Improving Cloud Application Performance with Simulation-Guided CPU State Management

Mathias Gottschlag
Karlsruhe Institute of Technology
mathias.gottschlag@kit.edu

Most modern scale-out applications are complex applications and frequently consist of multiple processes with a very large code working set. These applications are therefore highly processor front-end limited on most current server CPUs: The L1 and L2 caches are small compared to the working set [4], so the workload frequently misses the L2 cache. While most of these misses hit the L3 cache, the L3 cache is large and therefore has high access latency. Due to limited memory-level parallelism, the misses therefore significantly reduce the performance of the system [4] (e.g., many workloads spend half of all cycles waiting for caches[6]).

Several methods to reduce these pipeline stalls have been proposed, either in the form of adapted processor organization with leaner cores and smaller L3 cache [9], or in the form of improved instruction prefetching and branch prediction [7]. However, these solutions generally involve significant hardware changes compared to existing CPU designs. Also, being implemented in hardware, the solutions are usually less flexible compared to software-only techniques: For example, Kanev et al. show that many cloud applications frequently experience periods of more intense computation with higher extracted instruction-level parallelism [6]. Therefore leaner cores might offer significantly reduced performance for these workloads. More advanced hardware prefetchers, on the other hand, utilize on-chip tables of fixed size and therefore do not scale well to arbitrary code working set sizes, whereas the working set of many workloads continues to grow [6].

Software prefetching solutions, such as profile-guided *software call graph prefetching*[2], are more flexible and can be used on existing hardware. These solutions prefetch parts of the working set of the application into caches near the CPU core (L1 or L2 cache) in advance of the corresponding accesses, thereby reducing pipeline stalls.

Existing solutions do not utilize the full potential of software prefetching, though, as they are mostly designed to add prefetching code to an application at compile time. Software prefetching has to be carefully timed and should not load more data than necessary[8], because prefetch instructions compete with the workload for both CPU resources and memory throughput. In particular, our experiments have shown that data should not be prefetched if the resulting cache miss would not cause any significant pipeline stalls. Therefore, existing software solutions could likely benefit from more accurate cache miss cost estimation available at runtime. For example, existing solutions do not provide adequate prefetching for the OS itself, as cache misses in the OS depend on the size and location of the application working set. Because the OS has to provide good performance for a wide range of use cases, it cannot be fitted with application-specific prefetching code at compile time. Finally, runtime approaches can utilize OS knowledge (e.g., knowledge about scheduling or communication patterns) to reduce the effect of prefetching on the application's memory throughput.

Research Questions

With our work, we plan to answer the following research questions:

- How can the operating system collect enough information about the application for efficient software prefetching? In particular, the system might require information about memory access patterns and about accesses that are likely to miss the cache.
- At which points in the code of the operating system or of the application should software prefetching be triggered?
- How can efficient prefetching be implemented for parts of the CPU that are not accessible via dedicated prefetch instructions (e.g., instruction or micro-op caches)?

Approach

To answer the questions above, we plan to build a software prefetching solution for server applications that allows us to explore a number of potential mechanisms. In particular, we propose a design that utilizes simulated caches to identify the important data that has to be prefetched, and that uses available information about communication patterns in order to hide prefetching costs.

Simulation Our experiments have shown that data should not be prefetched if the resulting cache miss would not cause any significant pipeline stall. Existing CPUs do not contain any hardware facilities that provide sufficient information about memory accesses leading to pipeline stalls. To identify which data has to be prefetched, we therefore propose a

design where the targeted network application and its system calls are temporarily executed on a simulated processor for the duration of a single network request. The short simulation duration enables the use of simulated caches and hardware prefetchers in order to identify accesses that are likely to stall the pipeline.

Utilizing CPU Idle Time As described in [2], the call graph of the application can be used to determine the points in the application at which prefetching shall be performed. Prefetching while the application is running comes at a cost, though, due to the cost of prefetch instructions.

As server systems are usually partially idle to be able to process incoming requests with acceptable latency (e.g., [5] reports average utilization below 50% for the Google cloud), parts of the cost of prefetching can be hidden: As the operating system has knowledge about communication patterns of the system, it can predict the next process whenever the system is idle and can start to prefetch the working set of that process, thereby reducing overall prefetching cost. In situations where the system is fully utilized and for cache misses for which prefetching at idle time would be too early, the system can fall back to a variant of call graph prefetching[2].

Prefetching Mechanism We propose the use of existing prefetching instructions for software prefetching of both data and code working sets. Most existing CPUs do not provide any direct method to load data into the L1 instruction cache. The L2 cache, however, is unified, has a significantly lower latency than the L3 cache and is accessible through existing prefetching instructions. Instruction cache misses that hit L3 or external memory are often the most important source of avoidable pipeline stalls. While prefetching into the L2 cache cannot completely prevent these pipeline stalls, we expect a significant performance improvement.

Related Work

Architectures for server and scale-out applications: It has been shown that traditional server applications and cloud applications cannot exploit the full potential of modern wide out-of-order CPUs due to large numbers of stall cycles in the CPU front-end [1, 4]. As a result, a number of techniques have been developed to improve the performance of these applications, including more efficient instruction cache and branch prediction prefetchers [7] and multi-core architectures with lean out-of-order processors optimized for cloud applications [9]. In contrast to these solutions, our proposed design is usable on existing CPUs and can hide prefetching latency whenever the CPU is temporary idle.

Call graph prefetching [2] is a code prefetching approach that can be implemented completely in software and is based on profiling to generate a call graph of the application. Whenever a function calls another function, the caller is extended to unconditionally prefetch the first cache lines of the callee. In contrast, we plan to employ cache simulation during profiling in order to identify those memory accesses that cause significant stall cycles, as our experiments have shown that it is essential to minimize the number of prefetched memory blocks to obtain good performance.

Speculative precomputation [3] is another software approach which uses profiling to identify the memory instructions responsible for most cache misses. In contrast, our approach tries to identify the memory locations that are frequenty missed. The resulting prefetching code directly accesses these locations, without any additional address calculation which would cause further overhead.

Conclusion

Most scale-out and server workloads are limited by memory access latency, due to their large working sets. Previous approaches to reduce the number of cache misses are usually either hardware modifications with limited flexibility and significant hardware deployment cost or compiler modifications which suffer from limited information about runtime application behavior. We plan to investigate methods that use OS knowledge about scheduling to implement efficient software prefetching and prevent expensive pipeline stalls. Our proposed design uses simulation to establish the required information about the system, and we propose prefetching during idle times to reduce the cost of prefetching.

References

[1] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and David A. Wood. DBMSs on a modern processor: Where does time go? In *VLDB'99*, pages 266–277, 1999.

[2] Murali Annavaram, Jignesh M. Patel, and Edward S. Davidson. Call graph prefetching for database applications. *ACM TOCS*, 21(4):412–444, 2003.

[3] Jamison D Collins, Hong Wang, Dean M Tullsen, Christopher Hughes, Yong-Fong Lee, Dan Lavery, and John P Shen. Speculative precomputation: Long-range prefetching of delinquent loads. In *ISCA'2001*, pages 14–25. IEEE, 2001.

[4] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In *ACM SIGPLAN Notices*, volume 47, pages 37–48. ACM, 2012.

[5] Peter Garraghan, Paul Townend, and Jie Xu. An analysis of the server characteristics and resource utilization in google cloud. In *IC2E'2013*, pages 124–131. IEEE, 2013.

[6] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. In *ISCA'2015*, pages 158–169. IEEE, 2015.

[7] Cansu Kaynak, Boris Grot, and Babak Falsafi. Confluence: unified instruction supply for scale-out servers. In *MICRO-48*, pages 166–177. ACM, 2015.

[8] Jaekyu Lee, Hyesoon Kim, and Richard Vuduc. When prefetching works, when it doesn't, and why. *ACM TACO*, 9(1):2, 2012.

[9] Pejman Lotfi-Kamran, Boris Grot, Michael Ferdman, Stavros Volos, Onur Kocberber, Javier Picorel, Almutaz Adileh, Djordje Jevdjic, Sachin Idgunji, Emre Ozer, et al. Scale-out processors. In *ACM SIGARCH Computer Architecture News*, volume 40, pages 500–511. ACM, 2012.