Abstracting Multi-Core Topologies with MCTOP

Georgios Chatzopoulos[1], Rachid Guerraoui[1], Tim Harris[2] and Vasileios Trigonakis[2*†]

[1]*EPFL*
{first.last}@epfl.ch

[2]*Oracle Labs*
{timothy.l.harris, vasileios.trigonakis}@oracle.com

Abstract

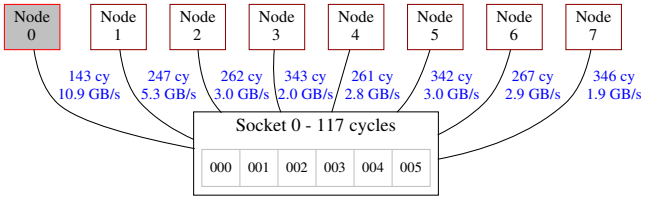
Portability and efficiency are usually antagonists in multi-core computing. In order to develop efficient code, one needs to take into account the topology of the target multi-cores (e.g., for locality). This clearly hampers code portability. In this paper, we show that you can have the cake and eat it too.

We introduce MCTOP, an abstraction of multi-core topologies augmented with important low-level hardware information, such as memory bandwidths and communication latencies. We show how to automatically generate MCTOP using `libmctop`, our library that leverages the determinism of cache-coherence protocols to infer the topology of multi-cores using only latency measurements.

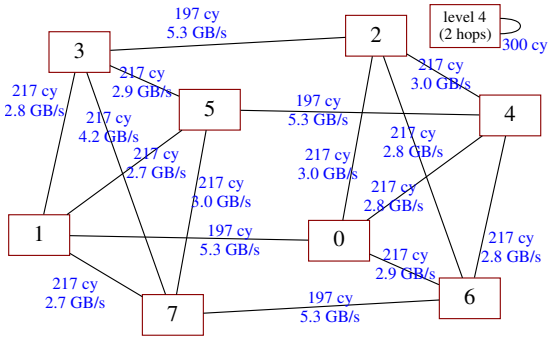
MCTOP enables developers to accurately and portably define high-level performance optimization policies. We illustrate several such policies through four examples: (i-ii) thread placement in OpenMP and in a MapReduce library, (iii) a topology-aware mergesort algorithm, as well as (iv) automatic backoff schemes for locks. We illustrate the portability of these optimizations on five processors from Intel, AMD, and Oracle, with low effort.

1. Introduction

Since 2000, computing systems are becoming more diverse in terms of the numbers of threads per core, cores per socket, as well as the on-chip and off-chip interconnects. This tendency makes the task of developers very challenging, for they need to fine-tune software to the underlying hardware in order to achieve performance (e.g., [12, 18, 19, 30, 37]). Furthermore, optimizing for specific multi-core topologies hinders software portability. In fact, the need for such optimizations raises two main questions: (i) how to harvest and expose multi-core details in software, and (ii) how to fine-tune according to these details, while ensuring portability.



(a) Topology of a single socket, augmented with the intra-socket communication latencies, and memory latencies and bandwidths.



(b) Cross-socket topology, augmented with socket-to-socket latencies and bandwidths. Level 4 represents non-direct links.

Figure 1: Visualization of the MCTOP topology of an 8-socket AMD processor. Both the topology and the graphs are automatically generated by `libmctop`.

Traditionally, developers have been relying on the topology information of operating systems, through libraries, for abstracting the topology of multi-cores (e.g., using `libnuma` [47] on Linux, `liblgrp` [6] on Solaris, or `hwloc` [19]). These libraries offer a topology representation of multi-cores, as well as a companion interface for placing threads (and data). However, the provided representations are low-level and offer only the limited topology view of the operating system. As such, developers do not have access to the performance characteristics of the underlying multi-core processor. Moreover, developers still need to optimize their software for each platform. For example, they need to manually identify the hardware contexts that belong to the

* This project started while the author was interning at Oracle Labs and was completed while the author was at EPFL.

† The authors appear in alphabetical order.

EuroSys '17, April 23 - 26, 2017, Belgrade, Serbia

ISBN 978-1-4503-4938-3/17/04…$15.00

DOI: http://dx.doi.org/10.1145/3064176.3064194

same cores (usually for avoiding them), calculate the best-connected sockets, and consult processor manuals for discovering the actual topology of their multi-core. The result is ad-hoc implementations, tied to the underlying platform.

We present in this paper an easier, more portable approach to optimizing software for multi-cores. We introduce MCTOP, a multi-core topology abstraction of important low-level information, such as communication latencies and memory bandwidths. MCTOP is automatically generated and exposed to software developers by our `libmctop` user-level library. Figure 1 depicts the visual representation of the MCTOP of an AMD Opteron. Of course, a developer could directly use this low-level information to fine-tune her software for this Opteron. For instance, she could decide to use sockets 0 and 1 as they are connected with minimum latency. However, such optimizations—that rely on the specifics of a processor—are not portable. Instead, she could write a *policy* that uses any two sockets (if available) that minimize latency.

MCTOP enables the design of easy, portable, and efficient optimizations using such high-level policies. In turn, these policies make use of the actual numbers included in MCTOP. Essentially, MCTOP allows developers to express high-level semantics that utilize the low-level performance details of multi-cores, thus delivering *portable optimizations*. For instance, using MCTOP, we can easily define policies such as "use one hardware context per core," "use two sockets with maximum bandwidth," or even "use the maximum number of threads, in the two most remote sockets, so that each thread has access to at least 3 MB of LLC."

`libmctop` is based on MCTOP-ALG, our novel algorithm for inferring the topology of multi-cores. MCTOP-ALG relies on two fundamental observations: (i) *cache-coherence protocols are deterministic in the absence of contention*, and (ii) *communication latencies characterize the topology*. These observations are in accordance with the network view of multi-cores that has been proposed for OS design [12, 13, 70]. MCTOP-ALG leverages these two observations by collecting accurate core-to-core communication latencies, which are used to infer the topology of the processor. On top of this topology, `libmctop` collects additional low-level measurements, such as cache latencies and memory latencies/bandwidths. The end result is an automatically-generated MCTOP representation of the multi-core.

We argue that MCTOP-ALG's measurement-based approach is superior to loading multi-core topologies from the underlying OS or hardware (e.g., using `CPUID`) for various reasons: (i) *portability*—collecting measurements is almost identical on any architecture or OS, unlike reading topology info from the OS or the hardware; (ii) *forward/backwards compatibility*—measurements do not depend on the OS version; (iii) *correctness*—numbers do not lie, while the OS can be misconfigured[1]; (iv) *extensibility*—independence from the information that vendors do or do not expose; and (v) *accuracy*—a measurement-based approach collects accurate low-level measurements that we need in MCTOP.

We illustrate portable optimizations with MCTOP through four examples on five processors from Intel, AMD, and Oracle. First, we automate backing off in locking using the latencies of MCTOP. Our optimized spinlocks deliver up to 39% average throughput improvements. Second, we design a topology-aware mergesort algorithm that builds a cross-socket merge tree on top of MCTOP. Our algorithm is 17% faster on average than the parallel sort algorithm of the C++ standard library which is topology agnostic.

Furthermore, we design a thread placement library, called MCTOP-PLACE, on top of MCTOP (`libmctop`) and use it in optimizing the Metis MapReduce library [53] and OpenMP [7]. MCTOP-PLACE includes 12 high-level performance policies that optimize for locality, bandwidth, or energy efficiency. We plug MCTOP-PLACE in Metis and achieve 17% better average performance, while consuming 14% less energy in four of the workloads. Similarly, we extend OpenMP with runtime support for configuring placement policies, enabling portable, high-level, and dynamic thread placement. We evaluate Green-Marl's [42] OpenMP-based graph workloads and improve the performance of various graph analytics, such as PageRank, by 22% on average.

To summarize, our main contributions are as follows:

1. MCTOP, a rich multi-core topology abstraction that enables policy-based portable software optimizations;
2. MCTOP-ALG, a portable algorithm for inferring the topology of multi-cores without relying on the topology information of the OS or the hardware;
3. `libmctop` and the software we build using `libmctop`, both available at:

   `http://lpd.epfl.ch/site/mctop`

Of course, `libmctop` has certain limitations. We have ported `libmctop` on `x86` and `SPARC` architectures, and cannot yet guarantee the effectiveness of MCTOP-ALG on other architectures (e.g., `ARM`, `POWER`). Additionally, in order to collect accurate measurements, `libmctop` requires a solo execution on the target processor for the one run that infers the topology (this means stopping all other applications for the duration of `libmctop`'s first execution).

The rest of the paper is organized as follows. In Section 2, we describe the programming interface of MCTOP. In Section 3, we introduce a generic algorithm for harvesting MCTOP topologies of multi-cores, while in Section 4, we present how to extend MCTOP topologies. We then describe examples of high-level performance optimization policies in Section 5 and use these ideas in designing a thread placement library in Section 6. Finally, we present practical examples, our related work, and conclude the paper in Sections 7, 8, and 9, respectively.

[1] On the multi-core of Figure 1, the OS has an incorrect mapping of cores to memory nodes, while MCTOP-ALG infers the correct mapping.

2. The MCTOP Topology Abstraction

The first step for achieving portable optimizations is to provide a programming abstraction of multi-core topologies. Software can build on this abstraction and avoid using the limited, non-extensible, specific view of multi-cores that is exposed by operating systems. To this end, we design MCTOP (*multi-core topology*), a portable topology abstraction. We opt for MCTOP, and for a user-level implementation of `libmctop`, instead of changing the existing OS interfaces and enriching them with extra information. This way, MCTOP and `libmctop` are (i) portable across OSs, and (ii) readily available to software developers, without the need to install new OS kernels. Additionally, MCTOP has two important characteristics. First, MCTOP is *generic*: It can be used to describe many modern multi-cores. Second, MCTOP is *extensible*: It can be extended to support any low-level details of multi-cores, which are necessary to achieve fine-tuning of software and portability at the same time.

In the remainder of this section, we first highlight certain characteristics of modern multi-cores that affect the design of MCTOP, we then describe the programming interface of MCTOP, and, finally, we illustrate several examples of MCTOP topologies.

State of Affairs of Modern Multi-Cores. Multi-core servers are typically *multi-socket* processors with *non-uniform memory accesses* (NUMA—i.e., the latency to access data depends on the placement of both the thread and the data). Non-uniformity is mainly a result of the multiple sockets of the processor. Traditionally, every socket is directly connected to one *local* memory node.[2]

Furthermore, modern multi-cores include several CPU cores in order to offer thread-level parallelism. Many processors also employ *simultaneous multi-threading* (SMT) for even higher thread-level parallelism. In short, with SMT, every core contains more than one *hardware context* (i.e., the scheduling granularity for software threads is hardware contexts, not cores). Each hardware context shares most of the core's resources with the other hardware context(s) (e.g., the caches and the pipeline). Both Linux and Solaris expose hardware contexts as individual cores to the user.

`libmctop` Programming Interface. MCTOP topologies are stored in description files, which are created by `libmctop` once and are then used to load the topology. Once a topology is loaded, the developer can either use `libmctop`'s programming interface to access MCTOP, or visualize the topology as textual output or as a graph. `libmctop` represents MCTOP topologies as a set of structures that are linked together to describe the processor. The most important structures of MCTOP are shown in Table 1.

These structures are interconnected (i) vertically, in order to represent the actual topology, and (ii) horizontally, for simplifying the traversal of all objects at each level. For instance, a `hw_context` holds pointers to its parent `hwc_group`, its parent `socket`, as well as its successor (in terms of proximity) `hw_context`. Additionally, every structure holds pointers to additional low-level information, such as memory latencies and bandwidths.

[2] On very large NUMA machines, it is possible to have fewer memory nodes than sockets (e.g., two sockets can share one memory node).

`hw_context`	The lowest scheduling unit of the processor. If SMT exists, `hw_context` represents a hardware context, otherwise it represents an actual core.
`hwc_group`	A group of `hw_context`s or `hwc_group`s, such as a core that contains two hardware contexts, or a group of cores that share the L2 cache. There might be multiple levels of `hwc_group` within a socket.
`socket`	A `hwc_group` with additional information about the NUMA memory nodes and the interconnection with other sockets.
`node`	A memory node with information such as capacity.
`interconnect`	The interconnection between two `socket`s. Contains information such as the communication latencies.
`mctop`	The structure that represents a processor and links everything together. Contains info about the latency levels, SMT, the number of sockets and cores, etc.

Table 1: The main structures of MCTOP.

We opt for a representation that uses terms that match any modern processor and are extensible for future designs. As such, the interface of `libmctop` uses terms which are familiar to system designers, such as:

- `mctop_get_local_node(hw_ctx)` to get the local node of a `hw_context`;
- `mctop_socket_get_cores(socket)` to get the cores of a `socket`; and
- `mctop_get_latency(id0, id1)` to get the latency between any two components.

2.1 Examples of MCTOP Topologies

`libmctop` can generate a simplified visual representation of MCTOP (using Graphviz [1]) in order to make the topology more accessible to developers. We illustrate MCTOP using `libmctop` on various `x86` and `SPARC` processors. We present below the automatically generated graphs and provide details of each platform. In Section 7, we use these platforms in our experiments.

Reading MCTOP Graphs. MCTOP's visual representation includes two main graphs, depicting the intra- and the cross-socket topologies respectively—e.g., Figure 2. The intra-socket graph (Figure 2a) includes the communication latencies inside a socket—28 cycles between SMT contexts of the same core and 116 cycles between difference cores. It also includes the latency and bandwidth from this socket to all the available memory nodes. The local node (Node 4—shown as a gray box) has a latency of 369 cycles and a maximum throughput of 13.1 GB/s.

The cross-socket graph (Figure 2b) shows the communication latencies between hardware contexts on different sockets of the machine (e.g., two threads on sockets 0 and

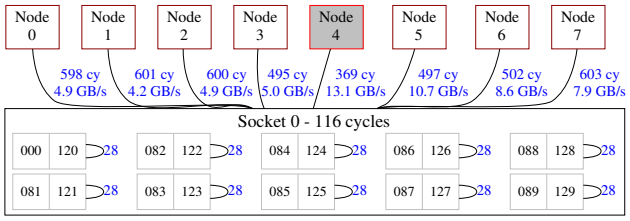
5 have a latency of 341 cycles), as well as the memory bandwidths when accessing the memory of another socket (cross-socket bandwidth is limited by the interconnect). Finally, the two-hops latency between hardware contexts that belong to sockets that are not directly connected is depicted as "*lvl 4*."

Intel Xeon Ivy Bridge (Ivy). The 20-core Intel Xeon (used as an example for explaining MCTOP-ALG—Figure 6) consists of two E5-2680 v2 10-core sockets (40 hardware contexts). Ivy runs at 1.2-2.8 GHz and includes 32 KB, 256 KB, and 25 MB (per die) L1, L2, and LLC, respectively.

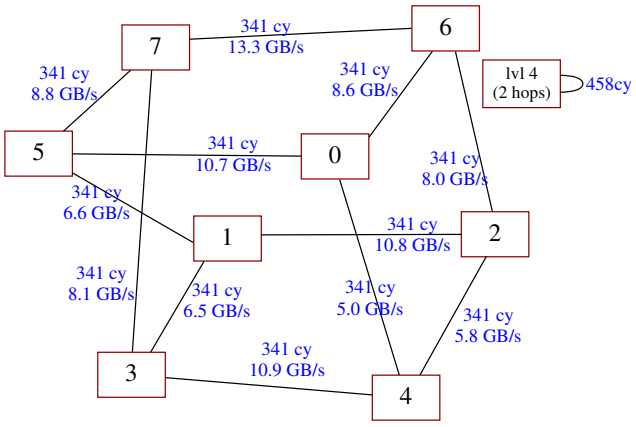
Intel Xeon Westmere (Westmere). The 80-core Intel Xeon (Figure 2) consists of eight E7-8867L 10-core sockets (160 hardware contexts). Westmere operates at 1.1-2.1 GHz and has 32 KB, 256 KB, and 30 MB (per die) L1, L2, and LLC data caches, respectively.

Intel Xeon Haswell (Haswell). The 48-core Intel Xeon (graph not shown due to space limitations) consists of four E7-4830 v3 12-core sockets (96 hardware contexts). Haswell operates at 1.2-2.7 GHz and has 32 KB, 256 KB, and 30 MB (per die) L1, L2, and LLC data caches, respectively.

AMD Opteron (Opteron). The 48-core AMD Opteron (Figure 1) contains four AMD Opteron 6172 multi-chip modules (MCMs) [27]. Each MCM has two 6-core dies, for a total of eight sockets. Opteron operates at 2.1 GHz and has 64 KB, 512 KB, and 5 MB (per die) L1, L2, and LLC data caches, respectively.



(a) Intra-socket topology of a socket.



(b) Cross-socket topology.

Figure 2: MCTOP of an 8-socket Intel processor.

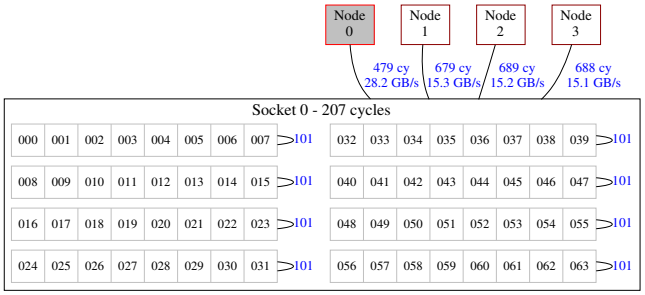


Figure 3: MCTOP of a socket of an Oracle processor.

Oracle SPARC T4-4 (SPARC). The Oracle SPARC T4-4 (Figure 3) consists of four T4 sockets with eight cores per socket and a total of 256 hardware contexts. SPARC operates at 3 GHz and has 16 KB, 256 KB, and 4 MB (per die) L1, L2, and LLC data caches, respectively.

3. MCTOP-ALG: Inferring Topologies

We introduce MCTOP-ALG, our algorithm that infers the basic topology of cache-coherent shared memory processors using only latency measurements. MCTOP-ALG relies on two simple, yet important observations regarding cache-coherence protocols of modern multi-core processors.

OBSERVATION 1
(Cache-coherence protocols are deterministic in the absence of contention)

Cache-coherence protocols are responsible for keeping data consistent in the various caches of multi-cores. Most modern processors implement the *MESI* coherence protocol [60], or variants of MESI.

Hardware cache-coherence protocols are deterministic by design. Still, non-deterministic schedules can appear, but only under contention (e.g., if multiple threads contend for a cache line, then the schedule of coherence messages is naturally not deterministic). In the absence of contention, hardware coherence protocols deliver deterministic schedules. In simple words, a given request type (e.g., requesting for writing), on a given multi-core, for a block of data in a specific MESI state and the same placement, always takes the same steps. Consider the simplified example of Figure 4, where a cache line cl is in the modified state[3] in the caches of core o and another core r is requesting the data for writing—a request known as *request for ownership* (RFO). The RFO request for cl misses in the private caches of r. The request finds that o holds the only copy of cl through the last-level cache (LLC) (or using a directory, depending on the specific implementation of MESI[4]). Once the copy is found, an invalidation request is sent to o's private caches to discard their

[3] Recall that the modified state means that this cache line is the only fresh copy of the data and this data is stale in memory.

[4] For example, modern Intel server processors use the LLC, while AMD servers use a directory (known as the probe filter [27]).

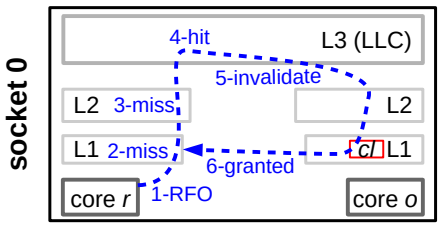


Figure 4: Coherence traffic for an RFO request.

	Thread x	Thread y
	thread_barrier()	
		CAS(shared_line)
	thread_barrier()	
	s = rdtsc()	
	CAS(shared_line)	
lat[x][y] =	rdtsc() - s - rdtsc_latency	

Figure 5: Lock-step execution of MCTOP-ALG's threads.

copy of cl, after which the RFO request is granted to r. If r is not in the same socket as o, the RFO request is propagated to the corresponding socket.

Overall, the coherence request takes deterministic steps. Hence, we can devise thread schedules that accurately measure the communication latency between any two contexts.

OBSERVATION 2
(Communication latencies characterize the topology) Multi-cores include several cache levels for minimizing latency to data. The latency of a request defines at large the *distance* between the source of the request and the placement of data. For instance, on the 2-socket Intel Xeon Ivy Bridge (see Section 2.1), 4, 12, and 42 cycles are (approximately) the latencies to access the three levels of caches, while 112 and 308 cycles represent the latencies to access data that are in the private caches of another core within the same socket and across sockets, respectively.

Two threads can potentially detect their relative placement based on their communication latency. For example, on Ivy, if two threads communicate in approximately 4 cycles, they *have to* reside on the same core as the L1 cache delivers this latency. In contrast, communication latency of 300+ cycles reveals that the two threads are on different sockets.

MCTOP-ALG *Algorithm.* MCTOP-ALG takes advantage of the aforementioned observations by collecting accurate hardware-context-to-hardware-context communication latency measurements and using them in inferring the topology of the machine. The implementation of MCTOP-ALG in `libmctop` requires only three functionalities from the underlying OS: A way to read the number of available hardware contexts and the number of memory nodes, and a way to pin threads to specific contexts.

MCTOP-ALG takes the following four steps:

1. Collects context-to-context latency measurements → latency table;
2. Clusters close values into groups and normalizes the latencies accordingly → normalized latency table;
3. For each latency value l, categorizes hardware contexts into groups of contexts that communicate with latency l with each other and with the same latency with other groups → per latency level components;
4. Creates the multi-core representation by assigning roles to components → topology;

We detail below these four steps using Ivy as an example—Figure 6. We then discuss several practical considerations regarding MCTOP-ALG and how the correctness of the inferred MCTOPs can be validated.

3.1 Context-to-Context Latencies

MCTOP-ALG uses two threads that move from hardware context to hardware context and fill up an $N \times N$ latency table, where N is the number of hardware contexts of the processor. For each data point, the two threads execute in lock step as shown in Figure 5 (similar measurements have been used in existing systems research [18, 30, 40, 73]). Thread y brings the data in a modified state in its local caches and then thread x measures the latency of its own access to the shared data using the timestamp counter of the core [4]. Reading the timestamp counter has a non-negligible latency which must be deducted from the latency measurements. Accordingly, the execution in Figure 5 subtracts the estimated cost of reading `rdtsc` (see Section 3.5 for more details).

The use of an atomic operation, such as compare-and-swap (CAS), is crucial for two reasons. First, CAS includes a memory fence, hence it precludes the effects of memory consistency models [67]. Second, CAS brings the data in the modified MESI state. The modified state is necessary for avoiding potential whole-machine communication when broadcasting invalidations for a shared cache line (e.g., in AMD Magny Cours [27, 30]). As we describe in Section 3.5, `libmctop` performs the measurements several times to produce stable results.

The outcome of this step is a latency table (Figure 6 ①). Note that in practice we only need to take measurements for either the upper or the lower triangular of the table because the topology is symmetric.

3.2 Latency Normalization

As the heatmap of Figure 6 ① shows, the relations between hardware contexts are rather clear. The white diagonal represents the individual contexts, the two light gray diagonals represent the hardware contexts of the same core, and the gray and dark-gray rectangles are the intra- and cross-socket latencies, respectively. To extract these relations, MCTOP-ALG calculates the cumulative distribution function (CDF) of the latency values—Figure 6 ②a. The value clusters of CDF represent these aforementioned relations. MCTOP-ALG

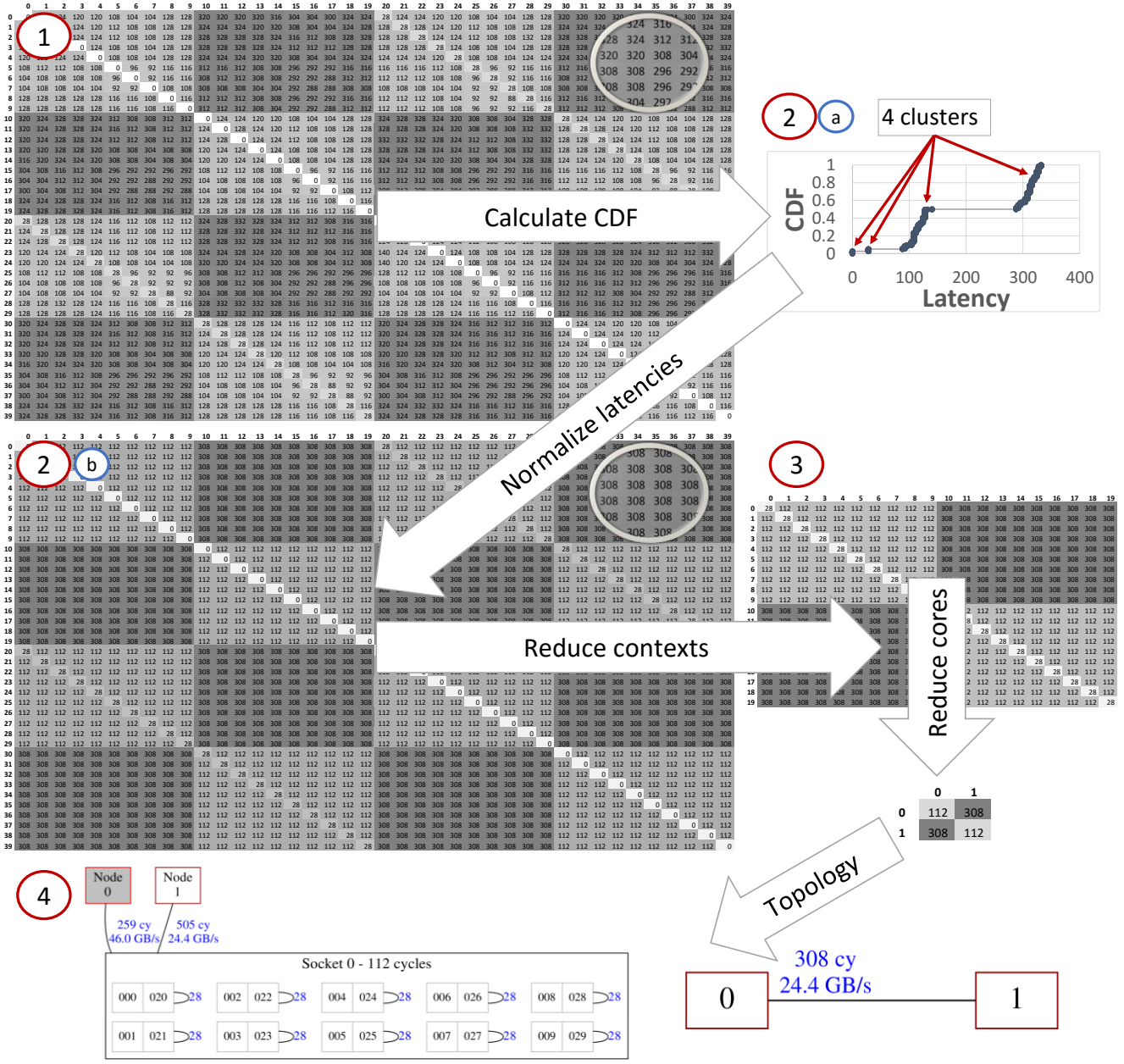


Figure 6: The four steps of MCTOP-ALG: From latency measurements to the automatic creation of MCTOP multi-core topology.

detects these latency clusters and for each cluster generates a triplet with the minimum, median, and maximum latencies.

MCTOP-ALG uses the latency clusters for normalizing the latency table (Figure 6 (2b)). Each value of the table is replaced with the median value of the corresponding cluster.

3.3 Component Creation

MCTOP-ALG uses the normalized latency table to extract the relations among hardware contexts for each latency level within the socket and assigns them to components. We recursively define a *component* C_l of level $l > 0$ as a set of components of level $l-1$ such that any two components in C_l communicate with the latency of level l and have the exact same normalized communication latencies with all the other components of level $l-1$. At level 0, with latency 0, every hardware context belongs to its own C_0 component.

Using this definition of components, MCTOP-ALG recursively groups hardware contexts together by performing classification and reduction of the latency table. For example, in Figure 6 (3), the first step is to group the contexts

(components C_0) of each core with each other and reduce the table by only keeping the components C_1 (i.e., the cores).[5] Then, the cores of each socket are reduced to C_2 components and we end up with only the cross-socket latencies table.

The outcome is a set of components for each latency level. This assignment of hardware contexts to components describes the relations between contexts.

3.4 Topology Creation

In this last step, MCTOP-ALG assigns "roles" to the components of different levels according to the MCTOP representation. The result is an abstraction of the actual topology of the processor as shown in Figure 6 ④ (the memory measurements are described in Section 4). MCTOP-ALG first detects whether the target multi-core includes SMT. If the multi-core has SMT, the components of the first non-zero latency group represent the physical cores of the processor. Similarly, MCTOP-ALG classifies as socket level the level with as many hardware contexts as $\frac{total\ \#contexts}{\#nodes}$. Every relation higher than sockets represents cross-socket connectivity.

3.5 Practical Considerations

Using the Timestamp Counter. Using the timestamp counter for the fine-grained measurements required by MCTOP-ALG produces various complications [59]. As we mention earlier in this section, reading the timestamp counter has a non-negligible latency that has to be accounted for in the measurements. To achieve this, `libmctop` explicitly estimates the cost of reading the counter (`rtdsc_latency` in Figure 5) and subtracts this cost from every measurement. Of course, even the estimation of `rtdsc_latency` is susceptible to various sources of variability, such as DVFS and interrupts. One potential approach to reducing these effects is to execute in kernel space (so that threads exclusively use the target core), to disable interrupts, and to modify the BIOS settings of the machine to remove "problematic" technologies such as DVFS [59].

However, we want `libmctop` to be readily available to developers in user space; thus, we do not employ these approaches. Instead, `libmctop` ensures the stability of measurements in user space, by taking into account technologies such as DVFS and by discarding spurious measurements—as described below. That being said, an implementation of MCTOP-ALG in kernel space would yield even higher accuracy of measurements than the one in `libmctop`.

Reducing the Effects of DVFS. Dynamic Voltage and Frequency Scaling (DVFS) is a common hardware technique for reducing power consumption, where underutilized cores can execute at various voltage/frequency settings. `libmctop` explicitly waits for the frequency of both cores to reach its maximum before proceeding to the lock-step execution. To detect DVFS, a thread executes a spin loop on a core and measures the time of this execution. If a subsequent execution of the same loop is faster, the core must be transitioning between DVFS states. Once a core reaches the maximum voltage/frequency setting, the core remains in this state as `libmctop` keeps the core fully occupied (e.g., even the thread barriers of `libmctop` are spin-based).

Stability of Measurements. MCTOP-ALG is designed to work solo on the target processor, as it relies on the accuracy of latency measurements. In order to improve this accuracy, each latency measurement is repeated n times ($n = 2000$ by default) and the median and standard deviation (stdev) are calculated. If stdev is higher than a threshold (7% of the median by default), the execution is repeated for that configuration and the stdev threshold is increased (up to 14% by default). The aforementioned default values are empirically selected and do not significantly affect the behavior of `libmctop`. Of course, the user of `libmctop` has to configure the tool in a reasonable manner. For example, if stdev is allowed to be 100% of the median value, `libmctop` might not get accurate measurements.

Still, `libmctop` might collect a few spurious measurements, mainly due to the effects of DVFS and of background OS processes executing on the same core of SMT-enabled processors. In these cases, if MCTOP-ALG is not able to infer the topology, an error message is printed and the user must retry the execution, possibly with different settings (see Section 3.6 for more details).

Detecting SMT. `libmctop` detects if the processor has symmetric multi-threading (SMT) using the same idea with "removing the effects of DVFS." A thread first executes a spin loop solo on a core and measures the time of this execution. Then, two threads execute the same loop on two contexts with minimum latency. If these are the hardware contexts of the same core, then due to SMT sharing, the duration of the spin loop will increase.

Performance. MCTOP-ALG in `libmctop` takes ∼3 seconds to infer the topology of our smallest platform (Ivy), and it takes 96 seconds to infer the topology of Westmere (160 contexts with DVFS). MCTOP-ALG is more stable and faster when DVFS is disabled. MCTOP-ALG is single-threaded, because (i) using more threads increases variability, and (ii) when threads of parallel pairs execute on contexts of the same core, measurements are severely impacted.

Dynamic Changes of Multi-Cores. `libmctop` does not currently support the detection of dynamic changes of the topology of a multi-core. If, after the execution of MCTOP-ALG, SMT is disabled through BIOS, or a hardware context is disabled via the OS, MCTOP-ALG must be re-executed in order to detect the new configuration.

[5] Note that the 28 cycles latency that we get for hardware contexts is higher that the L1 latency, because both threads execute on the same core while taking the measurements, thus increasing the latency.

3.6 MCTOP-ALG Output Validation

Validating that MCTOP-ALG inferred the correct topology for a multi-core is important. Currently, `libmctop` includes two methods for pointing out potential misbehaviors. Additionally, developers can consult processor manuals, or use manufacturer tools to validate MCTOP. In our experience, `libmctop` is able to infer the correct topology of multi-cores, except from the uncommon cases where `libmctop` is unable to perform clustering of values, as described below. Additionally, the measurements of `libmctop` match the expected (i.e., processor datasheet) values.

Unsuccessful Clustering of Latency Values. As we mention earlier, spurious measurements cannot be completely avoided in MCTOP-ALG. Our current implementation of MCTOP-ALG in `libmctop` relies on the symmetry and the hierarchical structure of modern multi-cores to detect such values. Specifically, `libmctop` expects that at every latency level, each component C_{l_i} of level $l > 0$, contains the same number of C_{l-1} components as any other C_{l_j} component. Additionally, every C_{l-1}, with $l > 0$, component exists in precisely one C_l component. Accordingly, if a spurious latency measurement is clustered in an incorrect group, `libmctop` can detect the problem and report an error.

Comparing MCTOP to the OS Topology. One basic sanity check is to compare the inferred MCTOP to the topology of the OS. If the two topologies match, we can be certain that the MCTOP topology is correct. Otherwise, if the two topologies differ, `libmctop` suggests which experiments to rerun, in order to understand whether MCTOP or the OS has the correct view of the hardware.

Comparing MCTOP to Other Tools and Manuals. Finally, if a developer is still in doubt regarding a MCTOP topology, she can refer to the official processor manuals for her multi-core. Some manufacturers also offer tools for measuring memory characteristics of their processors (e.g., [3, 72]).

4. Enriching MCTOP Topologies

The basic topology representation created with MCTOP-ALG includes the communication latencies of the processor by design. These latencies are sufficient for defining locality-oriented performance policies, such as "find the socket that is the closest to socket x." Although locality optimizations are very important on NUMA multi-cores, we argue that with access to further low-level information in the MCTOP abstraction, we can implement a broader set of performance policies (see Section 5 for examples). Therefore, MCTOP includes a set of additional multi-core measurements. We have implemented four essential plugins that measure memory latencies and bandwidths, cache information, and power-related measurements (only available on modern Intel processors). Essentially, `libmctop` gives the best-case bandwidth and latency of a multi-core—i.e., these characteristics in the absence of contention. Of course, developers can write their own plugins to further enrich MCTOP.

Memory Latency and Bandwidth. The two memory plugins use standard microbenchmark techniques (inspired by the ones used in Corey [18, 73]) to estimate memory latencies and bandwidths. In brief, the memory latency plugin creates a randomly connected linked list of cache lines out of a large allocated memory area. Due to the size and the randomness of the list, traversing it results in cache misses (memory accesses) for almost every iteration. The memory bandwidth plugin also allocates a large chunk of memory. performing sequential accesses instead. This way, it maximizes the memory accessed by each thread.

Cache Latency and Size. The cache plugin estimates both the size and the latency of the various levels in the cache hierarchy. To estimate latency, the plugin uses the same technique as the memory latency measurements. The cache size estimation is based on those latency measurements (i.e., it estimates the size of each level by detecting the data size that causes latency to increase). Additionally, the plugin loads and includes the cache sizes from the operating system.

Power Consumption. The latest Intel processors include Intel's running average power limit (RAPL) [4] interface for accurately measuring the power consumption of the cores, the package, and the DRAM. We design a `libmctop` plugin that uses RAPL to gather power measurements which indicate the breakdown of power to hardware contexts. In order to estimate the maximum power consumption, we use a memory intensive workload (the same that we use for bandwidth measurements). We measure and include in MCTOP measurements such as: Idle processor power, full power (all hardware contexts are active), power of the first hardware context, and power of the second context of one core.

5. Portable Optimizations with MCTOP

Optimizing a concurrent system for the underlying hardware hinders the portability to other processors. In certain cases, this lack of portability is inevitable. For example, using Intel's restricted transactional memory [4] results in software that can only execute on specific processor models.

However, for more traditional topology-oriented optimizations, such as for locality or bandwidth, we can achieve *portable optimizations* on top of MCTOP. We can do so because these traditional notions are accurately defined on MCTOP. For instance, "use n cores that are the closest to core x," is a policy that can be easily, accurately, and portably defined in MCTOP. Consequently, we can define high-level performance policies that leverage, but at the same time abstract the low-level details of multi-core topologies.

Essentially, MCTOP provides a topology query engine for multi-cores (similar to the system knowledge base of Barrelfish [65]). Of course, software should not rely on any assumptions regarding the underlying multi-core, but rather

build on top of `libmctop`'s programming interface to access information in a portable manner. For instance, an algorithm that explicitly allocates memory on nodes 0 and 1 will not work on a single-node processor. Instead, the developer can use `mctop_get_num_nodes` to provision for the available resources of any multi-core.

In what follows, we highlight several examples of portable optimizations with policies on top of MCTOP. Section 6 contains a detailed description of thread placement policies. In Section 7, we evaluate several of these examples.

Topology-Aware Work Stealing. Work stealing [16] is a commonly used technique in parallel runtimes that aims to minimize the imbalance of work across worker threads. In brief, worker threads have access to work queues from which they dequeue and execute chunks of work. In order to avoid imbalance, workers with no work must steal work from other worker threads. Ideally, work stealing must be performed in a way that (i) reduces the overhead of accessing the non-local queue, and (ii) optimizes the locality/bandwidth of the stealer to the work chunk.

↦ **MCTOP Policies.** On top of MCTOP, we can easily implement the following work-stealing policy: If the local work queue is empty, steal from the queue of worker threads that are the closest in terms of latency. If unsuccessful, continue with the contexts that are the next closest. Continue this process until work is found, or there is no work to steal. This policy defines work stealing with optimized locality.

Topology-Aware Reduction Trees. Many parallel algorithms and frameworks rely on the fork-join model. The most notable example is the MapReduce paradigm [31]. In the fork-join model, computation is split in chunks that are processed in parallel by multiple processes. The local results of each process are then reduced (joined) to get the final result. Intuitively, on a NUMA processor, when these local results represent a sizable amount of data, the thread and data placement of the reduction process can have a large effect on the performance of the computation.

↦ **MCTOP Policies.** We describe policies for cross-socket reduction trees that we believe are broadly applicable. The following steps assume that multiple threads can concurrently reduce the same data. Within sockets, all threads of a socket cooperate on reducing the same chunks. Across sockets, we build a binary reduction tree such that (i) the final destination socket/node is the one that requires the final data, and (ii) at each level of the tree, we choose the sockets to cooperate so that we maximize the bandwidth to data. We use these policies in a sorting algorithm in Section 7.

Power Consumption Estimation. Power consumption and energy efficiency have gained increased attention in the past few years [17, 21]. MCTOP's power-related measurements on modern Intel processors can be used to estimate the power consumption of a specific execution (i.e., a fixed thread placement) before the actual execution.

↦ **MCTOP Policies.** Being able to estimate the power consumption of an execution gives us the opportunity to trade performance for lower power. For example, a low-power policy (e.g., Section 6) prioritizes the use of hardware contexts that minimize power consumption.

Educated Backoffs. Waiting for a short period of time before retrying an operation, namely backing off, is an essential technique for alleviating congestion in software. For example, backoffs are used in lock implementations [10]. Estimating the correct amount of time to back off is difficult. Small backoffs miss a window for further optimization, while large backoffs could hurt performance by inducing idle periods of no work.

↦ **MCTOP Policies.** We define the granularity of backing off on top of MCTOP based on the intuition that "messages" on multi-cores travel as fast as coherence protocols. Accordingly, we set the backoff quantum to be the maximum latency between any two threads that are involved in the execution. We use this policy in lock algorithms in Section 7.

6. Thread Placement with MCTOP

A prominent way of using MCTOP is developing higher-level libraries that rely on performance policies. A natural construction on top of MCTOP is a library that abstracts the placement of threads to hardware contexts given some placement policy. For instance, we might need to place threads close to a specific node where some data resides.

We develop MCTOP-PLACE, a portable thread placement library. MCTOP-PLACE comprises two main components:

Name	Short description
`NONE`	Threads are not pinned to hardware contexts.
`SEQUENTIAL`	Use the sequential OS numbering.
`CON_HWC`	Starting from the socket with the maximum local memory bandwidth, place threads as compactly as possible on all hardware contexts of this socket and then continue to the next best connected neighboring socket.
`CON_CORE_HWC`	Same goal as `CON_HWC`. Instead of using all hardware contexts, use all unique cores of the socket before using the second hardware context. Still, fill up the first socket before using the next one.
`CON_CORE`	Same goal as `CON_HWC`. Instead of using all hardware contexts, use all unique cores of all used sockets. Once all cores are used, use the second+ context of each core.
`BALANCE`	Balanced `CON_*` placements. Instead of filling up a socket before using the next, balance threads to sockets.
`RR`	Place threads round robin to sockets. Prioritizes the sockets with maximum bandwidth to their local memory. Uses unique cores first (`RR_CORE`) or all hardware contexts of the core (`RR_HWC`).
`POWER`	Place threads so that the estimated maximum power consumption is minimized. (Intel processors only.)
`RR_SCALE`	Same as `RR_CORE`, but also re-adjusts the number of threads per socket to use as many as necessary to saturate the memory bandwidth to their local node.

Table 2: The set of policies offered by MCTOP-PLACE.

```
## MCTOP Placement   : MCTOP_PLACE_CON_HWC
# # Cores            : 15
# HW contexts (30 ) : 0 20 1 21 2 22 3 ...
# Sockets (2  )     : 20000 20001
# # HW ctx / socket : 20    10
# # Cores / socket  : 10    5
# BW proportions    : 0.655 0.345
# Max pow no DRAM   : 66.7  43.4   = 110.1 Watt
# Max pow with DRAM : 111.9 88.7   = 200.6 Watt
# Max latency       : 308   cycles
# Min bandwidth     : 24.28 GB/s
```

Figure 7: Example output of MCTOP-PLACE.

(i) individual thread placements, and (ii) a pool of placements that supports runtime modification of configurations.

MCTOP-PLACE. `libmctop` thread placement (MCTOP-PLACE) creates a mapping of threads to hardware contexts given a placement policy. Optionally, the user can provide the number of threads and the number of sockets to be used. The basic interface of MCTOP-PLACE includes functions for: (i) initializing a new MCTOP-PLACE object with a given policy, (ii) pinning a thread to the next available context of a MCTOP-PLACE object (if any), and (iii) unpinning a thread from the context and returning it to MCTOP-PLACE.

We implement 12 placement policies (Table 2). In non-SMT multi-cores, `CON_HWC`, `CON_CORE_HWC`, and `CON_CORE` policies are equivalent. We believe that the policies of MCTOP-PLACE cover the most prominent placement choices a developer can make, such as compacting or spreading threads as much as possible. Still, if none of these policies covers a required thread placement, implementing a new policy is straightforward since the basic data structures for doing so are already in place.

Apart from the mapping of threads to hardware contexts, MCTOP-PLACE provides a plethora of additional information and function calls to leverage MCTOP. Figure 7 shows an example output of `mctop_place_print` on Ivy (see Section 2.1). For a given allocation policy, MCTOP-PLACE calculates and exports details such as the number of cores used, the bandwidth proportions of each socket, and an estimation of the maximum power consumption with and without DRAM (assuming that the application will execute solo on the processor). Additionally, once a thread has been pinned, it has access to information such as its local node and its hardware context and core IDs within the socket. In Section 7 we use MCTOP-PLACE in various examples.

MCTOP-PLACE *Pool.* MCTOP-PLACE places threads according to a single policy. However, software systems might require different placement policies in different execution phases. To support this functionality, we build a MCTOP-PLACE pool object that offers runtime selection of placement policies. In Section 7 we show how we use MCTOP-PLACE's pool to extend the thread placement capabilities of OpenMP.

MCTOP-PLACE *vs. Thread Scheduling.* With MCTOP-PLACE, we enable developers to assign static thread placement policies to their workloads, optimizing them across platforms with no additional effort. Obviously, the developer still needs to select the right policy for optimizing a workload. Additionally, factors like contention and workload skews can affect the behavior of an application and change the optimal policy for a specific workload/platform combination. Furthermore, these static placements with MCTOP-PLACE might result in two similar applications utilizing the same hardware contexts of a processor, resulting in poor performance for both applications. These placement problems are present in existing multi-core libraries (e.g. `libnuma`, `hwloc`) and can be solved by OS-level thread scheduling—an orthogonal, very elaborate problem. We leave centralized scheduling with MCTOP in the OS for future work.

7. Examples of Portable Optimizations

We experimentally show how the performance policies on top of MCTOP achieve portable optimizations in software. Our goals for this section are to illustrate (i) the usefulness of the low-level measurements of MCTOP, (ii) the ability to optimize existing software using `libmctop`, and (iii) the portable efficiency of the resulting software.

Experimental Setup. We execute our experiments on all five platforms described in Section 2.1. We perform 11 runs of each experiment and present the median performance. We do not to show error bars for readability as our experiments have small variance. Whenever there is variability across runs, we discuss it in text. The duration of each of our lock experiments is 5 seconds.

7.1 Using Latencies to Optimize Locking

Traditional spinlock algorithms, such as ticket locks, resort to busy waiting when the lock is not free [11, 54]. While busy waiting, it can be beneficial to back off before re-accessing the shared memory location of the lock [10, 33]. As discussed in Section 5, with MCTOP it is straightforward to make educated backoff decisions for such algorithms. We use MCTOP to optimize three lock algorithms: test-and-set (TAS), test-and-test-and-set (TTAS) and ticket (TICKET) locks. We use as backoff quantum the maximum communication latency between any two threads involved in the execution. Different lock algorithms employ the backoff quantum in different ways. With ticket locks we set the backoff to be proportional to the position of the thread in the "queue" [30, 46, 54]. With TAS and TTAS, threads simply back off for one quantum before accessing the lock again.

Evaluation. Figure 8 includes the relative throughput of the three lock algorithms with and without our MCTOP-based backoff schemes. The experiment involves multiple threads competing for the same lock, performing 1000 cycles of work in the critical section, and then releasing the lock. Threads pause after each iteration to avoid long runs [56]. On both the `x86` and the `SPARC` processors, we use the `pause`

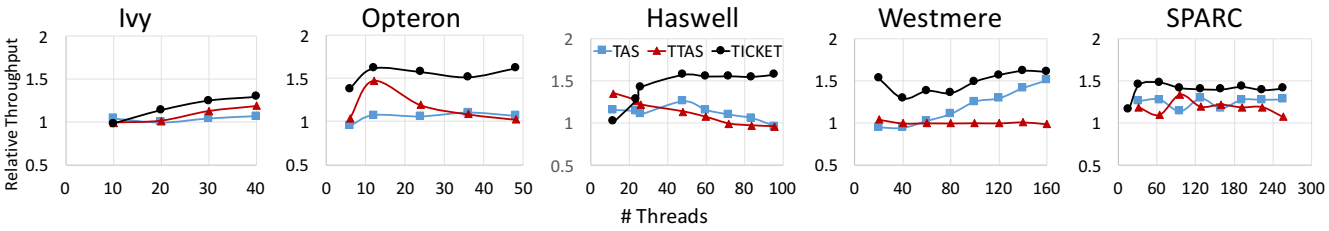


Figure 8: Throughput of different lock algorithms using educated backoffs from MCTOP.

instruction for pausing [4, 8] as the baseline. On `x86`, we invoke `pause` in a loop to implement our backoff quantum.

Backing off with the "correct" backoff granularity significantly improves performance: On average, we improve the performance of TAS, TTAS, and TICKET by 12%, 11%, and 39%, respectively. These performance gains are consistent across platforms, without requiring reconfiguration or recompilation of the applications. With TTAS, as contention increases, backing off does not make a difference, since most threads are still bashing the cache line with spinning.

Conclusion. MCTOP's low-level information can be used to optimize the performance of locking algorithms. The optimization is portable across platforms, as `libmctop`'s interface provides the necessary latencies on each platform.

7.2 Using `libmctop` in Parallel Mergesort

We use `libmctop` to devise a very fast, portable mergesort algorithm. Our novelty lies in the way we perform NUMA-aware merging. The starting point for our algorithm is the parallel sort algorithm of the C++ standard library (`gnu_parallel::sort`) [66]. `gnu_parallel::sort` involves two main steps: (i) it breaks the target array into n chunks and lets n threads sort these chunks with the standard sequential quicksort algorithm (n is the number of available threads), and (ii) it iteratively performs parallel merging on the sorted chunks until the result is a single sorted array. Our mergesort algorithm, namely `mctop_sort`, takes the same first step as `gnu_parallel::sort`. However, `mctop_sort` merges the sorted arrays using the topology-aware reduction tree policies presented in Section 5.

Using SMT Cleverly. Merging two sorted arrays using traditional comparison instructions is sub-optimal: The aggressive out-of-order cores are not able to predict the direction of the merge branch (i.e., which of the two arrays will give the next element). Recent projects [26, 43] show how to use SIMD instructions for efficient merging. Using 128-bit instructions, we can create a bitonic merge network that merges 8 elements at a time. We implement a variant of `mctop_sort`, namely `mctop_sort_sse`, that uses SIMD instructions. Once the sequential sorting is over, we let the first hardware context of each core use SIMD, while the remaining perform traditional merging. To compensate for the faster merging with SIMD, SIMD threads are allocated three-time more data than the non-SIMD threads.

Evaluation. Figure 9 presents a performance comparison between `mctop_sort`, `mctop_sort_sse` (only on platforms that support SIMD instructions) and `gnu_parallel::sort` (`gnu`). Both algorithms assume an unsorted array on socket 0 and produce the sorted array on the same socket. We present execution times for runs with 16 threads on every machine, as well as with the total number of hardware contexts available. With `mctop_sort`, threads are spread across sockets, in order to benefit from the large LLCs of each socket (using `RR` policy from MCTOP-PLACE), while the merging tree is designed

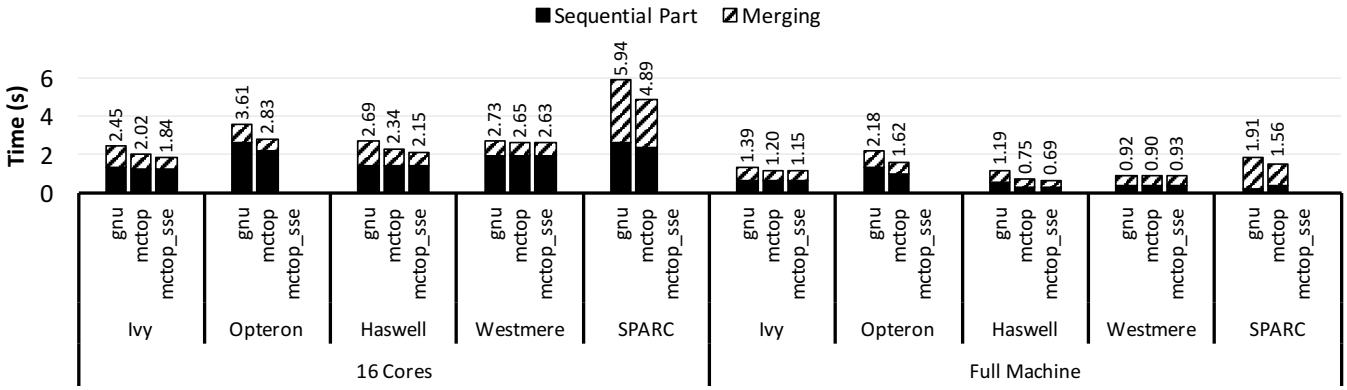


Figure 9: Breakdown of sorting time for 1 GB worth of integers on various platforms.

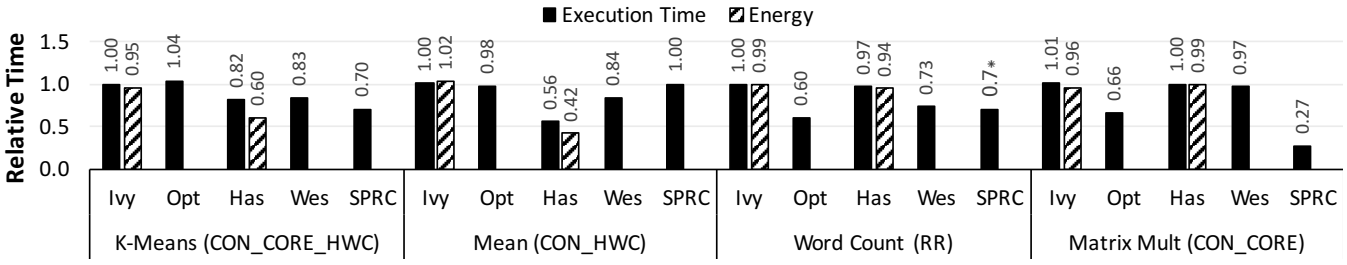


Figure 10: Relative execution time and energy consumption of Metis with `libmctop` compared to Metis without. Lower is better. All workloads are optimized for performance. (*On SPRC we use the `CON_CORE` placement for Word Count.)

to take advantage of the bandwidth of the machine. The performance of our algorithm is stable across runs, since the placement of threads and data is deterministic. In contrast, we notice big variance for `gnu_parallel::sort`, based on the thread placement of the OS.

`mctop_sort` is consistently faster than `gnu_parallel::sort` (we observe the same behavior for other data sizes), because it deterministically chooses a good placement of threads. On average, `mctop_sort` is 17% faster than `gnu_parallel::sort`, with merging being 25% faster (if we exclude the sequential part of the sorting that is the same on both algorithms). `mctop_sort_sse` delivers 18% higher performance than `gnu_parallel::sort` on average, and it can be up to 9% faster than `mctop_sort` (on Haswell). The performance benefits of `mctop_sort` are larger with 16 threads, as the OS scheduling for `gnu_parallel::sort` has more opportunities for bad thread placements.

Conclusion. Building topology-aware merging is straightforward on MCTOP. We leverage low-level details (e.g., bandwidth and latency for locality) of multi-cores without the need for any platform-specific optimizations.

7.3 Using `libmctop` to Improve Metis

We use `libmctop` to optimize the Metis MapReduce library for multi-cores [19, 53]. Metis pins worker threads to hardware contexts sequentially. We build a new version of Metis that uses the high-level placement policies of MCTOP-PLACE in `libmctop` at runtime (see Section 6).

Evaluation. We evaluate Metis in terms of performance and energy efficiency (where energy measurements are available). Due to space considerations, we present four of the applications shipped with the source code of Metis, for which placement has a significant effect on performance—Figure 10. We execute representative placement policies for each workload on *Ivy*, and choose the one that gives the best performance. We then use that policy for all the platforms. We also select the best-performance number of threads for both versions of Metis. Our MCTOP-enabled Metis always uses fewer or as many threads as the default Metis.

	Relative to performance oriented		
Workload	**Time**	**Energy**	**Energy Efficiency**
K-Means	1.186	0.774	1.089
Mean	1.045	0.915	1.046

Figure 11: Comparison of energy-oriented thread placement of Metis to performance-oriented placement (i.e., as in Figure 10) on Ivy. For energy efficiency, higher is better.

As Figure 10 reveals, different workloads have different placement needs. Thus, using the default sequential policy of Metis delivers sub-optimal performance in all workloads for different platforms. Our version of Metis delivers 17% better performance on average, across all platforms, with 14% less energy on the two Intel processors. Performance benefits are higher on bigger machines, as the communication latencies between two arbitrary cores vary significantly. It is worth noting that in one workload (Word Count), SPARC has different placement requirements than the `x86` platforms, delivering the best performance with cores of a single socket. Our performance analysis shows that Word Count has heavy memory allocation and synchronization that benefit from intra-socket locality. Finally, note that in this example we aim at performance, although we do achieve energy gains in some cases. In several Metis workloads, we can trade performance for energy (efficiency) (e.g., using the `POWER` policy)—shown in Figure 11. For instance, with K-Means on Ivy, we can trade 19% of performance to achieve 9% better energy efficiency by using fewer physical cores.

Conclusion. By modifying the Metis library, we show how a complex software system can easily take advantage of MCTOP, in order to achieve portable optimizations. General-purpose frameworks, such as Metis, can get out-of-the-box benefits from using MCTOP through `libmctop`.

7.4 Using `libmctop` to Enrich OpenMP

The GNU `libgomp` OpenMP runtime [5, 7] does not pin threads to cores by default. Still, `libgomp` allows users to set the available *places* of parallel threads on hardware contexts, as well as high-level strategies for assigning parallel threads to places. `libgomp` thread placement capabilities are: (i) offline—they are set through environmental variables

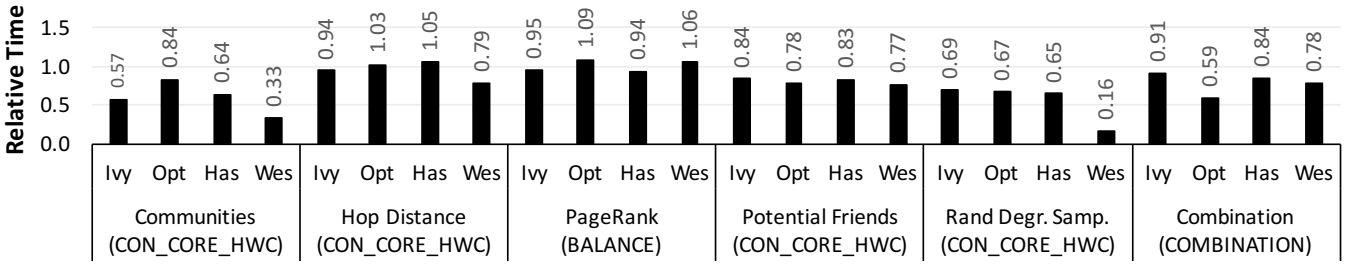


Figure 12: Relative execution time of MCTOP_MP compared to default OpenMP for various workloads.

before the execution, (ii) inflexible—placements cannot be modified at runtime and are dependent on the number of threads used during initialization, (iii) not fully portable—in many cases placements must be defined differently across platforms to achieve the same effects, (iv) not optimized—placements do not rely on latency or bandwidth numbers.

We extend the thread placement capabilities of `libgomp` (in gcc v4.9.3) using `libmctop` (MCTOP-PLACE) in order to offer richer and higher-level placement policies. In detail, we add the `omp_set_binding_policy` function to the OpenMP interface. Doing so, we enable developers to (i) choose placement policies during runtime, (ii) change placement policies between parallel regions, and (iii) leverage the high-level semantics of the MCTOP-PLACE placement policies that generate portable thread bindings.

Evaluation. We evaluate our extended OpenMP runtime (MCTOP_MP) against the vanilla `libgomp` OpenMP library on various graph algorithm workloads produced by Green-Marl [42] (due to space limitations, we only present workloads for which thread placement affects performance)—Figure 12.[6] We use large datasets (e.g., 100 million nodes with 800 million edges).

We use MCTOP_MP to enable a proof-of-concept automatic thread-placement policy-selection mechanism, by running small parts of the workload using different policies and identifying a good policy for each parallel section. In contrast, such online decisions are not possible with OpenMP, as it does not offer the same high-level semantics and also cannot adjust the thread placement at runtime. Even if the configuration was manually selected for OpenMP, the developer would still need to find which placement policy matches the characteristic of each algorithm and implement this policy across platforms. Still, there are a few cases where MCTOP_MP results in up to 9% lower performance than OpenMP due to the "pre-processing" stage. Overall, our MCTOP_MP version of the algorithms is on average 22% faster across platforms and workloads.

We further port MCTOP_MP's automatically-selected thread placements to the OpenMP runtime, in order to estimate the amount of work for reproducing these placements using the default OpenMP capabilities. We observe that in order to reproduce the exact same configurations (with possibly different number of threads per platform) we had to design one policy per-platform per-workload with OpenMP. In terms of performance—not shown in the graphs—our automatic MCTOP_MP solution delivers very similar performance to OpenMP with fixed placements.

Finally, we combine two kernels (PageRank and Potential Friends) into a single application, namely Combination. With OpenMP, it is impossible to recreate MCTOP_MP's placement: We have to choose the correct placement policy for either of the kernels, while the performance of the other suffers. This results in MCTOP_MP being 22% faster than OpenMP for this workload on average.

Conclusion. MCTOP_MP enables portable optimizations through `libmctop` in software libraries such as OpenMP. MCTOP_MP offers high-level placement policies and runtime support for policy selection and adaptation—characteristics that we believe are useful to OpenMP developers.

8. Related Work

Optimizing Software for Multi-Cores. As corroborated by a large amount of work in operating systems [12, 14, 18, 19, 35, 61, 70, 74], databases [36, 44, 62, 63, 75], programming languages [37, 58], parallel runtimes [7, 9, 41, 52], key-value stores [15, 51], and synchronization [22–24, 32, 45], system developers need to optimize software for the target platform to achieve good performance. We discuss below selected examples of multi-core optimizations.

Baumann et al. [12, 13] design the Barrelfish OS that views modern multi-cores as a network of processors and relies on message passing in order to avoid the intractable task of optimizing for every single platform. Our MCTOP-ALG algorithm essentially builds on top of this network view in order to automatically infer the topology of multi-cores. Moreover, MCTOP bears similarity to the system knowledge base [65] of Barrelfish that also exposes hardware information to software developers.

Giceva et al. [36] explore the efficient deployment of database query plans on multi-core hardware with the aim of improving database performance. Psaroudakis et al. [63] describe how data placement and access patterns can affect the performance of database workloads on modern NUMA

[6] The available implementation of Green-Marl [2] does not support `SPARC`.

machines. In the same vein, Gidra et al. [37, 38] improve the performance of the JVM garbage collector by mainly optimizing memory placement.

In our previous work [30], we showed that synchronization is mainly a property of the underlying hardware. Guiroux et al. [39] corroborate that different lock algorithms perform the best with different configurations. Similarly, various lock algorithms and techniques [23, 24, 32] are NUMA (i.e., topology) aware. Kaestle et al. [45] develop a library for generating efficient inter-core broadcast trees tuned to modern NUMA machines.

MCTOP is built based on the realization that multi-core optimizations are necessary for good system performance. With MCTOP, such optimizations can be made portable.

OS Scheduling. A significant amount of work has dealt with OS scheduling and memory/thread placement, with a focus on NUMA architectures [25, 29, 48–50, 55, 64, 68, 76]. We provide `libmctop` (MCTOP) in user-space, so that it is readily available for any application and exemplify thread placement with `libmctop` to illustrate portability. We acknowledge that thread scheduling is an orthogonal, very elaborate problem. We believe MCTOP to be a suitable substrate for designing schedulers.

Autotuning. There is a large number of systems and frameworks for offline and online autotuning [28, 34, 57, 71]. These typically focus on portability and application tuning to optimize for specific goals (e.g., performance, energy efficiency). Using MCTOP to feed configuration parameters to algorithms could be seen as a form of autotuning. However, typical autotuning solutions use experiments to select among a set of candidate implementations/configurations per platform. In contrast, MCTOP offers a query engine for hardware characteristics, deterministically defining notions such as locality (without parameter exploration).

Tools for Multi-Cores. Libraries with similar functionality to MCTOP already exist. The most prominent ones are `libnuma` [47], `liblgrp` [6], and `hwloc` [20]. Similarly to `libmctop`, all three provide some form of topology abstraction, as well as APIs for thread and memory placement. In contrast to `libmctop`, all three libraries rely on the OS for the topology of the machine (which as we have discussed can lead to inaccuracies). They also lack the low-level measurements that the enriched MCTOP abstraction offers.

Additionally, `libnuma` and `liblgrp` offer relative "distances" between resources. These depend on the OS and can be very inaccurate. Both `libnuma` and `liblgrp` are also OS-specific (`libnuma` works on Linux, while `liblgrp` on Solaris). `hwloc` is portable across platforms (i.e., it can load the topology from various operating systems), but is also missing the detailed latency and bandwidth measurements of MCTOP, which, as we have shown, are crucial for optimizing software. `hwloc` also offers an API that can be used across platforms. Unfortunately, it focuses mainly on locality and the available cache hierarchies of the platforms. In contrast, with MCTOP, we have both the portable abstraction of the topology, as well as the enriched measurements which can be used either directly or indirectly to optimize software across platforms.

`LIKWID` [69] is a set of command-line tools that visualize the thread and cache topology of a multi-core, as well as control the thread affinities of an application. `LIKWID` relies on the operating system for its topology (currently it supports only Linux) and focuses mainly on performance counter measurements.

`libmctop` uses latency and bandwidth measurements to augment MCTOP. Similar measurements have been presented in previous operating system and synchronization work [18, 30, 40, 73]. Intel's latency checker [3] and performance counter monitor [72] can be used to measure the memory latencies and bandwidths on Intel platforms.

As we show in this paper, `libmctop` and MCTOP contain all the necessary components to achieve portable optimizations on multi-cores.

9. Conclusions and Future Work

We introduced MCTOP, a topology abstraction that enables developers to optimize their software on multi-cores in a portable manner. MCTOP abstracts both the topology and important low-level performance information of the processor. MCTOP is automatically generated by our MCTOP-ALG algorithm and is exposed to developers through our `libmctop` library. We showed how developers can define high-level policies on top of MCTOP in order to achieve portable optimizations. We illustrated these high-level policies on various examples, including a topology-aware MapReduce library and an extended OpenMP runtime with dynamic support for thread placement based on `libmctop`.

Future Work. In future work, we intend to build thread scheduling on top of MCTOP inside the OS. Such a scheduler requires solving various interesting research questions. First, it asks for an approach of dynamically determining the optimal policy for an application, removing from the user the need to statically choose a thread policy for an application/-workload combination. Additionally, it requires the ability to schedule applications that co-execute on the same machine and (possibly) interfere in their execution. In order to perform scheduling of multiple applications, the scheduler needs to keep track of the effective topology characteristics. For example, if an application is already executing, the effective memory bandwidth for another application is less than the total bandwidth reported by MCTOP.

Acknowledgments

We wish to thank our shepherd, Jean-Pierre Lozi, and the anonymous reviewers for their fruitful comments on improving the paper. This work has been supported in part by the European Research Council (ERC) Grant 339539 (AOC).

References

[1] Graphviz - Graph Visualization Software. http://www.graphviz.org.

[2] Green-Marl. http://github.com/stanford-ppl/Green-Marl.

[3] Intel Memory Latency Checker. https://software.intel.com/en-us/articles/intelr-memory-latency-checker.

[4] Intel 64 and IA-32 Architectures Software Developer Manuals. http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.

[5] GNU libgomp. http://gcc.gnu.org/onlinedocs/libgomp/.

[6] Memory and Thread Placement Optimization Developer's Guide. http://docs.oracle.com/cd/E26502_01/html/E35301/toc.html.

[7] OpenMP Application Program Interface, Version 4.0. July 2013. http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf.

[8] SPARC T4 Supplement to the Oracle SPARC Architecture 2011. http://www.oracle.com/technetwork/server-storage/sun-sparc-enterprise/documentation/sparc-servers-documentation-163529.html.

[9] U. A. Acar, A. Charguéraud, and M. Rainey. Scheduling Parallel Programs by Work Stealing with Private Deques. PPoPP '13.

[10] A. Agarwal and M. Cherian. Adaptive Backoff Synchronization Techniques. ISCA '89.

[11] T. E. Anderson. The Performance of Spin Lock Alternatives for Shared-Money Multiprocessors. *IEEE IPDS '90*.

[12] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The Multikernel: A New OS Architecture for Scalable Multicore Systems. SOSP '09.

[13] A. Baumann, S. Peter, A. Schüpbach, A. Singhania, T. Roscoe, P. Barham, and R. Isaacs. Your Computer is Already a Distributed System. Why isn't Your OS? HotOS '09.

[14] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. OSDI '14.

[15] M. Berezecki, E. Frachtenberg, M. Paleczny, and K. Steele. Power and Performance Evaluation of Memcached on the TILEPro64 Architecture. *Sustainable Computing: Informatics and Systems '12*.

[16] R. D. Blumofe and C. E. Leiserson. Scheduling Multithreaded Computations by Work Stealing. *JACM '99*.

[17] S. Borkar. Design Challenges Of Technology Scaling. *IEEE Micro '99*.

[18] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: An Operating System for Many Cores. OSDI '08.

[19] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, N. Zeldovich, et al. An Analysis of Linux Scalability to Many Cores. OSDI '10.

[20] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst. hwloc: A generic framework for managing hardware affinities in HPC applications. PDP '10.

[21] D. J. Brown and C. Reams. Toward Energy-Efficient Computing. *CACM '10*.

[22] I. Calciu, D. Dice, Y. Lev, V. Luchangco, V. J. Marathe, and N. Shavit. NUMA-Aware Reader-Writer Locks. PPoPP '13.

[23] M. Chabbi and J. Mellor-Crummey. Contention-Conscious, Locality-Preserving Locks. PPoPP '16.

[24] M. Chabbi, M. Fagan, and J. Mellor-Crummey. High Performance Locks for Multi-Level NUMA Systems. PPoPP '15.

[25] V. Chegu and R. van Riel. Automatic NUMA Balancing. http://events.linuxfoundation.org/sites/events/files/slides/summit2014_riel_chegu_w_0340_automatic_numa_balancing_0.pdf.

[26] J. Chhugani, A. D. Nguyen, V. W. Lee, W. Macy, M. Hagog, Y.-K. Chen, A. Baransi, S. Kumar, and P. Dubey. Efficient Implementation of Sorting on Multi-Core SIMD CPU Architecture. *VLDB '08*.

[27] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes. Cache Hierarchy and Memory Subsystem of the AMD Opteron Processor. *IEEE Micro '10*.

[28] C. Ţăpuş, I.-H. Chung, and J. K. Hollingsworth. Active Harmony: Towards Automated Performance Tuning. SC '02.

[29] M. Dashti, A. Fedorova, J. R. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quéma, and M. Roth. Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems. ASPLOS '13.

[30] T. David, R. Guerraoui, and V. Trigonakis. Everything You Always Wanted to Know About Synchronization but Were Afraid to Ask. SOSP '13.

[31] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *CACM '08*.

[32] D. Dice, V. Marathe, and N. Shavit. Lock Cohorting: A General Technique for Designing NUMA Locks. PPoPP '12.

[33] B. Falsafi, R. Guerraoui, J. Picorel, and V. Trigonakis. Unlocking Energy. USENIX ATC '16.

[34] M. Frigo and S. G. Johnson. The Design and Implementation of FFTW3. *Proceedings of the IEEE*, 2005.

[35] B. Gamsa, O. Krieger, J. Appavoo, and M. Stumm. Tornado: Maximizing Locality and Concurrency in a Shared Memory Multiprocessor Operating System. OSDI '99.

[36] J. Giceva, G. Alonso, T. Roscoe, and T. Harris. Deployment of Query Plans on Multicores. *VLDB '14*.

[37] L. Gidra, G. Thomas, J. Sopena, and M. Shapiro. A Study of the Scalability of Stop-the-World Garbage Collectors on Multicores. ASPLOS '13.

[38] Gidra, Lokesh and Thomas, Gaël and Sopena, Julien and Shapiro, Marc and Nguyen, Nhan. Numagic: A Garbage

Collector for Big Data on Big NUMA Machines. In *ASPLOS '15*.

[39] H. Guiroux, R. Lachaize, and V. Quéma. Multicore Locks: The Case Is Not Closed Yet. USENIX ATC '16.

[40] D. Hackenberg, D. Molka, and W. E. Nagel. Comparing Cache Architectures and Coherency Protocols on x86-64 Multicore SMP Systems. ACM MICRO '09.

[41] T. Harris and S. Kaestle. Callisto-RTS: Fine-grain Parallel Loops. USENIX ATC '15.

[42] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun. Green-Marl: A DSL for Easy and Efficient Graph Analysis. ASPLOS '12.

[43] H. Inoue and K. Taura. SIMD- and Cache-Friendly Algorithm for Sorting an Array of Structures. *VLDB '15*.

[44] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi. Shore-MT: A Scalable Storage Manager for the Multicore Era. EDBT '09.

[45] S. Kaestle, R. Achermann, R. Haecki, M. Hoffmann, S. Ramos, and T. Roscoe. Machine-Aware Atomic Broadcast Trees for Multicores. OSDI '16.

[46] S. Kashyap, C. Min, and T. Kim. Scalability in the Clouds!: A Myth or Reality? APSys '15.

[47] A. Kleen. A NUMA API for Linux. *SUSE Labs white paper*, 2004.

[48] D. Koufaty, D. Reddy, and S. Hahn. Bias Scheduling in Heterogeneous Multi-Core Architectures. EuroSys '10.

[49] B. Lepers, V. Quéma, and A. Fedorova. Thread and Memory Placement on NUMA Systems: Asymmetry Matters. USENIX ATC '15.

[50] T. Li, D. Baumberger, D. A. Koufaty, and S. Hahn. Efficient Operating System Scheduling for Performance-Asymmetric Multi-Core Architectures. SC '07.

[51] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. NSDI '14.

[52] Z. Majo and T. R. Gross. A Library for Portable and Composable Data Locality Optimizations for NUMA Systems. PPoPP '15.

[53] Y. Mao, R. Morris, and M. F. Kaashoek. Optimizing MapReduce for Multicore Architectures. In *Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Tech. Rep*. Citeseer, 2010.

[54] J. Mellor-Crummey and M. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *TOCS '91*.

[55] A. Merkel, J. Stoess, and F. Bellosa. Resource-Conscious Scheduling for Energy Efficiency on Multicore Processors. EuroSys '10.

[56] M. Michael and M. Scott. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. PODC '96.

[57] S. Muralidharan, A. Roy, M. Hall, M. Garland, and P. Rai. Architecture-Adaptive Code Variant Tuning. ASPLOS '16.

[58] T. Ogasawara. NUMA-Aware Memory Manager with Dominant-Thread-Based Copying GC. OOPSLA '09.

[59] G. Paoloni. How to Benchmark Code Execution Times on Intel IA-32 and IA-64 Instruction Set Architectures. *Intel Corporation white paper*, 2010.

[60] M. S. Papamarcos and J. H. Patel. A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories. ISCA '84.

[61] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The Operating System is the Control Plane. OSDI '14.

[62] D. Porobic, I. Pandis, M. Branco, P. Tözün, and A. Ailamaki. OLTP on Hardware Islands. *VLDB '12*.

[63] I. Psaroudakis, T. Scheuer, N. May, A. Sellami, and A. Ailamaki. Scaling Up Concurrent Main-Memory Column-Store Scans: Towards Adaptive NUMA-Aware Data and Task Placement. *VLDB '15*.

[64] J. C. Saez, M. Prieto, A. Fedorova, and S. Blagodurov. A Comprehensive Scheduler for Asymmetric Multicore Systems. EuroSys '10.

[65] A. Schüpbach, S. Peter, A. Baumann, T. Roscoe, P. Barham, T. Harris, and R. Isaacs. Embracing Diversity in the Barrelfish Manycore Operating System. MMCS '08.

[66] J. Singler, P. Sanders, and F. Putze. MCSTL: The Multi-Core Standard Template Library. Euro-Par '07.

[67] D. J. Sorin, M. D. Hill, and D. A. Wood. A Primer on Memory Consistency and Cache Coherence. *Synthesis Lectures on Computer Architecture*, 2011.

[68] D. Tam, R. Azimi, and M. Stumm. Thread Clustering: Sharing-Aware Scheduling on SMP-CMP-SMT Multiprocessors. EuroSys '07.

[69] J. Treibig, G. Hager, and G. Wellein. LIKWID: A Lightweight Performance-Oriented Tool Suite for x86 Multicore Environments. PSTI '10.

[70] D. Wentzlaff and A. Agarwal. Factored Operating Systems (fos): The Case for a Scalable Operating System for Multicores. *SIGOPS '09*.

[71] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated Empirical Optimizations of Software and the ATLAS Project. *Parallel Computing '01*.

[72] T. Willhalm, R. Dementiev, and P. Fay. Intel Performance Counter Monitor-a Better Way to Measure CPU Utilization. `http://software.intel.com/en-us/articles/intel-performance-counter-monitor`.

[73] K. Yotov, K. Pingali, and P. Stodghill. Automatic Measurement of Memory Hierarchy Parameters. SIGMETRICS '05.

[74] G. Zellweger, S. Gerber, K. Kourtis, and T. Roscoe. Decoupling Cores, Kernels, and Operating Systems. OSDI '14.

[75] W. Zheng, S. Tu, E. Kohler, and B. Liskov. Fast Databases with Fast Durability and Recovery Through Multicore Parallelism. OSDI '14.

[76] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing Shared Resource Contention in Multicore Processors via Scheduling. ASPLOS '10.