecture before the software is implemented.

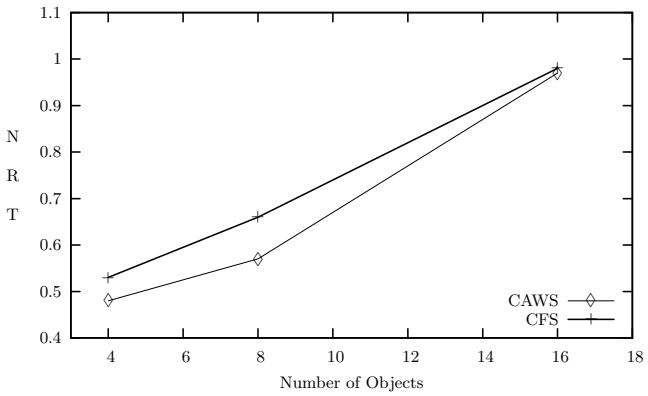


**Fig. 8.** Scheduling many subject-observer tuples with different policies

In our experiments with design pattern implementations, the benefit obtained from cache utilization degrades as the number of objects reach beyond the number of cores in the system. This situation is caused by increased number of cache misses as different objects starts to be swithced on the cores of the system. Nevertheless this problem loses its significance as the number of cores reside in a chip tend to increase over time.

## 6 Cache-Aware Scheduling on a Real-World Case Study

We implement an image filtering software which contains two design patterns that we are going to apply cache-aware scheduling. Image filtering software simply reads in an image(approximately 1.6MBs in size) as a matrix of gray levels for each pixel and convolves it in a parallel way with the filters defined in the software. To create different scenarios, we implemented the software in two different ways by applying different design patterns. Firstly each filter is implemented as a thread and applied separately on the same image. This way filters work similar to the strategies in the strategy design pattern. Secondly three filters are applied successively as a composite filter on the image. This feature is implemented using a Composite pattern(GoF). Each composite filter thread works on a different subsection of the image matrix where each subsection is held in an image buffer. A simplified class diagram(without attributes and methods) that shows the relation among classes of the related software can be found in Figure 9.

For our first case we have applied a scenario to apply different number of filters on the same image in a parallel way. In this scenario the user selects some filters to apply on the image at once. In our software we read the image data for all the filters and apply each filter seperately in a parallel way. This way all the filters use the same image data, enabling the filters to share caches effectively. In

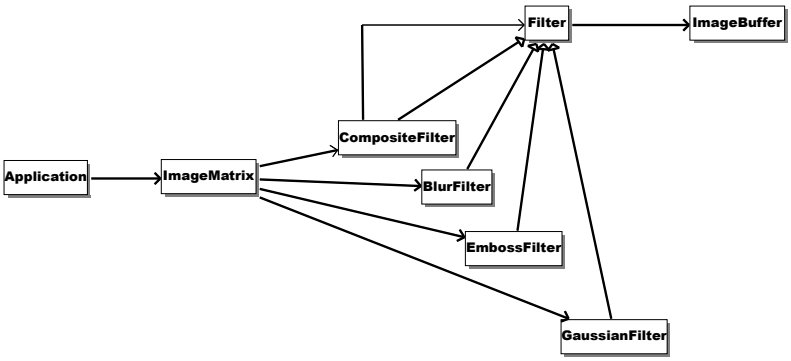


**Fig. 9.** Simplified Class Diagram of Image Filtering Software

Figure 10 we can see a similar performance improvement with our experiments on strategy pattern.

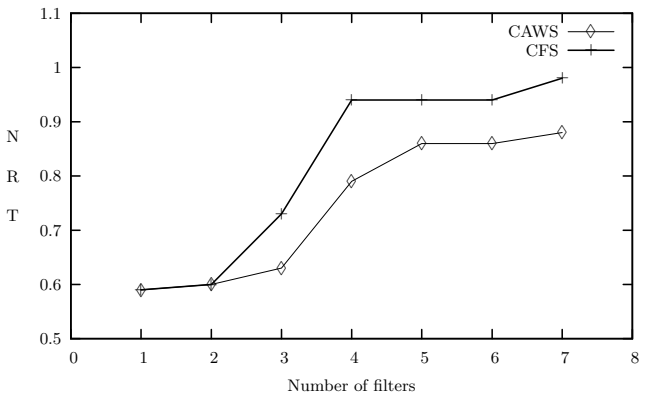


**Fig. 10.** Scheduling image filtering operation using Strategy pattern

For our second scenario we have applied a different approach on image filtering. This time, three filters are chosen to be applied on the image consecutively as a composite filter. This time the image is divided into a number of subsections in order for the filters to run on different parts of the image in parallel. After the image has been divided, chosen filters are applied on each subsection of the image one after another. Subsections are processed in parallel but filters run seqential for each subsection. This way the amount of shared data between filters are decreased and therefore in Figure 11 we can see a smaller performance speedup compared to our first case.

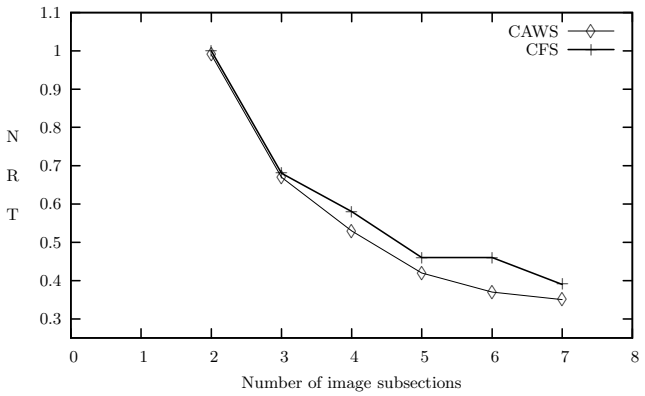


**Fig. 11.** Scheduling image filtering operation using Composite pattern

For both of our cases cache aware scheduling provides an improvement over the performance of the CFS scheduler, especially when the application behaviour involves denser data sharing between objects. An improvement of 10% is obtained for our first case, and for our second case cache aware scheduling didn't degrade the CFS performance. We believe that, for a scheduler that operates on the application level, using information from the very early design stage of software development, improving operating system's scheduling performance supports the applicability of cache-aware scheduling.

## 7 Conclusion and Future Work

Our studies on cache-aware scheduling show that considering shared data during scheduling increases the scheduling performance when multicore processors are used. It is important to utilize shared data among software components in guiding the scheduling process, even if it is not always possible to make accurate predictions on data sharing among software components before the system is run.

Our approach uses software models to reason about data sharing among the classes of a software. In our study we have experimented on three different commonly used software design patterns to consider the effect of cache-aware scheduling. We obtained promising results to apply our model-based approach on larger software considering the three important factors (parallelization, data sharing and resource utilization) that effect the overall performance of the system in our case studies. Beside its positive effects on scheduling performance, using a model driven approach may lead us to reason about software design for various core organizations that processors can include in the future.

In our future studies we plan to improve our approach and implement a model based parallelization methodology based on the principles obtained from our experiments in this paper. By using such methodology it can be possible to reason

about parallelization and data sharing during the early design stage of software development. Moreover it can be possible to steer the design/development process to produce more competing designs for parallelization when different processor architectures are used.

## References


1. T. Zangerl, "Optimisation: Operating system scheduling on multi-core architectures." `http://tzangerl.net/doc/MulticoreScheduling.pdf`, 2008.
2. S. Siddha, "Multi-core and linux kernel." `http://software.intel.com/sites/oss/pdfs/mclinux.pdf`, 2007.
3. "MSDN section on windows scheduling." `http://msdn.microsoft.com/en-us/library/ms685096%28VS.85%29.aspx`.
4. Oracle, "Solaris 11 programming interfaces guide." `http://docs.sun.com/app/docs/doc/821-1602/psched-23069?a=view`.
5. S. Kim, D. Ch, and Y. Solihin, "Fair cache sharing and partitioning in a chip multiprocessor architecture," in *In IEEE PACT*, pp. 111–122, 2004.
6. D. Tam, R. Azimi, L. Soares, and M. Stumm, "Managing shared l2 caches on multicore systems in software," in *Workshop on the Interaction between Operating Systems and Computer Architecture*, 2007.
7. D. Tam, R. Azimi, and M. Stumm, "Thread clustering: sharing-aware scheduling on smp-cmp-smt multiprocessors," in *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, (New York, NY, USA), pp. 47–58, ACM, 2007.
8. A. Merkel and F. Bellosa, "Memory-aware scheduling for energy efficiency on multicore processors," in *HotPower*, 2008.
9. J. Ha, M. Arnold, S. M. Blackburn, and K. S. McKinley, "A concurrent dynamic analysis framework for multicore hardware," in *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, (New York, NY, USA), pp. 155–174, ACM, 2009.
10. B. Zhou, J. Qiao, and S. kuan Lin, "Research on dynamic cache distribution scheduling algorithm on multi-core processors," in *E-Business and Information System Security, 2009. EBISS '09. International Conference on*, pp. 1 –4, 23-24 2009.
11. S. Boyd-Wickizer, R. Morris, and M. F. Kaashoek, "Reinventing scheduling for multicore systems," in *Proceedings of the 12th Workshop on Hot Topics in Operating Systems (HotOS-XII)*, (Monte Verità, Switzerland), May 2009.
12. L. Xue, M. T. Kandemir, G. Chen, F. Li, O. Ozturk, R. Ramanarayanan, and B. Vaidyanathan, "Locality-aware distributed loop scheduling for chip multiprocessors," in *VLSID '07: Proceedings of the 20th International Conference on VLSI Design held jointly with 6th International Conference*, (Washington, DC, USA), pp. 251–258, IEEE Computer Society, 2007.
13. R. Fedorova, M. Seltzer, and M. D. Smith, "Cache-fair thread scheduling for multicore processors," tech. rep., Harvard University, 2006.
14. V. Kumar and J. Delgrande, "Optimal multicore scheduling: An application of asp techniques," in *LPNMR '09: Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning*, (Berlin, Heidelberg), pp. 604–609, Springer-Verlag, 2009.

15. T. Ovatman and F. Buzluca, "Software design pattern behavior in shared memory multiprocessor systems," in *International Conference on Computational Intelligence and Software Engineering*, CiSE'09, pp. 1 –4, 2009.
16. B. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.
17. F. Bergenti and A. Poggi, "Improving uml designs using automatic design pattern detection," in *Proc. 12th International Conf. Software Eng. and Knowledge Eng. (SEKE 00)*, pp. 336–343, 2000.
18. G. Antoniol, G. Casazza, M. D. Penta, and R. Fiutem, "Object-oriented design patterns recovery," *Journal of Systems and Software*, vol. 59, no. 2, pp. 181 – 196, 2001.
19. D. Heuzeroth, T. Holl, G. Högström, and W. Löwe, "Automatic design pattern detection," in *Proceedings of the 11th IEEE International Workshop on Program Comprehension*, IWPC '03, (Washington, DC, USA), pp. 94–, IEEE Computer Society, 2003.
20. Z. Balanyi and R. Ferenc, "Mining design patterns from C++ source code," in *19th International Conference on Software Maintenance (ICSM 2003), The Architecture of Existing Systems, 22-26 September 2003, Amsterdam, The Netherlands*, pp. 305–314, 2003.
21. N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. Halkidis, "Design pattern detection using similarity scoring," *Software Engineering, IEEE Transactions on*, vol. 32, no. 11, pp. 896 –909, 2006.
22. J. Dong, Y. Zhao, and Y. Sun, "A matrix-based approach to recovering design patterns," *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, vol. 39, no. 6, pp. 1271–1282, 2009.
23. B. Nichols, D. Buttlar, and J. P. Farrell, *Pthreads Programming*. O'Reilly, 1998.
24. GNU, "Gnu c library's section on limiting execution to certain cpus." `http://www.gnu.org/software/libc/manual/html_mono/libc.html#CPU-Affinity`.