Reinventing Scheduling for Multicore Systems

Silas Boyd-Wickizer, Robert Morris, M. Frans Kaashoek (MIT)

Abstract

High performance on multicore processors requires that schedulers be reinvented. Traditional schedulers focus on keeping execution units busy by assigning each core a thread to run. Schedulers ought to focus, however, on high utilization of on-chip memory, rather than of execution cores, to reduce the impact of expensive DRAM and remote cache accesses. A challenge in achieving good use of on-chip memory is that the memory is split up among the cores in the form of many small caches. This paper argues for a form of scheduling that assigns each object and its operations to a specific core, moving a thread among the cores as it uses different objects.

1 Introduction

As the number of cores per chip grows, compute cycles will continue to grow relatively more plentiful than access to off-chip memory. To achieve good performance, applications will need to make efficient use of on-chip memory [11]. On-chip memory is likely to continue to come in the form of many small caches associated with individual cores. A central challenge will be managing these caches to avoid off-chip memory accesses. This paper argues that the solution requires a new approach to scheduling, one that focuses on assigning data objects to cores' caches, rather than on assigning threads to cores.

Schedulers in today's operating systems have the primary goal of keeping all cores busy executing some runnable thread. Use of on-chip memory is not explicitly scheduled: a thread's use of some data implicitly moves the data to the local core's cache. This implicit scheduling of on-chip memory often works well, but can be inefficient for read/write data shared among multiple threads or for data that is too large to fit in one core's cache. For shared read/write data, cache-coherence messages, which ensure that reads see the latest writes, can saturate system interconnects for some workloads. For large data sets, the risk is that each datum may be replicated in many caches, which decreases the amount of distinct data stored on the chip and may increase the number of DRAM accesses.

Even single-threaded applications can use memory resources inefficiently. For example, a single threaded application might have a working set larger than a single core's cache capacity. The application would run faster with more cache, and the processor may well have spare cache in other cores, but if the application stays on one core it can use only a small fraction of the total cache.

We advocate use of a scheduler that assigns data objects to on-chip caches and migrates threads amongst cores as they access objects, in a manner similar to NUMA systems that migrated threads among nodes. This migration can decrease memory access times, since it brings threads close to the data they use, as can also decrease duplication of data among the many core caches, allowing more distinct data to be cached. We refer to a scheduler that moves operations to objects as an O^2 scheduler.

This position paper makes the following contributions: (1) it argues that scheduling at the level of objects and operations (O^2) is important; (2) it presents some challenges in designing an O^2 scheduler; (3) it presents a preliminary design for an O^2 scheduler, which we call CoreTime; and (4) it presents some evidence using a synthetic workload that even on today's commodity multicore processors an O^2 scheduler can improve performance. The benefits for real workloads are likely to be realized only as the number of cores per chip grows, and with it the difference between the total amount of data those cores can consume and the limited throughput to off-chip memory.

2 O^2 Scheduling

The pseudocode in Figure 1 gives an example of a workload that can perform better under an O^2 scheduler than traditional thread based schedulers. Each thread repeatedly looks up a file in a randomly chosen directory. Each directory is an object and each search is an operation. Workloads like these can be a bottleneck when running a Web server [14].

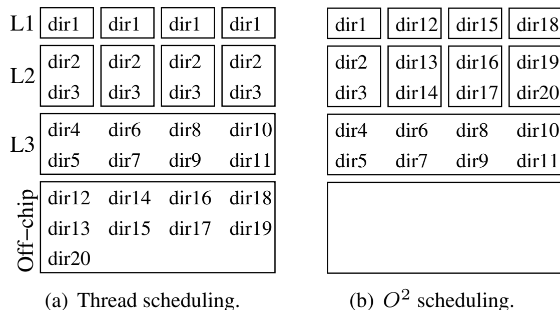
On a system with four cores, a thread-based scheduler will schedule each thread to a core. Each core will independently cache recently used directories. If the working set is less than the size of one core's cache, performance will be good. If the working set is larger than a single core's cache, then all threads will likely spend a lot of time waiting for off-chip DRAM. This will be true even if the working set could fit in the total on-chip memory. Thread clustering [12] will not improve performance since all threads look up files in the same directories.

An O^2 scheduler, on the other hand, will partition the directories across all the caches to take advantage of the total on-chip memory. The O^2 scheduler will migrate each search to the core caching the relevant directory. If

```
thread(void) {
    while(1) {
        dir = random_dir();
        file = random_file();
        /* Search dir for file */
        for (i = 0; i < dir.n; i++)
            if (dir.file[i].name == file)
                break;
    }
}

main(void) {
    start(thread, "thread 0");
    start(thread, "thread 1");
    start(thread, "thread 2");
    start(thread, "thread 3");
}
```

Figure 1: Pseudo code for an example directory lookup workload. Directories are objects and searches are operations.



(a) Thread scheduling. (b) O^2 scheduling.

Figure 2: The levels of the memory hierarchy from which cores might load directory contents.

the working set of directories is larger than one core's cache and the cost of migration is relatively cheap, the O^2 schedule will provide better performance than the thread based one.

Figure 2 shows the levels of the memory hierarchy from which cores might load directory contents when a system uses thread scheduling and when a system uses O^2 scheduling. Under O^2 scheduling all directory loads are satisfied from on-chip memory, but under thread scheduling about half of the directory loads require off-chip access.

3 CHALLENGES

Developing a practical O^2 scheduler for real workloads (or even simple ones as in Figure 1) faces the following challenges.

An O^2 scheduler must balance both objects and operations across caches and cores. It should not assign more objects than fit in a core's cache or leave some cores idle while others are saturated.

The scheduler must understand enough about the workload to schedule it well; it must be able to identify objects and operations, find sizes of objects, and estimate execution times of operations.

The scheduler must be able to control how objects are stored in caches, even though typical multicore hardware provides little explicit control over where data is cached.

Finally, the scheduler needs an efficient way to migrate threads.

4 CORETIME

CoreTime is a design for an O^2 scheduler that operates as a run-time library for C programs.

Interface: CoreTime relies on application developers to specify what must be scheduled. CoreTime provides two code annotations with which the programmer marks the beginning and end of an operation. They take one argument that specifies the address that identifies an object.

Figure 3 shows how the annotations could be used in the example discussed in Section 2. `ct_start(`o`)` performs a table lookup to determine if the object o is scheduled to a specific core. If the table does not contain a core for o the operation is executed locally, otherwise the thread is migrated to the core returned by the lookup. `ct_start` automatically adds an object to the table if the object is expensive to fetch.

CoreTime annotations provided by developers help reduce the number of objects CoreTime considers for scheduling, and we expect that for many applications they will not be a huge programmer burden. For example, most shared memory multiprocessor applications already use locks (or other synchronization primitives), to protect manipulations of objects that might be shared. The code in such a critical section is likely to be a good candidate for CoreTime annotation too, and compilers could likely insert them automatically.

CoreTime has an alternative interface around method invocations, but it restricts threads to migrate at the beginning and end of a method, which is cumbersome. For example, a programmer might want to migrate only part of method invocation (e.g., the part that manipulates a large member of an object.)

Algorithm: CoreTime uses a greedy first fit "cache packing" algorithm to decide what core to assign an object to. Our choice of algorithm is motivated by the observation that migrating an operation to manipulate some object o is only beneficial when the cost of migration is less than the cost of fetching the object from DRAM or a remote cache. The cache packing algorithm works by assigning each object that is expensive to fetch to a cache with free space. The algorithm executes in $\Theta(nlogn)$ time, where n is the number of objects.

Cache packing might assign several popular objects to a single core and threads will stall waiting to operate on the objects. For example, several cores may migrate threads to the same core simultaneously. Our current solution is to detect performance pathologies at run-

```
thread(void) {
    while(1) {
        dir = random_dir();
        file = random_file();
        /* Search dir for file */
        ct_start(dir);
        for (i = 0; i < dir.n; i++)
            if (dir.file[i].name == file)
                break;
        ct_end();
    }
}

main(void) {
    start(thread, "thread 0");
    start(thread, "thread 1");
    start(thread, "thread 2");
    start(thread, "thread 3");
}
```

Figure 3: Pseudo code from Figure 1 with CoreTime annotations.

time and to improve performance by rearranging objects. As we describe next, we use hardware event counters to detect such pathologies.

Runtime monitoring: CoreTime uses AMD event counters[1] to detect objects that are expensive to fetch and should be assigned to a core. For each object, CoreTime counts the number of cache misses that occur between a pair of CoreTime annotations and assumes the misses are caused by fetching the object. In the example in Figure 3, if each thread performs lookups on more directories than fit in a local cache, there will likely be many cache misses for each directory search. When there are many cache misses while manipulating an object, CoreTime will assign the object to a cache by adding it to the lookup table used by `ct_start`; otherwise, CoreTime will do nothing and the shared-memory hardware will manage the object.

CoreTime also uses hardware event counters to detect when too many operations are assigned to a core or too many objects are assigned to a cache. CoreTime tracks the number of idle cycles, loads from DRAM, and loads from the L2 cache for each core. If a core is rarely idle or often loads from DRAM, CoreTime will periodically move a portion of the objects from that core's cache to the cache of a core that has more idle cycles and rarely loads from the L2 cache.

Migration: When a thread calls `ct_start(o)` and *o* is assigned to another core CoreTime will migrate the thread. The core the thread is executing on saves the thread context in a shared buffer, continues to execute other threads in its run queue, and sets a flag that the destination core periodically polls. When the destination core notices a pending migration it loads the thread context and continues executing. Eventually the thread will call `ct_end`, which saves the thread context and sets a flag that indicates to the original core that the operation is complete and the thread is ready to run on another core.

Implementation: CoreTime runs on Linux, but could be ported to other similar operating systems. CoreTime creates one `pthread` per core, tied to the core with `sched_setaffinity()`, and sets the process scheduling priority to the highest possible using `setpriority()` to avoid being descheduled by the kernel. CoreTime provides cooperative threading within each core's `pthread`.

[1] Other processor have similar event counters.

5 Preliminary Evidence

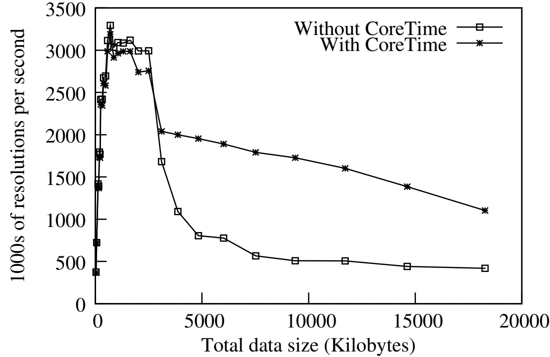
This section explores the performance of CoreTime with a synthetic directory workload on a 16-core AMD system.

Hardware: The AMD system has four quad-core 2 GHz Opteron chips connected by a square interconnect. The interconnect carries cache coherence broadcasts to locate and invalidate data, as well as point-to-point transfers of cache lines. Each core has its own L1 and L2 cache, and four cores on each chip share an L3 cache. The latencies of an L1 access, L2 access, and L3 access are 3 cycles, 14 cycles and 75 cycles respectively. Remote fetch latencies vary from 127 cycles to fetch from the cache of a core on the same chip to 336 cycles to fetch from the most distant DRAM bank. The measured cost of migration in CoreTime is 2000 cycles.

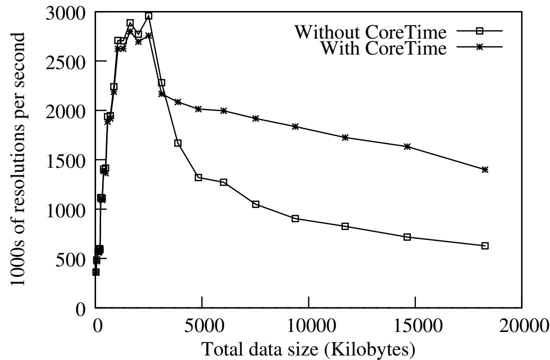
Setup: We measured the performance of CoreTime when applied to the file system using two directory lookup benchmarks. The file system is derived from the EFSL FAT implementation [8]. We modified EFSL to use an in-memory image rather than disk operations, to not use a buffer cache, and to have a higher-performance inner loop for file name lookup. We focused on directory search, adding per-directory spin locks and CoreTime annotations. Each directory is a CoreTime object and each file name lookup is an CoreTime operation. The workloads we focused on involved a thread on each core repeatedly looking up a randomly chosen file from a randomly chosen directory. Each directory contains 1,000 entries, and each entry uses 32 bytes of memory. CoreTime could be expected to improve the performance of these workloads when the total set of directory entries is large enough that it does not fit in a single core's cache.

Result: Figure 4(a) shows the performance of the file system benchmark when it randomly selects a file name to resolve using a uniform distribution. The number of directories varies along the x-axis, which indicates the total size of all directory contents. The y-axis indicates the total number of name resolutions completed per second.

At the extreme left of Figure 4(a) both with and without CoreTime has relatively lower performance. The rea-



(a) File system results for uniform directory popularity.



(b) File system results for oscillated directory popularity.

Figure 4: File system benchmark results.

son is that there are fewer than 16 directories, which restricts the degree of parallel speedup. The threads have to wait for locks both with and without CoreTime.

For total data sizes between about 512 Kbytes and 2 Mbytes, a complete copy of the entire set of directories can fit in each of the four AMD chips' L3 caches. Without CoreTime and with CoreTime perform well, since all lookups operate out of locally cached data.

Once the total amount of data is noticeably larger than 2 Mbytes, a complete copy no longer fits in each of the AMD chips' L3 caches and the performance with CoreTime is between two to three times faster than without CoreTime. CoreTime automatically assigns directories to caches when it detects cache misses during lookups. Without CoreTime, the cores must read directory contents from DRAM or remote caches. With CoreTime, there is no duplication of data in the cache and each lookup is executed on the core that holds the directory in its cache. The total amount of cache space is 16 Mbytes (four 2 Mbyte L3 caches and 16 512 Kbyte L2 caches), so CoreTime can avoid using DRAM until there is more than 16 Mbytes of data. The performance goes down before that point as more data must be stored in the L3 rather than L2.

Figure 4(b) presents the results when number of directories accessed oscillates from the value represented on the x-axis to a sixteenth of that value. We chose this benchmark to demonstrate the ability CoreTime to rebalance objects to achieve good performance. CoreTime is able to rebalance directories across caches and performs more than twice as fast for most data sizes than without CoreTime.

6 DISCUSSION

Although the results from the synthetic benchmark in the previous section are encouraging, there are many open questions.

6.1 Future Multicores

On the AMD system, CoreTime improves the performance of workloads whose bottleneck is reading large objects (*e.g.* scanning directory contents). The improvement possible with CoreTime for other workloads is limited by the following properties of the AMD hardware: the high off-chip memory bandwidth, the high cost to migrate a thread, the small aggregate size of on-chip memory, and the limited ability of the software to control hardware caches. We expect future multicores to adjust some of these properties in favor of O^2 scheduling. Future multicores will likely have a larger ratio of compute cycles to off-chip memory bandwidth and have larger per-core caches. Furthermore, the increasing number of CPU instructions that allow software to control caching [1] is evidence that chip manufactures recognize the importance of allowing software to control caching behavior. These trends will result in processors where O^2 scheduling might be attractive for a larger number of workloads.

The evolution of multicore hardware design may have a substantial impact on the design of O^2 schedulers. Future processors might have heterogeneous cores, which would complicate the design of a O^2 scheduler. Processors might not have global cache-coherent memory and might instead rely on software to manage placement of objects. If this were the case then the O^2 scheduler must be involved in this placement. Also if active messages were supported by hardware this could reduce the overhead of migration.

6.2 O^2 Improvements

The O^2 scheduling algorithm presented in Section 4 is preliminary. For example, it is likely that some workloads would benefit from object clustering: if one thread or operation uses two objects simultaneously then it might be best to place both objects in the same cache, if they fit.

There are also unexplored tradeoffs in the O^2 scheduling algorithm. For example, sometimes it is better to replicate read-only objects and others times it might be better to schedule more distinct objects. Working sets

larger than the total on-chip memory present another interesting tradeoff. In these situations O^2 schedulers might want to use a cache replacement policy that, for example, stores the objects accessed most frequently on-chip and stores the less frequently accessed objects off-chip.

To be able use an O^2 scheduler as the default system-wide scheduler, the O^2 scheduler must track which process owns an object and its operations. With this information the O^2 scheduler could implement priorities and fairness. Of course if there are user-level and kernel-level O^2 schedulers some interface must exist to ensure overall good performance.

Compiler support might reduce the work for programmers and provide a O^2 scheduler with more information. For example, the compiler could add CoreTime-like annotations automatically using relationships between basic blocks. Compilers might also improve the performance of O^2 schedulers by, for example, not loading from and storing to the stack between `ct_start` and `ct_end`. This will decrease thread migration latencies, because a thread's stack will not have to migrate with the thread. Compilers might also infer object clusters statically and convey this information to the runtime to improve performance. In high-level languages, such as Java and Python, it might be possible to implement O^2 scheduling transparently to the developer.

7 RELATED WORK

Most previous multiprocessor schedulers solve a variation of the multiprocessor scheduling problem, which can be stated as "How can a set of threads T be executed on a set of P processors subject to some optimizing criteria?" Scheduling techniques that improve the use of memory resources often use thread working sets as an optimizing criteria. For example, thread clustering algorithms [12, 13] try to improve performance by co-locating threads that have similar working sets to the same core, which can reduce interconnect traffic. Chen et al. [6] investigate two schedulers that attempt to schedule threads that share a working set on the same core so that they share the core's cache. Cho and Jin [7] investigate page migration algorithms such as used in Cellular Disco for achieving better cache locality on multicore processors. Bellosa and Steckermeiser [3] use cache-miss counters to do better thread scheduling.

CoreTime dynamically decides to migrate an operation to a different core, which is related to computation migration in distributed-shared memory systems (e.g., [4, 10]) and object-based parallel programming language systems for NUMA systems (e.g., [2, 5, 9]). These systems use simple heuristics that do not take advantage of hardware counters, and do not consider the technique as part of a general scheduling problem.

8 CONCLUSIONS

This paper has argued that multicore processors pose unique scheduling problems that require an approach that utilizes the large, but distributed on-chip memory well. We advocated an approach based on scheduling objects and operations to caches and cores, rather than a traditional scheduler that optimizes for CPU cycle utilization.

REFERENCES

[1] AMD. Software Optimization Guide for AMD Family 10h Processors, February 2009.

[2] H. E. Bal, R. Bhoedjang, R. Hofman, C. J. a nd Koen Langendoen, T. Rühl, and M. F. Kaashoek. Performance evaluation of the Orca shared-object system. *ACM Trans. Comput. Syst.*, 16(1):1–40, 1998.

[3] F. Bellosa and M. Steckermeier. The performance implications of locality information usage in shared-me mory multiprocessors. *J. Parallel Distrib. Comput.*, 37(1):113–121, 1996.

[4] M. C. Carlisle and A. Rogers. Software caching and computation migration in Olden. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1995.

[5] R. Chandra, A. Gupta, and J. L. Hennessy. Cool: An object-based language for parallel programming. *Computer*, 27(8):13–26, 1994.

[6] S. Chen, P. B. Gibbons, M. Kozuch, V. Liaskovitis, A. Ailamaki, G. E. Blelloch, B. Falsafi, L. Fix, N. Hardavellas, T. C. Mowry, and C. Wilkerson. Scheduling Threads for Constructive Cache Sharing on CMPs. In *Proceedings of the 19th ACM Symposium on Parallel Algorithms and Architectures*, pages 105–115. ACM, 2007.

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

[8] EFSL. `http://efsl.be`.

[9] R. J. Fowler and L. I. Kontothanassis. Improving processor and cache locality in fine-grain parallel comput ations using object-affinity scheduling and continuation passing. Technical Report TR411, 1992.

[10] W. C. Hsieh, M. F. Kaashoek, and W. E. Weihl. Dynamic computation migration in dsm systems. In *Supercomputing '96: Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)*, Washington, DC, USA, 1996.

[11] S. E. Perl and R. L. Sites. Studies of windows nt performance using dynamic execution traces. In *Proc. of OSDI*, pages 169–183, 1996.

[12] D. Tam, R. Azimi, and M. Stumm. Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 47–58, New York, NY, USA, 2007. ACM.

[13] R. Thekkath and S. J. Eggers. Impact of sharing-based thread placement on multithreaded architectures. In *In Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 176–186, 1994.

[14] B. Veal and A. Foong. Performance scalability of a multi-core web server. In *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*, pages 57–66, New York, NY, USA, 2007. ACM.

This material is based upon work supported by the National Science Foundation under Grant No. 0834415 and 0915164.