1

Green Governors: A Framework for Continuously Adaptive DVFS

Concern over energy efficiency has grown at an increasing rate over the last decade. In response, a great deal of work has gone into developing new CPU frequency governors. The work in this paper aims to alleviate a previously untargeted source of energy waste, memory bound application phases. Classic CPU frequency governors have targeted cores which are identified as underutilized, meaning they spend time running without a program to execute. This work aims to do DFVS during application phases. As cores are often running a program, this offers a much larger percentage of time in which they can try to reduce energy use. This is implemented as a standalone governor for a proof of concept, but this functionality could potentially be implemented by the other governors alongside what they already do.

When it comes to energy reduction, it is always easy to reduce energy by making computation take longer. The name of the game is to attempt to get as much energy reduction as possible while only slowing down applications by a negligible amount. The key insight in this work is that accessing DRAM will take the same amount of time no matter what frequency the core is running at. During phases of applications that are heavily memory bound, such as streaming through an array or iterating through a linked list, the core will spend most of the time doing stalled cycles waiting on the system to bring in memory from DRAM. If the frequency of the core is reduced, then it is mostly stalled cycles being issued at a lower rate, which will not increase the time to completion of that task by much. Since power usage scales quadratically with frequency, running a core at half the power only takes one quarter the power, offering large increases in energy efficiency.

As DVFS has become a standard practice in most systems, much work has gone into making frequency changes happen quickly, generally in the order of tens of microseconds. This is however still enough of an overhead that a smart model must be created for deciding when to

do this DVFS so that the system is not switching back and forth, essentially spending most of its time doing frequency changes. They create models to optimize for the EDP (energy delay product) and ED^2P (energy delay squared product), the latter of which places greater emphasis on not adding any delay to the application and only attempting to save energy is the total savings would be much greater than the expected slowdown. Through the use of hardware CPU performance counters, they can track information such as how many stalled cycles or LLC misses a program has. They create models which work off of either of those counters since LLC misses and stalled cycles have a strong correlation, as most stalled cycles happen when waiting on DRAM which only happens after a LLC miss.

As with much systems work, they use past application behavior to predict future behavior. While this is the best predictor we currently have, there are some issues with this. This is a very good prediction scheme if the interval rate at which the application is profiled is much shorter than the application phase length. The interval at which they profile is 10ms. This is much too large as most applications will switch behavior at a shorter interval than this. This led to very good results on benchmarks which had very long phase lengths, but very bad results on benchmarks which switched behavior more quickly. This is the reason this work was not adopted into the current CPU frequency governors. I believe 10ms was probably chosen because they were using a kernel compiled at 100hz with a timer that went off every 10ms. A low-latency kernel could have been compiled to do this at a 1ms interval. Additionally, there are ways to do this profiling at a much smaller interval (any number of cycles). While this does incur more overhead with a smaller interval length, profiling at an interval of 100 microseconds would still only incur about 0.1% overhead. As the frequency changes are only a few tens of microseconds, the frequency switching could happen at a much finer granularity than is done in this work, offering even more energy reduction.

Lucky scheduling for energy-efficient heterogeneous multi-core systems

As general purpose computing has seen diminishing returns in performance and energy efficiency, many systems are moving toward heterogeneity to offer differing performance characteristics based on application and system needs. Heterogeneous processor systems include cores running the same ISA that operate with differing power and performance characteristics. They generally include big cores which have high performance and little cores which have better energy efficiency. Tasks have differing resource needs, motivating the need for new management techniques any time heterogeneity is introduced into a system.

The work presented in this paper is a simple, elegant method of scheduling on these asymmetric-core systems using previously existing techniques. The important insight into scheduling on these systems is that not all tasks run as efficiently on different cores. Compute-bound tasks run more efficiently on big cores and memory-bound tasks run more efficiently on little cores. Previous work has been done on bias scheduling which schedules tasks on the cores in which they will most efficiently execute. Aptly named, bias scheduling creates a bias, in that some tasks will always get powerful cores and others will always get cores that run slowly in comparison. For workloads in which you must wait until the last task finishes, this can be hurtful to the makespan. Other work has been done in heterogeneous aware scheduling which aims to give a fair share of the big cores to all of the tasks, while not caring about the efficiency achieved by the applications on particular cores.

This work can be seen as a hybrid of the two approaches. Hardware CPU performance counters are tracked for tasks as they run. Counters such as LLC misses and IPC are used to classify tasks as memory-bound (low IPC and high LLC misses) or compute-bound (high IPC and low LLC misses). The magnitude of these metrics can determine the degree to which a task inhibits the behavior of that class and consequently, how efficient it will run on one core or the other. Where this diverges from bias scheduling is that it does not always schedule the tasks on

their most efficient core. Instead, a lottery mechanism is used to determine which tasks will be allowed to run on the big cores. More tickets are assigned to the tasks that exhibit higher degrees of compute-bound intensity. This allows tasks to be mapped to the cores they will run on most efficiently, while not starving any of the tasks from being able to use the big cores.

Because this scheduler does not starve any tasks of the big cores, the total makespan for the workload is decreased, offering more energy savings than bias scheduling can achieve. SPEC 2006 benchmarks were used and lucky scheduling was able to achieve an average Energy Delay Product (EDP) that improved upon bias scheduling by 39%. In contrast, big core fair share scheduling was only able to achieve an EDP of 16% better than bias scheduling.

User-Centric Energy-Efficient Scheduling on Multi-Core Mobile Devices

The work presented in this paper revolves around the kernel taking into account the role of user interaction in making decisions about task management. Modern Android system use the Completely Fair Scheduler (CFS) which is the standard scheduler for most Linux kernels. While it works great on desktops and servers, it is lacking in how well it is suited for mobile devices. This work aims to match scheduling decisions with the needs of the user of a mobile device.

One problem that they identify with CFS is inherent in its purpose, that it treats all tasks fairly. With mobile phone use, the foreground application has differing needs from background applications and therefore should not be treated equally. The scope of this extends beyond mobile devices to desktop environments in which users are interacting with an application, but this work focuses on mobile devices. Because the application that is being interacted with has more strictly defined latency needs, it can be given a prioritized share of resources, in this case the processor. This work could be extended to prioritize other resources such as cache, main memory, network bandwidth etc. Background tasks will necessarily have their computation share diminish, increasing the latency, but if a task is in the background then this will not be an important effect.

To achieve this, they classify tasks into three categories: high, medium, and low sensitivity. High sensitivity tasks are those with strict latency needs because they are being interacted with in the foreground. Medium sensitivity is for system tasks and foreground tasks which are not being interacted with. Low sensitivity tasks are those which are running in the background and as such they are not sensitive to increased latency. Tasks inherit their classification from the task that spawned them to ensure that priority inversion is not an issue.

To achieve the needs of these classifications, the authors use the priority system that is built into the Linux kernels preexisting scheduler, CFS. CFS gives a larger timeslice to a

process with a higher priority. By assigning high priority to the high sensitivity tasks and low priority to the low sensitivity tasks, CFS ensures that the computational needs of the foreground application are given priority. Additionally, although the authors failed to mention this, the load balancing mechanisms in Linux take into account the process priorities and their load is increased with a higher priority, meaning high priority (high sensitivity) tasks will be spread across the processors, giving them better interactivity.

While the total computation including the background tasks can end up taking longer, the computation of the foreground task is increased. What this means varies by applications but it can mean ending sooner or having faster framerates. Since the background applications are not sensitive to latency, they are able to coalesce on a core and the governor can do DVFS more aggressively, improving energy efficiency greatly. In comparison to CFS+On Demand Governor (CFS+OD), their system reduces energy consumption anywhere from 9% to 37%. The experimental analysis could have benefitted from testing a more thorough and diverse mix of application, but overall this paper made great insights about use interaction in scheduling.

LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation

The recent increase in complexity of modern systems due to the presence of heterogeneous hardware devices has become an issue for modern OS kernels. With so many different pieces of hardware that can be more efficiently utilized in differing ways, it is hard for kernels to optimize how a given subsystem works. This works presents the SplitKernel architecture. This splits the kernel, breaking off the functionality that controls various subsystems to the hardware that actually implements that functionality.

The architecture is based off of a network-like design where the hardware parts communicate with the other pieces of hardware through their respective kernels. These kernels for a piece of hardware all called monitors. As not all pieces of hardware are able to run general code, the monitors can be implemented by adding in a cheap general purpose CPU or a higher performance piece of hardware like an FPGA or an ASIC that is able to run the required part of the kernel for the monitor.

With kernel functionalities being optimized by the subsystem that needs to perform that function, they are able to achieve increased performance. In addition, they are able to develop more fault-tolerant systems that are able to recover from disaster more effectively. Kernels in charge of a specific piece of hardware can go down and be restarted without affecting the other monitors in the system. One great use case of this that they fail to mention is the ability to scale up a particular resource by replacing that hardware with more or higher performant pieces of hardware while not affecting the other resources.

When tested, runtime was heavily negatively affected by the SplitKernel architecture. Workloads received a 0.8X to 3.2X slowdown in comparison to being run on a similar system with the same hardware setup but a monolithic kernel. There is some overhead in communication costs which are unavoidable, but much of this is due to the fact that LegoOS was created by a group of four people, whereas monolithic kernels such as Linux have had

thousands of developers. While the actual OS that was created may not be of practical use, the motivation for this type of architecture and much of the theory that went into is sound and will probably see use in an architecture that stems from this.

CRUISE: Cache Replacement and Utility-aware Scheduling

Caches have bridged the gap between registers and main memory, offering much improvement in performance. Application performance can vary greatly depending on how efficiently it is able to use its cache. When multiple applications are co-located on the same core, cache efficiency is taken out of the hands of the application developer. When tasks on the same core or tasks on different cores with a shared cache are both heavily using the cache, they often negatively affect each other by evicting the cache lines that the other application needs to use. This work aims to solve these co-location issues at the system level through intelligent scheduling.

The approach taken was to build a classification system which tracks performance counters of tasks and classifies them into four different behaviors of cache usage. Scheduling logic is then developed to co-locate tasks intelligently based on the interactions between the different cache behaviors. Additionally, the authors take into account the replacement schemes in the underlying cache, allowing this to work in caches that are LRU-managed, offering cache based on demand, or DRRIP-managed, offering cache based on utility.

By tracking hardware CPU performance counters related to L1 and LLC references and misses, they are able to classify applications into the following groups: Core Cache Fitting (CCF), LLC Thrashing (LLCT), LLC Fitting (LLCF), and LLC Friendly (LLCFR). CCF applications have a working set size that fits entirely in the private L1 cache of a core. These do not receive any benefit from their LLC. LLCT applications have a working set size that is much larger than the LLC. This includes streaming applications. They degrade the performance of any task that is sharing a LLC as they will evict much of the cache. LLCF applications have a working set size that is larger than the private cache but near the size of the LLC. In order to perform well, they need to have most of the LLC to themselves. LLCFR applications benefit from LLC resources and their performance scales with the amount of LLC that they are given.

Based on these classifications, co-scheduling decisions can be made that are more efficient that would otherwise happen in a modern system which would mainly just do load-balancing among cores. Since LLCT applications will trash the shared cache, these applications are co-scheduled with other LLCT applications and also CCF applications, since neither of those require a LLC for performance. CCF applications can actually be spread across all available cores, since they will not be negatively affected by the LLC behavior of any other applications. LLCF applications are best co-scheduled with CCF applications which don't trash their cache. If there are no available CCF applications to schedule with LLCF, then LLCF should be scheduled as LLCT since that is how it will behave. Once CCF, LLCT, and LLCF applications have been scheduled, LLCFR applications are scheduled wherever they can be (attempting to not schedule with LLCF), since LLCFR will take advantage of whatever amount of cache it is able to. This assumes the classic LRU-managed cache scheme. If a DRRIP-managed scheme is used, LLCT applications will not receive use of the LLC, and as such they can be spread out in a similar manner to CCF applications since they will not trash the LLC.

Their experimental methodology was extensive. Many people use only a few applications which work well for their research. They identify 15 spec CPU benchmarks which have different cache behaviors, then they run test on all 15 choose 4 sets of applications. Across 1,365 different workloads, these scheduling policies out-perform distributed intensity or random scheduling, coming close to the optimal schedule based on trying every scheduling combination. They test on 4 and 8 core setups with varying cache sizes and conclude that this approach scales well with both. This type of resource contention alleviation will be more and more important as systems become more and more complex. This work does a great job identifying how classifications of resource use can be used to intelligently balance resource needs at the system level.

Siena: Exploring the Design Space of Heterogeneous Memory Systems

As supercomputers grow in size, the cost of the memory and the cost in power to run the memory is increasing greatly. Conventional memory technologies such as DRAM are approaching limits in terms of density, power, and cost. This has motivated a move toward heterogeneous memory systems that balance the performance characteristics of multiple types of memory. Siena offers simulation to test the trade-offs of different memory designs as it is not possible to implement and test all of these designs at the scale that is required. Siena offers simulation at a much faster speed than existing simulators. Through the use of these simulations, desirable hybrid memory configurations are achieved and the interplay between these memory configurations and different application behaviors are brought to light.

Through the use of an analytical compiler, they are able to extract information about application behavior (computation, memory access, depencies, etc…) which they convert into a domain specific language to run on their simulators. For accuracy, they add optional specifics that they know about their application. This requires an in depth knowledge of the applications in which someone who would be using this simulator is testing. This is not a problem for groups designing supercomputers as they must have intimate knowledge of the applications they will be running in order to effectively design the nodes and interconnects, but this would be a lot to ask of most application developers.

Through use of the simulator, they test the effectiveness of integrating hpm as a replacement for some of the DDR4 DRAM. They test both vertical (hpm as a cache for dram) and horizontal (data can be placed on either hpm or dram or nvm) integrations and see how application specifics can determine the performance of the memory system. They find that vertical organizations are better when an application receives a lot of benefits from a fixed small amount of hpm. In workloads that show low utilization of hpm, horizontal organizations are preferred, although they are much more complicated to manage. In performing tests on

horizontal systems, they found that the slow memory could damage the performance of the fast memory as pending requests would block requests to fast memory.

In horizontal implementations, they tested many different configurations in which they placed specific data structures in different types of memory. Where the data was placed had a huge impact on the runtime. This was done with a simple program which did not have many data structures. In complex programs, the number of combinations of which memory to place data in is too great. An application programmer cannot manage this complexity and it is not feasible to brute force the solution from running simulations on all possible combinations. This necessitates a dynamic approach in which the kernel tracks page usage at runtime and allocates the data in the most efficient memory itself, possibly migrating it. This abstraction from the programmer is important and should be explored.

The integration of heterogeneous memory systems will be important in more than just supercomputing. As more areas are starting to focus on energy reduction, hybrid main memory systems will most likely be integrated into mobile, desktop, and server machines. The ability to use pre-made memory models to test is very useful. Some skepticism is warranted about how accurate the simulation could be based on analyzing the code offline and building a skeleton of the functionality which may not capture intricate details which affect the performance. Adding in additional information about a task is also a lot of work. If a simulator like this could run the real applications, that would be of immense use.