Augustana University

Advancements of Schedulers in Operating Systems:

Contention-Aware Schedulers

Andrew Martens

COSC 310: Operating Systems

Professor Dan Steinwand

10 May, 2024

The operating system is the base software used in all computer systems. It acts as the middleman between the physical hardware and the higher-level software. Simply put, the operating system manages the resources a computer has and allocates those resources to different processes. Modern computers have many types of resources and multiple processes may need to work together creating different interactions. These interactions can grow to be very complex and it is the operating system's job to manage what is going on to ensure everything runs smoothly. Instead of giving a rundown on all of the operating system's parts, this paper will hone in on the process scheduler, covering how the present-day scheduler has been created and innovations that aim to improve future schedulers.

Understanding the scheduler requires background knowledge of many modern computer systems' architecture, specifically the CPU. In the past, a CPU could achieve higher speeds of computation by increasing the clock rate or through a process called superscalar execution. These solutions, while feasible, create a lot of heat, and the cooling technology hasn't kept up with these advancements and would be too expensive for most users to implement. This led engineers to another solution, using multiple cores on a single die (on the same silicon integrated circuit). In other words, these cores act as mini CPUs are all located on the same chip, and are called Chip Multiprocessors (CMPs). This type of architecture is extremely popular and almost all computers running contain CMPs. With this architecture, each core shares certain resources including the last level cache (LLC), memory controller, bus, and prefetch hardware. Figure 1 shows a simple diagram of a multicore processor, notice how each core has a "personal" and shared cache.

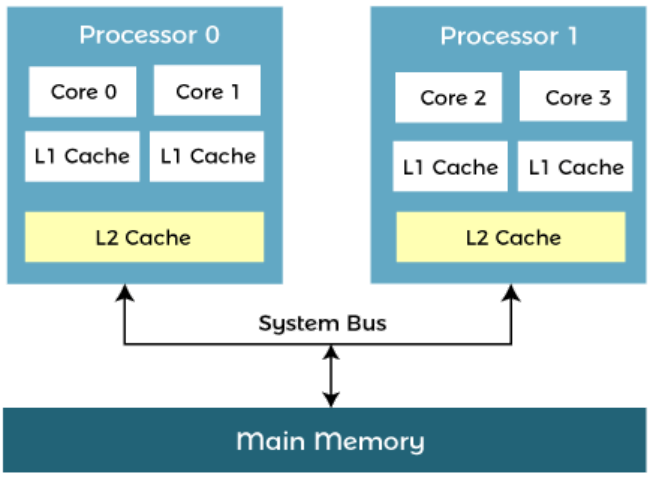

Figure 1: Diagram of a multicore architecture. The CPU has two processors and four cores.

This architecture changes the efficiency of processes in a couple of ways. First, imagine a scenario where two processes are being run in cores 0 & 1 (As seen in Figure 1). One process is very memory intensive while the other uses considerably less memory. Since both CMPs share the last level cache, the process needing more memory could take more space in the shared cache, evicting the other process's memory and reducing the second process's speed. Balancing how the cache is partitioned in this architecture becomes essential and most schedulers don't dynamically allocate shared cache to each CMP. This issue of processes competing for shared resources is called contention and can cause increased runtime if resources aren't utilized correctly on the low-level hardware.

Imagine another scenario where, using Figure 1 as an example, a scheduler would be able to assign processes to any of the four cores. Let's say, for example, there is a job that needs to be completed, and two methods are created to work on this job. Since both processes are working on the same job there is a good chance they might need to interact with each other in some way and it would be more efficient if these processes are put in the same processor to take advantage of the L2 Cache they both share.

However, as mentioned above, this architecture shares more than the cache. Cores can also share the front-side bus, DRAM controller, and prefetch hardware. To accurately determine the best scheduler, knowing how much each shared resource contributes to the total amount of degradation due to contention is important. Researchers at Simon Fraser University experimented with isolating each piece of hardware and measured the percentage of degradation each shared resource caused [5]. Acknowledging the hardware differences in AMD and Intel systems, two separate tests were done on two different processors to determine the degradation breakdown on each architecture. After running six benchmarking applications on Intel's system, these researchers determined the main source of degradation is from the prefetch hardware followed by the cache. The prefetch hardware accounted for ~50% degradation in five of the six applications. For the full breakdown of degradation in the Intel system refer to Figure 2.

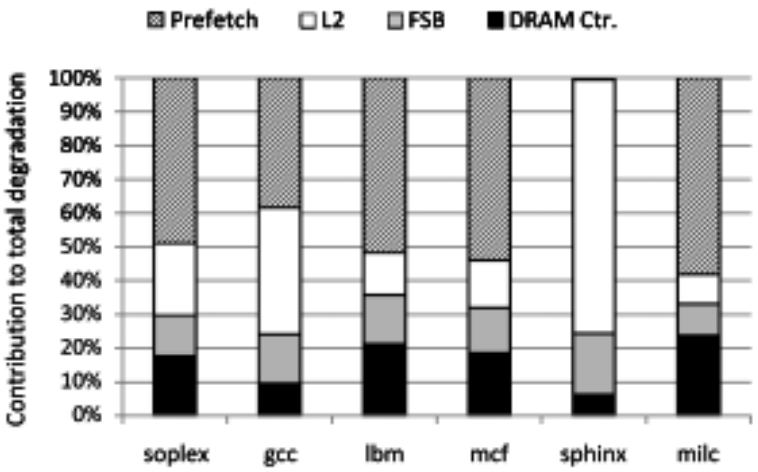

Figure 2: Chart showing degradation breakdown in Intel's CMP architecture [5]

The breakdown in AMD's architecture is astoundingly different which displays the challenge of creating an efficient scheduler for all hardware configurations. Before breaking down the chart, two degradation-contributing factors in the AMD system weren't present in the intel system: interconnect contention and remote latency overhead. Interconnect contention contention occurs when a thread accesses remote memory and competes for inter-processor interconnects with other threads. Remote latency occurs when a thread accesses remote memory and experiences longer wire delays. The results, as seen in Figure 3, show us the LLC doesn't contribute a whole lot of degradation. Instead, the DRAM controller and IC make up the majority of degradation for AMD's chips.

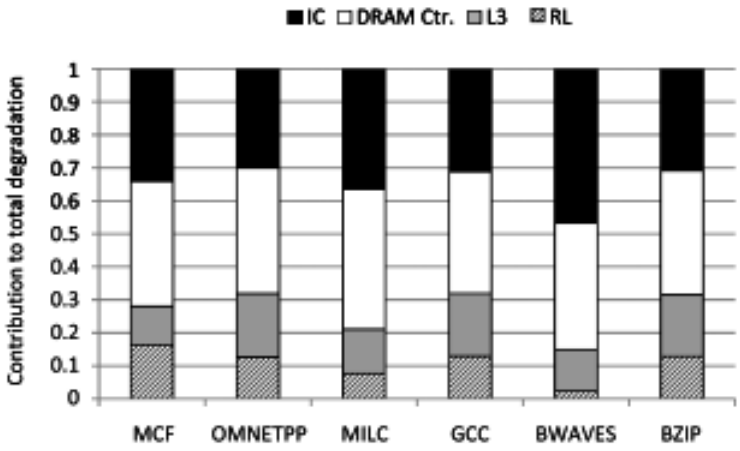

Figure 3: Chart showing degradation breakdown in AMD's CMP architecture [5]

Although the cache never caused the majority of degradation, minimizing cache misses results in fewer trips to memory which reduces degradation of the other shared resources. To get an idea of how much efficiency can be achieved by utilizing the last level cache researchers with the Department of Computer Science at the College of William and Mary, experimented to find the significance of cache sharing on multi-threaded applications. After their experimentation, they concluded that when multi-threaded applications are aware of cache sharing, performance was boosted by up to 53% [2]. These observations show how powerful the CMP architecture can be but identified a large gap between the hardware and software. Meaning software must be developed much further to take advantage of the current architecture, including the creation of a smarter scheduler.

The algorithm current schedulers use is more concerned about load balancing and as it turns out, Linux's Completely Fair Scheduler (CFS) doesn't consider how to utilize the low-level hardware. Instead, the CFS aims to balance the workload between each of the cores. In Linux version 2.6.38 the CFS is the most complicated part of the Linux scheduler located in <kernel/sched/fair.c> containing ~10,000 lines of code. All runnable threads are stored in a red-black tree which balances itself according to each thread's virtual runtime. When it becomes time for the CFS to pick a task to run, the task with the most insignificant virtual runtime is returned. Virtual runtime corresponds to how long a process has been running meaning the task returned is the least executed task. It's also important to note tasks with high priorities accumulate virtual runtime slower than those with lower priorities to give more important tasks more runtime. Since the CFS is a process itself, it periodically runs this load-balancing process as an interrupt in efforts to balance the workload across the cores. Each balance works by taking work from busy cores and moving it to cores with less work. While this algorithm effectively balances CPU workload, more efficiency could be achieved if the low-level architecture was taken into consideration.

To solve this problem a scheduler must be created that can predict how processes might interact with each other. This concept of prediction in computer science often requires some sort of machine learning and this scenario is no different. In theory, a machine learning model could be trained, tested, and then imported into the scheduler to make these predictions. While this may seem like a brilliant idea, there are some challenges to implementing a machine-learning model into the scheduler. The first challenge is to develop an accurate way to train a model. The reason

for this is that training a model within the operating system would carry too much overhead. Secondly, every time a task needs to move it would be run through the model to see which core is the best candidate. This additional computation uses extra time and if the machine learning model isn't accurate enough, the implementation of the model would slow things down. This leaves us with the question, do machine learning models carry too much overhead for them to be productive?

The only way to answer such a question is to implement a machine-learning model into the scheduler to observe the results. Computer scientists at the University of Illinois at Urbana-Champain implemented a machine-learning model inside the Linux kernel's CFS [1]. They decided to make a multi-layer perceptron neural network. This model included 15 input features, one hidden layer with ten nodes, and a single output node. They decided to make such a small model to reduce the amount of overhead they brought into the scheduler. When they trained the model they reached a 99.24% accuracy rate and 0.0955 loss on the validation data. Once they were confident with their model's performance they created a new method called "should_migrate_task()" which used the model to evaluate if a task should migrate. These scientists were confident and optimistic about their model before the official test. The first thing they wanted to measure was if the model's overhead would slow down the scheduler, and after a handful of tests they concluded their model's overhead barely impacted performance. However, when they benchmarked the performance of Linux's CFS and their new scheduler, they found that both schedulers performed very similarly with no significant discrepancies. The computer scientists determined they needed to further develop their model and tweak their inputs. Personally, I wonder if the applications they used to benchmark both schedulers' performance were built to take advantage of cache sharing. This is because applications won't benefit from a smart scheduler if they aren't coded to use the benefits of a smart scheduler and therefore keep a consistent runtime. However, it is hard to say whether inefficient benchmarking code could pose an issue as the paper they published didn't go into much detail about the programs and I am just speculating.

Even though the results from the University of Illinois's study were a bit of a letdown, there is still optimism that machine learning could have a place in the scheduler. Implementing a machine learning model contains so many variables that even if one aspect is lacking the whole model itself won't reap any benefits. Starting with the model, it is possible a different algorithm

could show better performance such as a random forest or gradient boosting algorithm. Then the size of the model comes into play where the number of nodes and how many/what inputs are used also make a huge difference if a model is effective. Beyond the specs of the model, if its training isn't optimal the whole process before and after is essentially ruined which can be done through faulty data or over/underfitting the model. All of these different factors that come with creating a machine-learning model make the results from the previous study a bit cloudy in the sense that we can't conclude a machine-learning model in the scheduler is sub-optimal until more tuning of the model is done. Because of this, there are handfuls of computer scientists who are testing their models in the hopes of finding the perfect model.

Although many people are working towards a machine learning model to create a contention-aware scheduler, others have taken a different approach. Similar to how Linux's CFS distributes tasks using a balancing tree, a new algorithm could be created to determine how to distribute tasks by considering the low-level architecture. For this to be possible there must be a metric used to determine if a task is in an optimal location or if it should be balanced. As mentioned above, there are many shared resources so deciding on a singular metric may not be as straightforward as it seems. Thankfully, much of the shared resource contention stems from multiple threads needing to access main or virtual memory and it's the cache's job to minimize the number of trips to main memory so a common metric for such an algorithm usually has to do with the cache's hit or miss rate.

Computer scientists at Simon Fraser University took the algorithmic approach when trying to create a contention-aware scheduler [5]. Their goal with each algorithm is to minimize the last-level cache miss rate and use this rate to compare algorithms against each other to determine the best one. After much experimentation, these scientists landed on an algorithm they named Distributed Intensity (DI). In this algorithm, all threads are assigned a value corresponding to the number of misses every one million instructions (solo miss rate). Once this rate is calculated, threads are spread evenly across the system so the miss rates are distributed evenly. The thinking behind this distribution is to reduce the number of cache misses to reduce the number of times a thread needs to access the main memory. This algorithm uses memory hierarchy entities when distributing the intensiveness across the system: machine, physical package, chip, and core. Each level is made up of the entities in the previous level. For example, a group of cores makes a chip, a group of chips makes a physical package and physical packages

make up a machine. This hierarchy correlates with how requested data is checked: first, check the small private cache closest to the core. If missing, check the larger cache that is shared between a group of cores on the chip. If the data is still absent, the request goes through the front side bus that is shared with a group of chips called the physical package. Excluding the core, the resource on each level is shared and has the potential for contention. DI tries to even out the miss rate on each level of this memory hierarchy.

The results of DI when compared to Linux's CFS on an Intel machine show this new algorithm outperforms the non-contention-aware scheduler, see Figure 4. As a side note, DIO is another algorithm they came up with but the only difference from DI is how it obtains the miss rate and it contains the same inner workings as DI. While the boost in performance is there, the more important aspect of this new algorithm is how consistent it is. Since Linux's CFS doesn't pay attention to the low-level architecture, there are instances, when scheduled by the CFS, where tasks are "luckily" assigned to the most optimal core. This means a lower percentage of performance from the new algorithms is most likely due to the CFS performing more optimally than DI(O) performing worse.

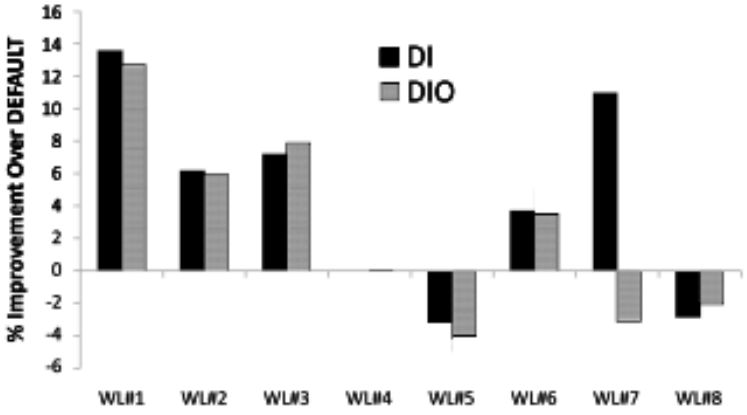

Figure 4: Comparison of workload performance of DI and DIO against Linux's CFS (DEFAULT) on an Intel system

Comparing the performance of these algorithms on an AMD system, we can see similar results. The contention-aware schedulers consistently outperform the default Linux scheduler since they can predict better locations for tasks along with balancing workloads across cores. DI and DIO's best performance was ~10% better than Linux's CFS over the eight workloads and

there was only one instance where DI was outperformed by the CFS. These results back the claim that a contention-aware scheduler will be more consistent and outperform a typical scheduler.

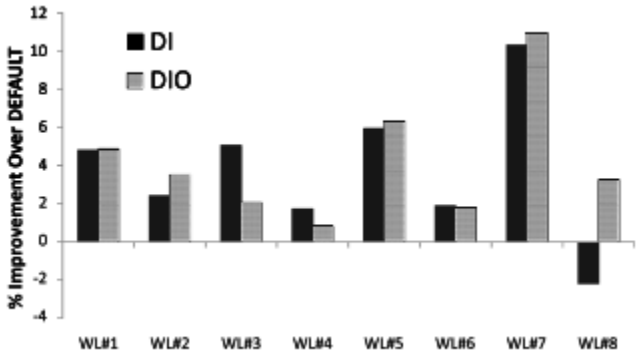

Figure 5: Comparison of workload performance of DI and DIO against Linux's CFS (DEFAULT) on an AMD system

In conclusion, the CMP architecture splits up cores in a way that allows certain resources to be shared. The cores are then contending against each other for these resources. The process each core is working on often impacts the amount of contention and if these processes are configured poorly degradation will occur. This issue led to the concept of a contention-aware scheduler that can balance processes evenly across the cores while also minimizing cache miss rates. The importance of minimizing the cache miss rates (in the shared cache) is because a lot of the contended resources are beyond the scope of the cache and occur when many processes need to access main or virtual memory. When creating a contention-aware scheduler, a machine-learning model or a new algorithm must be implemented. While both approaches are feasible, a machine-learning approach brings a lot of variables to creating a good model such as quality data, the right inputs, and model specifications. Due to these factors, many have decided to create a new algorithm such as the example mentioned above. However, both approaches show optimism in a contention-aware scheduler and this area of OS research is presently being explored in efforts to take advantage of the low-level architecture.

Works Cited

[4] Anderson, G. G. (2013). Operating system scheduling optimization (Order No. 28376045). . (2572713403). Retrieved from https://augie.idm.oclc.org/login?url=https://www.proquest.com/dissertations-theses/operating-system-scheduling-optimization/docview/2572713403/se-2

[2] E. Z. Zhang, Y. Jiang and X. Shen, "The Significance of CMP Cache Sharing on Contemporary Multithreaded Applications," in IEEE Transactions on Parallel and Distributed Systems, vol. 23, no. 2, pp. 367-374, Feb. 2012, doi: 10.1109/TPDS.2011.130.

[3] Jichuan Chang and Gurindar S. Sohi. 2007. Cooperative cache partitioning for chip multiprocessors. In ACM International Conference on Supercomputing 25th Anniversary Volume. Association for Computing Machinery, New York, NY, USA, 402–412. https://doi.org/10.1145/2591635.2667188

[1] Jingde Chen, Subho S. Banerjee, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. 2020. Machine learning for load balancing in the Linux kernel. In Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '20). Association for Computing Machinery, New York, NY, USA, 67–74. https://doi.org/10.1145/3409963.3410492

[5] Sergey Blagodurov, Sergey Zhuravlev, and Alexandra Fedorova. 2010. Contention-Aware Scheduling on Multicore Systems. ACM Trans. Comput. Syst. 28, 4, Article 8 (December 2010), 45 pages. https://doi.org/10.1145/1880018.1880019