Two-Level Main Memory Co-Design: Multi-Threaded Algorithmic Primitives, Analysis, and Simulation

Michael A. Bender*§ Jonathan Berry† Simon D. Hammond† K. Scott Hemmert† Samuel McCauley*
Branden Moore† Benjamin Moseley‡ Cynthia A. Phillips† David Resnick† and Arun Rodrigues†

*Stony Brook University, Stony Brook, NY 11794-4400 USA {bender, smccauley}@cs.stonybrook.edu
†Sandia National Laboratories, Albuquerque, NM 87185 USA
{jberry, sdhammo, kshemme, bjmoor, caphill, drresni, afrodi}@sandia.gov
‡Washington University in St. Louis, St. Louis, MO 63130 USA bmoseley@wustl.edu
§Tokutek, Inc. www.tokutek.com

***Abstract*—A fundamental challenge for supercomputer architecture is that processors cannot be fed data from DRAM as fast as CPUs can consume it. Therefore, many applications are memory-bandwidth bound. As the number of cores per chip increases, and traditional DDR DRAM speeds stagnate, the problem is only getting worse. A variety of non-DDR 3D memory technologies (Wide I/O 2, HBM) offer higher bandwidth and lower power by stacking DRAM chips on the processor or nearby on a silicon interposer. However, such a packaging scheme cannot contain sufficient memory capacity for a node. It seems likely that future systems will require at least two levels of main memory: high-bandwidth, low-power memory near the processor and low-bandwidth high-capacity memory further away. This near memory will probably not have significantly faster latency than the far memory. This, combined with the large size of the near memory (multiple GB) and power constraints, may make it difficult to treat it as a standard cache.**

In this paper, we explore some of the design space for a user-controlled multi-level main memory. We present algorithms designed for the heterogeneous bandwidth, using streaming to exploit data locality. We consider algorithms for the fundamental application of sorting. Our algorithms asymptotically reduce memory-block transfers under certain architectural parameter settings. We use and extend Sandia National Laboratories' SST simulation capability to demonstrate the relationship between increased bandwidth and improved algorithmic performance. Memory access counts from simulations corroborate predicted performance. This co-design effort suggests implementing two-level main memory systems may improve memory performance in fundamental applications.

I. Introduction

Recently vendors have proposed a new approach to improve memory performance by increasing the bandwidth between cache and memory [18], [21]. The approach is to bond memory directly to the processor chip or to place it nearby on a silicon interposer. By placing memory close to the processor, there can be a higher number of connections between the memory and caches, enabling higher bandwidth than current technologies. While the term *scratchpad*[1] is overloaded within the computer architecture field, we use it throughout this paper to describe a high-bandwidth, local memory that can be used as a temporary storage location.

The scratchpad cannot replace DRAM entirely. Due to the physical constraints of adding the memory directly to the chip, the scratchpad cannot be as large as DRAM, although it will be much larger than cache, having gigabytes of storage capacity. Since the scratchpad is smaller than main memory, it does not fully replace main memory, but instead augments the existing memory hierarchy.

The scratchpad has other limitations besides its size. First, the scratchpad does not significantly improve upon the latency of DRAM. Therefore, the scratchpad is not designed to accelerate memory-latency-bound applications, but rather bandwidth-bound ones. Second, adding a scratchpad does not improve the bandwidth between DRAM and cache. Thus, the scratchpad will not accelerate a computation that consists of a single scan of a large chunk of data that resides in DRAM.

This gives rise to a new multi-level memory hierarchy where two of the components—the DRAM and the scratchpad—work in parallel. We view the DRAM and scratchpad to be on the same *level* of the hierarchy because their access times are similar. There is a tradeoff between the memories: the scratchpad has limited space and the DRAM has limited bandwidth.

Since the scratchpad does not have its own level in

[1]Note that the name "scratchpad" also refers to high speed internal memory used for temporary calculations [5], [29] which is a different technology than that discussed in this paper.

the cache hierarchy, when a record is evacuated from the cache there is an algorithmic decision whether to place the record in the scratchpad or directly into the DRAM. In the currently-proposed architecture designs, this decision is user-controlled.

Under the assumption that the memory is user-controlled, an algorithm must coordinate memory accesses from main memory and the scratchpad. Ideally the faster bandwidth of the scratchpad can be leveraged to alleviate the bandwidth bottleneck in applications. Unfortunately, known algorithmics do not directly apply to the proposed two level main memory architecture. The question looms, can an algorithm use the higher bandwidth scratchpad memory to improve performance? Unless this question is answered positively, the proposed architecture, with its additional complexity and manufacturing cost, is not viable.

Our interest in this problem comes from *Trinity* [30], the latest NNSA (National Nuclear Security Administration) supercomputer architecture. This supercomputer uses the Knight's Landing processor from Intel with Micron memory. This processor chip uses such a two-level memory [18]. The mission for the Center of Excellence for Application Transition [21], a collaboration between NNSA Labs (Sandia, Los Alamos, and Lawrence Livermore), Intel, and Cray, is to ensure applications can use the new architecture when it becomes available in 2015-16. This is the first work to examine the actual effect of the scratchpad on the performance of a specific algorithm.

A. Results

In this paper we introduce an algorithmic model of the scratchpad architecture, generalizing existing sequential and parallel external-memory models [1], [3]. (See Section VI for a brief background on the architectural design and motivation for the scratchpad.) We introduce theoretically optimal scratchpad-optimized, sequential and parallel sorting algorithms. We report on hardware simulations varying the relative-bandwidth parameter. These experiments suggest that the scratchpad will improve running times for sorting on actual scratchpad-enhanced hardware for a sufficient number of cores and sufficiently large bandwidth improvement.

Theoretical Contributions. We give an algorithmic model of the scratchpad, which generalizes existing sequential and parallel external-memory models [1], [3]. In our generalization, we allow two different block sizes, B and ρB ($\rho > 1$) to model the bandwidths of DRAM and the larger bandwidth of the scratchpad. Specifically, $\rho > 1$ is the relative increase in bandwidth of the scratchpad in comparison to DRAM.

We exhibit, under reasonable architectural assumptions of the scratchpad, sorting algorithms that achieve a ρ-factor speedup over the state-of-the-art sorting algorithms when DRAM is the only accessible memory. We begin by introducing a sequential algorithm for sorting, and then generalize to the multiprocessor (one multicore) setting. We complement this result by giving a matching lower bound in our theoretical model. Our algorithms supply theoretical evidence that the proposed architecture can indeed speed up fundamental applications.

Empirical contributions. After theoretically establishing the performance of our algorithms, we performed an empirical evaluation. The scratchpad architecture is yet to be produced, so we could not perform experiments on a deployed system. Instead, we extended Sandia National Laboratories' SST [25] simulator to allow for a scratchpad memory architecture with variable bandwidth, and ran simulations over a range of possible hardware parameters.

Our empirical results corroborate the theoretical results. We consider our algorithm in the cases where the application is memory-bandwidth bound. The benchmark we compare against our algorithm is the well-established parallel GNU multiway merge sort [27]. We show a linear reduction in running time for our algorithm when increasing the bandwidth from two to eight times. We also show that in the case where the application is not memory bound, then, as expected, we do not achieve improved performance.

Our empirical and theoretical results together show the first evidence that the scratchpad architecture could be a feasible way to improve the performance of memory-bandwidth-bound applications.

Algorithms and architecture co-design. This work is an example of co-design. The possible architectural design motivates algorithm research to take advantage of the new architectural features. The results, especially the experiments, help determine ranges of architectural parameters where the advantage is practical, at least for this initial application.

In particular, we find that the bandwidth-expansion parameter ρ introduced in the theoretical model also is an important performance parameter in our simulations. We estimate the ratio of processing capability to memory bandwidth necessary for sorting to become memory-bandwidth bound. From this, we estimate the number of cores that must be on a node at today's computing capability for the scratchpad to be of benefit. The core counts and minimum values of ρ could guide vendors in the design of future scratchpad-based systems.

The SST extension, designed and implemented by computer architects, is a co-design research tool. SST is an open source, modular framework for designing simulations of advanced computing systems [24], [25]. Once this extension is released, algorithms researchers will be able to test scratchpad algorithms before the hardware,

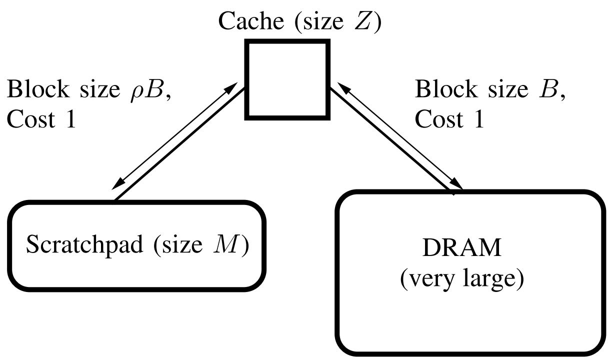


Fig. 1. The scratchpad memory model.

such as Knights Landing, is generally available.

II. Algorithmic Scratchpad Model

In this section, we propose an algorithmic model of the scratchpad, generalizing the external-memory model of Aggarwal and Vitter [1] to include high- and low-bandwidth memory.

Scratchpad model. Both the DRAM and the scratchpad are independently connected to the cache; see Figure 1. The scratchpad hardware has an access time that is roughly the same as that of normal DRAM—the difference is that the scratchpad has limited size but higher bandwidth. We model the higher bandwidth of the scratchpad by transferring data to and from the DRAM in smaller blocks of size B and to and from the scratchpad in larger blocks of size ρB, $\rho > 1$.

We parameterize memory as follows: the cache has size Z, the scratchpad has size $M \gg Z$, and the DRAM is modeled as arbitrarily large. We assume a tall cache; that is, $M > B^2$. This is a common assumption in external-memory analysis, see e.g. [6], [9], [10], [17].

We model the roughly similar access times of the actual hardware by charging each block transfer a cost of 1, regardless of whether it involves a large or small block. Performance is measured in terms of block transfers. Computation is modeled as free because we consider memory-bound computations.

Note that the scratchpad block size, ρB, in the algorithmic performance model need not be the same as the block-transfer size in the actual hardware. But for the purpose of algorithm design, one should program assuming a block size of ρB, so that the latency contribution for a transfer is dominated by the bandwidth component.

In the following, we say that event E_N on problem size N occurs *with high probability* if $\Pr(E_N) \geq 1 - \frac{1}{N^c}$, for some constant c.

Putting the scratchpad model in context. The external-memory model of Aggarwal and Vitter [1], assumes a two-level hierarchy comprised of a small, fast memory level and an arbitrarily large second level. Data is transferred between the two levels in blocks of size B.

Because the scratchpad model has two different bandwidths but roughly similar latencies, we have extended the external-memory model to have two different block sizes, each with the same transfer cost; see Figure 1.

There do exist other memory-hierarchy models, such as the cache-oblivious model [17], [23], which apply to multi-level memories. However, the different bandwidths but similar latencies means that the cache-oblivious model does not seem to apply. Instead, the scratchpad model can be viewed as a three-level model, but where two of the levels are in parallel with each other.

Background on external-memory sorting. The following well-known theorems give bounds on the cost of external-memory sorting.

Theorem 1 ([1]). *Sorting N numbers from DRAM with a cache of size Z and block size L (but no scratchpad) requires $\Theta\big((N/L)\log_{Z/L}(N/L)\big)$ block transfers from DRAM. This bound is achievable using multi-way merge sort with a branching factor of Z/L.*

Theorem 2 ([1]). *Sorting N numbers from DRAM with a cache of size Z and block size L (but no scratchpad) using merge sort takes $\Theta\big((N/L)\lg(N/Z)\big)$ block transfers from DRAM.*

In the analysis of sorting algorithms in the following sections, we will sometimes apply these theorems with $L = B$ and sometimes apply them with $L = \rho B$.

III. Sorting with a Scratchpad

This section gives an optimal randomized algorithm for sorting with one processor with a scratchpad. This algorithm generalizes multiway merge sort and distribution sort [1], which in turn generalize merge sort and sample sort [16].

We are given an array $A[1, \ldots, N]$ of N elements to sort, where $N \gg M$. For simplicity of exposition we assume that all elements in A are distinct, but this assumption can be removed.

The algorithm works by recursively reducing the size of the input until until each subproblem fits into the scratchpad, at which point it can be sorted rapidly. In each recursive step, we "bucket" elements so that for any two consecutive buckets (or ranges) R_i and R_{i+1}, every element in R_i is less than every element in R_{i+1}. We then sort each bucket recursively and concatenate the results, thus sorting all elements.

A. Choosing Bucket Boundaries

To perform the bucketing, we randomly select a sample set X of $m = \Theta(M/B)$ elements from $A[1, \ldots, N]$

and sort them within the scratchpad. (We assume sampling with replacement, but sampling without replacement also works.)

The exact value of m is chosen by the algorithm designer, but must be small enough to fit in the scratchpad.

Our implementation uses multiway mergesort from the GNU C++ library to sort within the scratchpad [11]. Other sorting algorithms could be used, such as quicksort. If ρ is sufficiently large, either sorting algorithm within the scratchpad leads to an optimal algorithm (see Theorem 6); however, the value of ρ based on current hardware probably is not large enough to make quicksort practically competitive with mergesort.

Corollary 3. *Sorting x elements that fit in the scratchpad uses $\Theta\big((x/\rho B)\log_{Z/B} x/B\big)$ block transfers using multi-way merge sort with a branching factor of Z/B, or $\Theta\big((x/\rho B)\lg (x/Z)\,\big)$ in expectation using quicksort. Both algorithms use $O(x\lg x)$ work.*

B. Bucketizing

We are given the sample set $X = \{x_1, x_2, \ldots, x_m\}$ $(x_1 < x_2 < \ldots < x_m)$ of sorted elements and input array A. Since $|X| = m$, X fits in the scratchpad, and A does not.

Our objective is to place each element $A[i] \in A \setminus X$ into bucket R_j if $x_j < A[i] < x_{j+1}$. (We may assume that $x_0 = -\infty$, and $x_{m+1} = \infty$.)

We perform a linear scan through A, ingesting $M - \Theta(m)$ elements into the scratchpad at a time. All of X remains in the scratchpad for the entire scan.

Once each new group of elements is loaded into the scratchpad, we sort them (along with X). Their position relative to the elements of X indicate which bucket they are stored in. The resulting bucketed elements get written back to DRAM, while the set X remains in the scratchpad. We call this process a *bucketizing scan.*

Lemma 4. *Each bucketizing scan costs:*

1) *$O(N/B)$ block transfers from DRAM,*
2) *$O(N/(\rho B)\log_{Z/(\rho B)} M/(\rho B))$ block transfers from the scratchpad, and*
3) *$O(N \lg M)$ operations in the RAM model.*

Proof: We select and sort the set X of sampled points in the scratchpad (necessary for mergesort), and then bucketize the rest of A. Assume for simplicity that $m \leq M/2$.

Sampling X requires $O(m)$ block transfers and $O(m)$ work, given constant time to choose a random word. Sorting X requires $O(m\lg m)$ work and $O(\frac{m}{\rho B}\log_{Z/B}\frac{m}{B})$ block transfers (via Corollary 3).

To bucketize, we read all elements from DRAM to the scratchpad in blocks of size $M - \Theta(m) \geq M/2$. This requires $O(M/B)$ block transfers from DRAM. The algorithm sorts each of these blocks in $O(M/(\rho B)\log_{Z/(\rho B)} M/(\rho B))$ scratchpad block transfers and requires $O(M\lg M)$ work to decide their buckets. There are $O(N/M)$ such groups, so this requires $O(N/B{+}N/(\rho B)\log_{Z/(\rho B)} M/(\rho B))$ total block transfers. We write each bucket to a separate piece of DRAM memory. This does not increase the cost: were the writing done to a contiguous location, it would require $\Theta(M/B)$ block transfers as the reading does. By writing the elements in $O(m) = O(M/B)$ pieces, we may incur up to two extra block transfers per piece: one to start each bucket, and perhaps another to end it. Thus the total number of transfers to write buckets to DRAM (even in seperate pieces) is $\Theta(M/B)$. ■

C. Bounding the Recursion Depth for Sorting

We now conclude our analysis of scratchpad sorting. Recall that the algorithm successively scans the input, refining it into progressively smaller buckets. Lemma 4 from the previous section gave a bound of the complexity of each scan.

In this section, we first show that the random samples are good enough so that with high probability each bucket is small enough to fit into cache after $O(\log_m(N/M))$ bucketizing scans. Then we give a bound on the sorting complexity.

This randomized analysis applies to any external-memory sort. However, to the best of our knowledge, this kind of analysis does not appear in the literature. Previous external memory sorting algorithms were deterministic. However, randomized algorithms are practical.

Lemma 5. *Each bucket fits into cache after $O(\log_m(N/M))$ bucketizing scans with high probability.*

Proof: Each newly-created bucket results from a *good* or *bad* split depending on its size. If a split makes a new bucket at least $\sqrt{m}$ factor smaller than the old one we call this a good split; otherwise it is bad.

We show that in a sufficiently large set of $O(\log_m(N/M))$ splits, there are $3\log_m(N/M)$ good splits with high probability. Thus, after $O\left(\log_m(N/M)\right)$ scans, each bucket is involved in at least $3\log_m(N/M)$ good splits, and fits in scratchpad.

We first show that the probability of a bad split is exponentially small in $\sqrt{m}$. Let R_i be the bucket being split. For simplicity, assume R_i lists the elements in sorted order, though the bucket itself may not be sorted yet. If a subbucket starting at j came from a bad split, no element in $\{R_i(j), R_i(j{+}1), \ldots, R_i(j{+}(|R_i|\sqrt{m}/m))\}$ is sampled to be in X. This happens with probability at most $(1-\sqrt{m}/m)^m \approx e^{-\sqrt{m}}$.

We next show that of any $(3/2)c\log_m(N/M)$ splits on a root-to-leaf path of the recursion tree, at most $c\log_m(N/M)$ are bad with high probability. Thus there are at least $(c/2)\log_m(N/M)$ good splits; choosing $c \geq 6$ proves the lemma.

By linearity of expectation, the expected number of bad splits is at most $(3/2)ce^{-\sqrt{m}}\log_m(N/M)$. Applying Chernoff bounds [20] with $\delta = (2/3)e^{\sqrt{m}} - 1$, we obtain

$$\Pr\left[\#\text{ bad splits } > c\log_m \frac{N}{M}\right] \leq \left(\frac{e}{2e^{\sqrt{m}}/3}\right)^{c\log_m \frac{N}{M}}.$$

We want to show that this is bounded above by $1/N^\alpha$. Taking the log of both sides of the inequality, this simplifies to

$$c\left((\sqrt{m}-1)\lg e - \lg 3/2\right)\log_m N/M \geq \alpha \lg N.$$

Now there are two cases. First, assume $M \geq \sqrt{N}$. Recall the tall cache assumption, that $M > B^2$. Then $\sqrt{m} = \sqrt{M/B} \geq M^{1/4} \geq N^{1/8}$. Substituting, we obtain $c((N^{1/8}-1)\lg e - \lg 3/2)\log_m N/M \geq \alpha \lg N$, which is true for sufficiently large N and $c = \alpha$.

On the other hand, let $M < \sqrt{N}$. Note that for $m \geq 2$, $((\sqrt{m}-1)\lg e - \lg 3/2)/\lg m \geq 1/2$. Then for $c \geq 4\alpha$,

$$c\left((\sqrt{m}-1)\lg e - 2\right)\log_m \frac{N}{M} \geq \frac{c}{2}\lg\frac{N}{M} \geq \alpha \lg N.$$

This shows that each bucket is sufficiently small with high probability after $O(\log_m N/M)$ bucketizing scans. To show that all must be small enough, take the union bound to obtain a probability no more than $1/N^{\alpha-1}$. Performing the analysis with $\alpha' = \alpha + 1$ gives us the desired bound. ■

Theorem 6. *There exists a sorting algorithm that performs $\Theta\left(\frac{N}{B}\log_{M/B}\frac{N}{B} + \frac{N}{\rho B}\log_{Z/\rho B}\frac{N}{B}\right)$ block transfers with high probability to partition the array into buckets, all of size $\leq M$. This bound comprises $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ block transfers from DRAM and $O(\frac{N}{\rho B}\log_{Z/\rho B}\frac{N}{B})$ high bandwidth block transfers from the scratchpad. There is a matching lower bound, so this algorithm is optimal.*

Proof: By Lemma 5 there are $O(\log_m N/M) \leq O(1 + \log_m N/M) = O(\log_{M/B} N/B)$ bucketizing scans; multiplying this by the block transfers per scan in Lemma 4 gives us the desired bounds. Note that once the buckets are of size $\leq M$, we can sort all of them with $O(\frac{N}{\rho B}\log_{Z/\rho B}\frac{M}{\rho B})$ additional block transfers by Corollary 3; this is a lower-order term.

To obtain the lower bound we will combine two lower bounds from weaker models. First consider a model where computation can be done in the scratchpad as well as in cache. Then the standard two-level hierarchy lower bound applies, and sorting requires $\Omega(\frac{N}{B}\log_{M/B}\frac{N}{B})$ block transfers from DRAM (since $M + Z = O(M)$). This bound is the same as the one given in Theorem 1, and can be found in [1].

Consider a second model where the DRAM allows high-bandwidth block transfers. Again, the standard two-level hierarchy lower bound applies, and sorting requires $\Omega(\frac{N}{\rho B}\log_{Z/(\rho B)}\frac{N}{\rho B})$ high-bandwidth block transfers.

Both of these bounds must be satisfied simultaneously. Thus, all together we require $\Omega(\frac{N}{B}\log_{M/B}\frac{N}{B} + \frac{N}{\rho B}\log_{Z/\rho B}\frac{N}{\rho B})$ block transfers. Since $(N/\rho B)\log_{Z/\rho B}\rho < N/B$, this simplifies to $\Omega\left(\frac{N}{B}\log_{M/B}\frac{N}{B} + \frac{N}{\rho B}\log_{Z/\rho B}\frac{N}{B}\right)$. ■

The following corollary explains how the sorting algorithm performs when quicksort is used within the scratchpad, and follows by linearity of expectation. The corollary shows that the algorithm remains optimal as long as the bandwidth of the scratchpad, determined by ρ, is sufficiently large.

Corollary 7. *An implementation of sorting using quicksort within the scratchpad uses $O(\frac{N}{B}\log_{M/B}\frac{N}{B} + \frac{N}{\rho B}\lg\frac{M}{Z}\log_{M/B}\frac{N}{B}))$ block transfers in expectation. This is optimal when $\rho = \Omega(\lg M/Z)$.*

IV. Sorting in Parallel

In this section, we show how to sort using a scratchpad that is shared by multiple CPUs. Our objective is to model one compute node of a scratchpad-enhanced supercomputer.

A. Parallel Scratchpad Model

The parallel scratchpad model resembles the sequential model, except that there are p processors all occupying the same chip. Each processor has its *own* cache of size Z, and all p processors are connected to a *single* size-M scratchpad and to DRAM. In DRAM the blocks have size B and in the scratchpad ρB.

In any given *block-transfer step* of the computation, our model allows for up to p' processors to perform a block transfer, either from DRAM or from the scratchpad. Thus, although there are p processors, bandwidth limitations reduce the available parallelism to some smaller value p'. (In the sequential case, $p' = 1$, and in the fully parallel case, $p' = p$.)

B. Other Parallel External-Memory Models

The parallel external-memory (PEM) model [3] has p processors each with its own cache of size M. These caches are each able to access an arbitrarily large external memory (DRAM in our terminology) in blocks of size B. These block transfers are the only way that processors communicate. This model works with CRCW, EREW, and CREW parallelism, though most research in this model (i.e. [2], [4], [28]) uses CREW.

Another line of study has examined *resource-oblivious* algorithms, which work efficiently without knowledge of the number of processors [7], [13], [14]. This work uses a similar model to PEM, but concentrates on the algorithmic structure. Subsequent work extended this to the hierarchical multi-level multicore (HM) model, featuring a multi-level hierarchy [12].

The Parallel Disk Model [32] is another model of a two-level hierarchy with multiple processors. However, this model has the important distinction of having multiple disks (multiple DRAMs in our terminology). Multiple disks—rather than a single, external memory—lead to fundamentally different algorithmic challenges.

Other parallel external-memory models take more parameters into account, like communication time between processors [31] and multilevel cache hierarchies [8].

C. Parallel Sorting

To obtain a parallel scratchpad sorting algorithm, we parallelize two subroutines from the sequential sort. In particular, we ingest blocks into the scratchpad in parallel, and we sort within the scratchpad using a parallel external-memory sort. These two enhancements reduce the bounds in Theorem 6 by a factor of p'.

We begin by analyzing the bucketizing procedure. We can randomly choose the elements of X and move them into the scratchpad in parallel.

To see how to sort in parallel, observe that when we have p processors, each with its own cache, but sharing the same scratchpad, we have the PEM model [3] and can apply PEM sorting algorithms.

Theorem 8 ([3]). *Sorting N numbers in the PEM model with p' processors, each with a cache of size Z and block size L, requires $\Theta\big((N/p'L)\log_{Z/L}(N/L)\big)$ block-transfer steps.*

By applying Theorem 8, we obtain a bound for a parallel bucketizing scan, replacing the bound from Lemma 4.

Lemma 9. *Each parallel bucketizing scan costs $O(N/(p'B)) + O(N/(p'\rho B)\log_{Z/(\rho B)} M/(\rho B))$ block-transfer steps, of which $O(N/(p'B))$ are from DRAM and $O(N/(p'\rho B)\log_{Z/(\rho B)} M/(\rho B))$ are from the scratchpad.*

Proof: Sampling to obtain X and moving X to the scratchpad requires $O(m/p')$ block-transfer steps.

During the scan, we bring each group of $M-m$ elements into the scratchpad in parallel, using $O(M/Bp')$ block-transfer steps. Then we sort these elements within the scratchpad using $O(\frac{M}{p'\rho B}\log_{Z/\rho B}\frac{M}{\rho B})$ block-transfer steps via Theorem 8. Multiplying by the number of groups of $M-m$ elements, $\Theta(N/M)$, gives the final bounds. ■

Combining with Lemma 5, we obtain the total number of block transfers.

Theorem 10. *Sorting on a node that allows p' processors to make simultaneous block transfers requires $O(\frac{N}{p'B}\log_{M/B}\frac{N}{B} + \frac{N}{p'\rho B}\log_{Z/(\rho B)}\frac{N}{B})$ block-transfer steps with high probability.*

D. Practical Sorting in Parallel

In the algorithm described above, after each "chunk" of values gets sorted and bucketized in the scratchpad, the elements in the scratchpad that belong in bucket i, are appended to an array in DRAM that stores all bucket i's elements. Empirically, the number of elements destined for any given bucket might be small, so these appends can be inefficient.

As a practical improvement, we modify the algorithm to do fewer small memory transfers to DRAM. The practical algorithm, denoted Near-Memory sort or *NMsort*, operates in two phases, depicted respectively in Figures 2 and 3 and described in detail below. In Phase 1 we sort the chunks of elements and store metadata associated with bucket information rather than explicitly creating individual buckets. The bucket metadata enables us to avoid small memory transfers. Without this innovation, we were unable to exploit the scratchpad effectively.

We start Phase 1 by loading a chunk of $\Theta(M)$ elements from DRAM into the scratchpad and sorting it using a parallel algorithm (Figure 2(a)). That sorted chunk can immediately be scheduled for transfer back to DRAM (Figure 2(b)). This transfer could happen asynchronously as work proceeds, but our prototype implementation simply waits for the transfer to complete. Thus, it is likely that the simulation results we present later could be nontrivially improved.

After sorting and writing, we extract bucket boundaries in parallel from the sorted pivots array. Instead of populating individual buckets with chunk elements at that point, we simply record the bucket boundaries. Specifically, for each $\Theta(M)$-sized chunk of values, we create an array of size $|X| = \Theta(M/B)$ where the ith element gives the index for the first element that belongs in bucket i. We then write out this auxiliary array, called *BucketPos* in Figure 2(c), to DRAM. We also keep global counts of the number of elements in each bucket in an array, called *BucketTot* in Figure 2(c). This array remains in scratchpad throughout both phases, and will hold the aggregate bucket counts from all sorted chunks.

We now consider the memory overhead (extra memory) of this method compared to storing bucket elements contiguously by writing to each bucket at each iteration. The chunks each have $\Theta(M)$ elements. The auxiliary array that gives bucket boundaries is of size $\Theta(M/B)$. If B (the cache line size) is 128, then the memory overhead is less than 1%, and larger cache lines reduce the relative

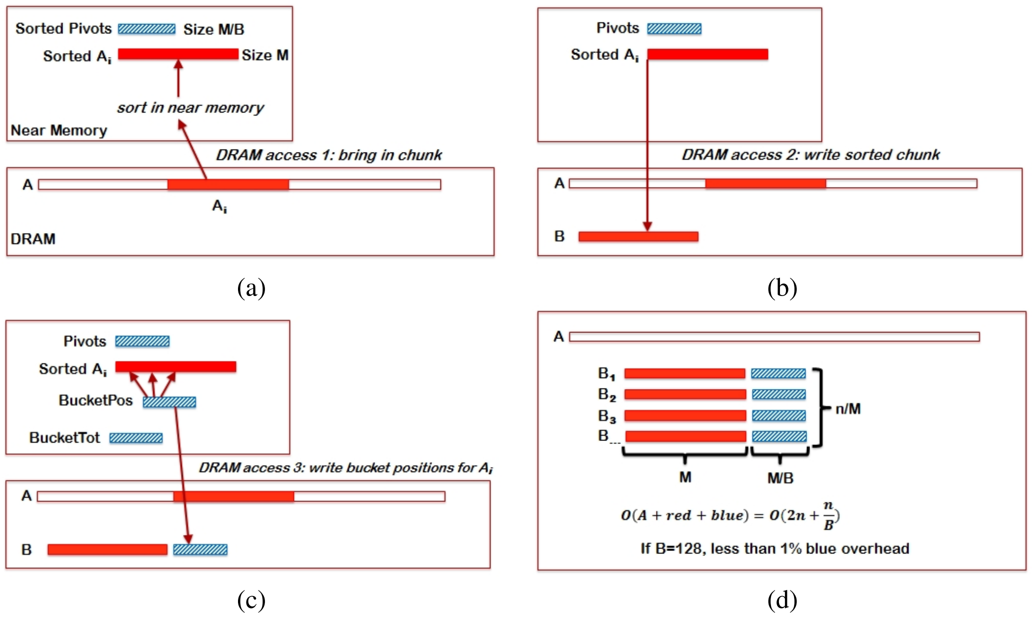

Sorted Pivots
Size M/B
Sorted A$_i$
Size M
sort in near memory
Near Memory
DRAM access 1: bring in chunk
A
A$_i$
DRAM
(a)
Pivots
Sorted A$_i$
DRAM access 2: write sorted chunk
A
B
(b)
Pivots
Sorted A$_i$
BucketPos
BucketTot
DRAM access 3: write bucket positions for A$_i$
A
B
(c)
A
B$_1$
B$_2$
B$_3$
B$_{...}$
n/M
M
M/B
$O(A + red + blue) = O(2n + \frac{n}{B})$
If B=128, less than 1% blue overhead
(d)

Fig. 2. Phase 1 of NMsort.

overhead. Figure 2(d) shows the state of DRAM at the end of Phase 1.

After sorting all $\Theta(N/M)$ chunks, we create the final array in Phase 2. Using the array *BucketTot*, we determine how many buckets can fit into the scratchpad. Specifically, we find the largest k such that $\sum_{i=0}^{k} \mathit{BucketTot}[i] \leq M$ (Figure 3(a)). We then read in all elements from buckets 0 though k from each of the chunks in DRAM (Figure 3(b)). If i is the index of the first element of bucket $k+1$, then we read in all elements with index $0 \ldots i$. We then use multi-way search to merge the $\Theta(N/M)$ sorted strings into a single sorted string. This is the start of the final list, which we write to DRAM, as shown in Figure 3(c). We continue with the next set of buckets that (almost) fills the scratchpad until we are done. For the examples we used in our experiments, we batched thousands of buckets into one transfer. This consolidation was a prerequisite for the simulated time advantage we show in Section V.

V. Experiments

We implemented the NMsort algorithm described in Section IV-D. As no hardware systems with multi-level memory subsystems of the type targeted by NMsort are currently available, we used the Structural Simulation Toolkit (SST) [25] as a basis for predicting performance.

The implementation of NMsort bucketizes once and then sorts the buckets. Because inputs for sorting are random 64-bit integers, the buckets will generally be about the same size, and we do not expect any buckets to be large enough to require recursion. The nonrecursive algorithm is sufficient for our experiments. The bucketization is calculated using a multithreaded algorithm that determines bucket boundaries in a sorted list.

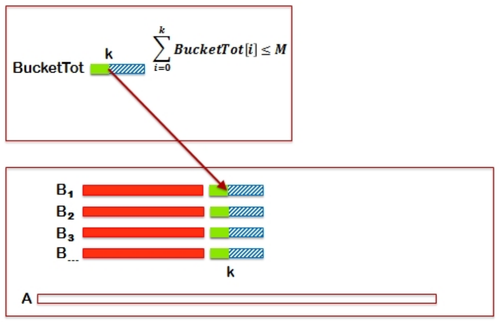
For a comparison to existing contemporary single-level memory sorting algorithms we use the GNU parallel C++ library's multi-way merge sort (originally from the MCSTL [27]). The latter is the fastest CPU-based multithreaded sort from our benchmarking exercises. We also employ the same calls to the GNU parallel sort as a subroutine call in NMsort for sorting integers once they are located in the scratchpad.

On current hardware, sorting may not be a memory-bandwidth-bound operation. However, we claim that it will become memory-bandwidth-bound in future high-performance-computing architectures as memory becomes more distant and the number of cores per NUMA region increases.

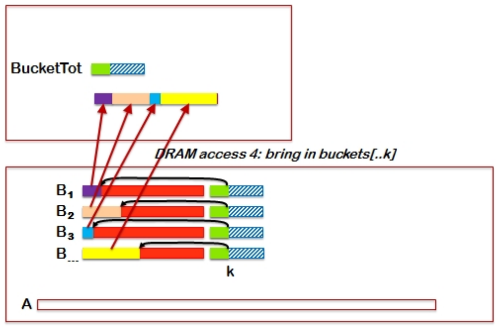
A. Experimental Design

We simulated a multi-level memory system with the parameters shown in Figure 4. In this system, we predict that sorting will be memory bound based on the following simple computation:[2] (1) given the available processing rate and the expected amount of work that needs to be done, compute the expected processing time, and (2) given the available memory bandwidth on/off-chip and the expected amount of memory transfers required, compute the expected time required for memory access. If the expected processing time is much less than

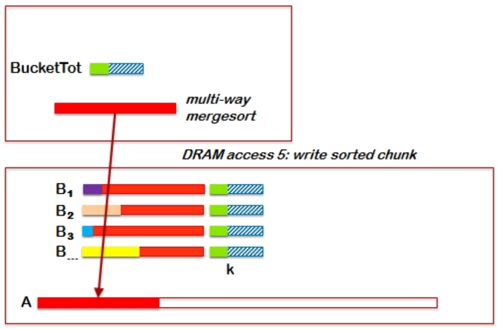
[2]We thank Marc Snir for personal communication suggesting when applications become bandwidth bound.



(a)



(b)



(c)

Fig. 3. The beginning of Phase 2 of NMsort.

- **Processor of 256-cores split into quad-core groups**
  - Cores run at 1.7GHz
  - Each core has a private 16KB L1 data cache
  - Each core group shared a 512KB L2 cache
  - Each core group has a 72GB/s connection to the on-chip network
- **Far (Capacity) Memory (e.g., DRAM)**
  - 1066MHz DDR configuration
  - 4 channels
  - 36GB/s connection per channel to the on-chip network
  - Approx. 60GB/s of STREAM Triad Performance
  - Block size 64 bytes
- **Scratchpad Near Memory**
  - 500MHz clock, 50ns constant access latency
  - 8, 16 or 32 channels of memory (2X, 4X and 8X bandwidth)
  - Block size 64 bytes

Fig. 4. Simulation system parameters.

the expected memory-transfer time, the application is memory-bandwidth bound.

Let x be the processing rate in operations per second (in our case, integer comparisons in the sort). Let y be the memory bandwidth between off-chip memory and on-chip cache (in array elements per second). Further, let N be the number of elements to sort and Z be the number of blocks in on-chip memory (i.e., L1, L2, L3 cache). Up to constants, we derive the inequality:

$$\frac{N \log N}{x} < \frac{N \log N}{y \log Z},$$

because there are $N \log N$ comparisons in the algorithm and the minimum aggregate memory transfer is $N \frac{\log N}{\log m}$ [1]. Before plugging in the numbers, we can rearrange the inequality:

$$y \log Z < x.$$

Thus, the sorting instance size is not a factor in this computation. In our experimental setup, we have (roughly), $Z \approx 10^6$, $x \approx 10^{10}$, and $y \approx 10^9$. Ignoring constants, we see that these quantities are comparable:

$$10^9 \log 10^6 \approx 10^{10}.$$

In our simulations, we observe that sorting is memory bound if the number of cores is 256 and not memory bound when that number is reduced to 128 (reducing x). We give results for the plausible case of 256 cores on-chip, as in Figure 4. In the future, it is expected that the number of cores per chip will grow past 256. When we run NMsort, we assume a near memory that can store several copies of an array of 10 million 64-bit integers.

To simulate this system, we use the Structural Simulation Toolkit (SST) [25], a parallel discrete event simulator that couples a number of architectural simulation components. The full processor is simulated by the Ariel

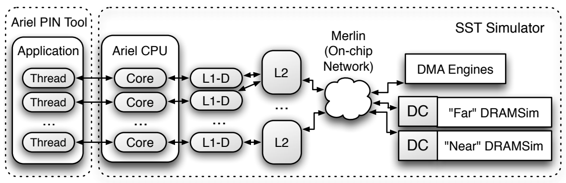


Fig. 5. Simulation Architectural Setup.

model which permits an arbitrary (in this case 256) number of cores to be modeled subject to a single executing application thread per core. The Ariel processor connects to an application running on real hardware through a shared-memory region for high performance.

Each application instruction is trapped using the Pintool dynamic instrumentation framework [22]. These instructions, including executing thread ID, and special operation codes including any specialized memory allocation/free functions, are routed through the shared memory queues. The virtual Ariel processor cores then extract instructions from their executing thread queue and issue memory operations into the memory subsystem as required.

Figure 5 shows the simulation configuration including the local, private L1 data caches, shared L2 caches and the on-chip network (named "Merlin") which allows routing of memory requests at the flit-level to the various memory regions. Near and far memory timings use the DRAMSim2 simulation component for increased accuracy [26].

B. Results

Results of our simulated NMsort algorithm appear in Table I. We observe increasingly positive results for NMsort as the algorithm is given a higher-performing scratchpad. Each NMsort column shows the bandwidth expansion factor (the ratio of the scratchpad to the DRAM bandwidth). This represents ρ from Section II.

A bandwidth expansion factor of 8 gives NMsort a wall-clock-time advantage of more than 25% compared to GNU sort. Furthermore, these simulations did not use direct memory access (DMA) transfers. We expect that specialized DMA capabilities could improve the performance of NMsort still further. Note that NMsort makes only half as many DRAM accesses as GNU sort. All of the sorting comparisons done by NMsort involve the scratchpad; the DRAM is used only for storing the blocks that do not fit in the scratchpad. We also note that related simulations with 128 cores rather than 256 are not memory-bandwidth bound and hence do not benefit from scratchpad usage. However, in our simulations the 128 runs are predicted to take less wall clock time. We believe that this anomaly is most likely explained by a lack of GNUsort strong scalability from 128 to 256 cores. Hence, two prerequisites for the successful application of NMsort are the compute-heavy, memory-bound systems anticipated in the future and scalable parallel sorting algorithms for the scratchpad sorts.

VI. Architecture of the Scratch Pad

In this section we give a brief overview of the memory architecture that underlies our theoretical model. We start with motivation: why have architects introduced this new architecture? Then we discuss specifics of the design of the architecture.

A. Motivation for the Scratchpad

There are multiple technologies that are in development that may be a basis for a level of memory between a processor's caches and conventional capacity memory. The WideI/O-2 standard, which is developed by JEDEC,[3] has memory parts with four ports that are 128-bits wide each. Such a memory thus has a bandwidth that is between two and four times higher than standard memory DIMMs. There are also several efforts underway that enable multiple memory chips to be "stacked" or vertically packaged on top of each other to reduce wire distances between components and improve performance. When such a part comes to market, it is expected that stacks of between four or eight dies will be economical. The result will be considerably higher bandwidth rates to these memory parts in relatively smaller spaces and lower power consumption than existing memory technologies.

However, the high pin density and electrical issues are likely to prevent stacked memory from becoming a complete replacement for existing memory DIMMs. Due to this limitation, the semiconductor industry is looking at mounting multiple die components either directly on top of CPU chips, or, more likely, as multiple dies that are mounted on the same substrate as the CPU. The result will be memory capacities of several gigabytes than can be implemented with higher bandwidths ("near memory") but, in the short term at least, the greatest portion of memory capacity will continue to be built using lower-bandwidth "far" memories such as existing DDR DIMMS (Figure 6).

Memory in the stacks described is likely to be organized differently than existing technologies where a single reference activates all dies on the referenced DIMM. In order to save power and provide greater parallelism, a reference will likely move data to/from a single memory die. With a sufficiently wide interface

[3]JEDEC (jedec.org) is a consortium that develops standards for the microelectronics industry. DDR3 and DDR4 memory result from standards developed by JEDEC, which also does many other standardization efforts.

	GNU Sort	NMsort (2X)	NMsort (4X)	NMsort (8X)
Sim Time (s)	898.419	756.731	693.889	640.126
Scratchpad Accesses	0	415669342	372160301	368351141
DRAM Accesses	394774287	161440225	162584052	158521515

TABLE I
SST SIMULATION RESULTS FOR VARIOUS SCRATCHPAD NEAR MEMORY BANDWIDTHS.

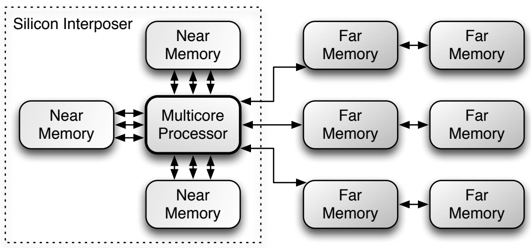


Fig. 6. Two Level Memory Architecture.

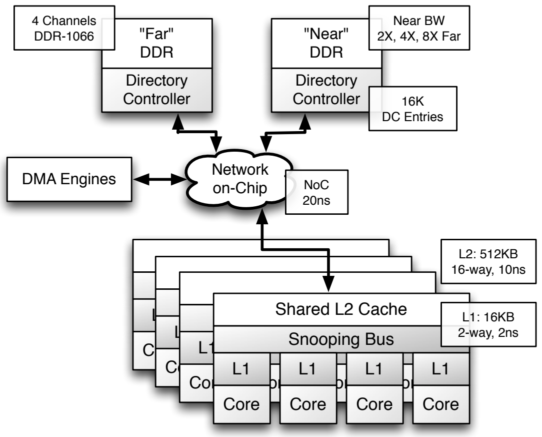


Fig. 7. Architectural Setup.

the memory will be able to provide increased bandwidth performance and somewhat reduced latency.

Our proposed scratchpad architecture bears some similarities to contemporary GPU memory architectures. The memory architecture of the GPU—like the scratchpad—has similar access times to main memory, but provides a higher bandwidth. Since the release of NVIDIAs CUDA-6 SDK, a feature called Unified Virtual Memory (UVM) has allowed for automatic migration of memory between the host DDR and the local GPU-based GDDR memories. However, it is generally accepted in the community that the current UVM implementation provided by NVIDIA can be outperformed by handwritten data movement operations in the general case [15]. As we move into the future where more complex and elaborate memory hierarchies are likely, our model will become increasingly relevant, particularly in multi-level memory scenarios where automatic migration of pages is unlikely to be available for all memory levels.

B. The DRAM Scratchpad Design

Architecturally, the scratchpad memory is attached behind a directory controller after the last level cache, as a "peer" to the conventional main memory. Our experiments assume a level-2 cache with separate L1 caches for each core in the CPU (See Figure 7).

The scratchpad memory appears as a separate portion of the physical address space. L2 cache can contain both conventional memory and scratchpad references. References to scratchpad data from the cache are functionally the same as references to main memory from the cache on the basis of a fixed address range.

In this work, the conventional and scratchpad memories are accessed through standard load/store instructions. In future work, we will examine the use of DMA engines which can transfer data between the near and far memory in the background, allowing overlap of computation and communication.

1) Why Not Simply a Large Cache. A multi-gigabyte cache would have to devote a substantial area to memory tags. This brings size and performance issues that would raise cost and complexity. Additionally, DRAM has fairly high access latencies which could make tag comparisons costly. SRAM memories with small latencies could be added to hold the tags, but this would have a significant impact on area and power.

For example, a 4GB DRAM cache, consisting of 2^{23} 64-byte cache lines might require 17 bits of SRAM tag and flag information per data block, or about 272 MB. If SRAM is about 1/24th the density of DRAM (an updated estimate from what is given in [19]), this would be almost 80% of the area of the DRAM data arrays and would consume much more power. Of course, larger page sizes could reduce this overhead, but that invites its own complexities.

By enabling an application to use an intermediate level of memory directly, the application can have a level of control such as prefetching data that is known to be needed and saving data that does not need further references. Another benefit is that the intermediate memory can be used to hold temporary values in calculations. This directly reduces the bandwidth needed to main memory, which will remain bandwidth limited.

2) Programmatic Interface. For this set of experiments it is assumed that the scratchpad is given a portion of memory address space, and that memory accesses (i.e., load/stores) treat each space identically.

We assume a modified `malloc()` call to allocate a portion of the scratchpad space and that the OS or runtime handles the virtual-to-physical mapping and contention between applications.

VII. Conclusions and future work

This paper gives a preliminary description of co-design work that gives evidence to support the potential usefulness of a scratchpad near memory for a single multicore node of a future extreme-scale machine. There are a number of ways to improve this work. For example, future systems could take advantage of DMA (direct memory access) to move large blocks between DRAM and scratchpad.

The sorting algorithms take advantage of algorithmically predictable prefetching. Besides the algorithms in this paper, we have preliminary algorithms for k-means clustering which take advantage of prefetching. All our k-means algorithms run a factor of ρ faster using scratchpad for many sizes of data and k. It remains to determine what other kinds of algorithms can run efficiently on a scratchpad architecture.

VIII. Acknowledgments

Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

This research is also supported in part by NSF grants CNS 1408695, CCF 1439084, IIS 1247726, IIS 1251137, CCF 1217708, and CCF 1114809.

References

[1] Aggarwal, A., and Vitter, J. S. The input/output complexity of sorting and related problems. *Communications of the ACM 31*, 9 (Sept. 1988), 1116–1127.

[2] Ajwani, D., Sitchinava, N., and Zeh, N. Geometric algorithms for private-cache chip multiprocessors. In *Proceedings of the Eighteenth Annual European Symposium on Algorithms (ESA)*. 2010, pp. 75–86.

[3] Arge, L., Goodrich, M. T., Nelson, M., and Sitchinava, N. Fundamental parallel algorithms for private-cache chip multiprocessors. In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures (SPAA)* (2008), pp. 197–206.

[4] Arge, L., Goodrich, M. T., and Sitchinava, N. Parallel external memory graph algorithms. In *Proceedings of the IEEE International Symposium on Parallel & Distributed Processing (IPDPS)* (2010), pp. 1–11.

[5] Banakar, R., Steinke, S., Lee, B.-S., Balakrishnan, M., and Marwedel, P. Scratchpad Memory: A Design Alternative for Cache On-chip Memory in Embedded Systems. In *Proceedings of the Tenth International Symposium on Hardware/Software Codesign (CODES)* (2002), pp. 73–78.

[6] Bender, M. A., Ebrahimi, R., Fineman, J. T., Ghasemiesfeh, G., Johnson, R., and McCauley, S. Cache-adaptive algorithms. In *Proceedings of the Twenty-Fifth Symposium on Discrete Algorithms (SODA)* (2014), pp. 116–128.

[7] Bilardi, G., Pietracaprina, A., Pucci, G., and Silvestri, F. Network-oblivious algorithms. In *Proceedings of the Twenty-First International Parallel & Distributed Processing Symposium (IPDPS)* (2007), pp. 1–10.

[8] Blelloch, G. E., Fineman, J. T., Gibbons, P. B., and Simhadri, H. V. Scheduling irregular parallel computations on hierarchical caches. In *Proceedings of the 23rd Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)* (2011), pp. 355–366.

[9] Brodal, G. S., Demaine, E. D., Fineman, J. T., Iacono, J., Langerman, S., and Munro, J. I. Cache-oblivious dynamic dictionaries with update/query tradeoffs. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)* (2010), pp. 1448–1456.

[10] Brodal, G. S., and Fagerberg, R. On the limits of cache-obliviousness. In *Proceedings of the Thirty-Fifth Annual ACM Symposium on Theory of Computing (STOC)* (2003), pp. 307–315.

[11] Carlini, P., Edwards, P., Gregor, D., Kosnik, B., Matani, D., Merrill, J., Mitchell, M., Myers, N., Natter, F., Olsson, S., Rus, S., Singler, J., Tavory, A., and Wakely, J. *The GNU C++ Library Manual*. 2012.

[12] Chowdhury, R. A., Ramachandran, V., Silvestri, F., and Blakeley, B. Oblivious algorithms for multicores and networks of processors. *Journal of Parallel and Distributed Computing 73*, 7 (2013), 911–925.

[13] Cole, R., and Ramachandran, V. Resource oblivious sorting on multicores. In *Proceedings of the 37th International Colloquium Conference on Automata, Languages and Programming (ICALP)*. 2010, pp. 226–237.

[14] Cole, R., and Ramachandran, V. Efficient resource oblivious algorithms for multicores with false sharing. In *Proceedings of the Twenty-Sixth International Parallel & Distributed Processing Symposium (IPDPS)* (2012), pp. 201–214.

[15] Edwards, H. C., Trot, C., and Sunderland, D. Kokkos, a manycore device performance portability library for C++ HPC applications. Presented at GPU Technology Conference, 2014.

[16] Frazer, W. D., and McKellar, A. C. Samplesort: A sampling approach to minimal storage tree sorting. *Journal of the ACM 17*, 3 (1970), 496–507.

[17] Frigo, M., Leiserson, C. E., Prokop, H., and Ramachandran, S. Cache-oblivious algorithms. *ACM Transactions on Algorithms 8*, 1 (2012), 4.

[18] http://www.hpcwire.com/2014/06/24/micron-intel-reveal-memory-slice-knights-landing/.

[19] Kogge, P. Exascale computing: embedded style, 2009. Slides from a talk given at the Fault-Tolerant Spaceborne Computing Employing New Technologies Workshop, Sandia National Laboratories.

[20] Motwani, R., and Raghavan, P. *Randomized algorithms*. Chapman & Hall/CRC, 2010.

[21] http://nnsa.energy.gov/mediaroom/pressreleases/trinity.

[22] Patil, H., Cohn, R., Charney, M., Kapoor, R., Sun, A., and Karunanidhi, A. Pinpointing representative portions of large Intel Itanium programs with dynamic instrumentation. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2004), pp. 81–92.

[23] Prokop, H. Cache-oblivious algorithms. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, June 1999.

[24] Rodrigues, A., Murphy, R., Kogge, P., and Underwood, K. The Structural Simulation Toolkit: A Tool for Exploring Parallel Architectures and Applications. Tech. Rep. SAND2007-0044C, Sandia National Laboratories, 2007.

[25] Rodrigues, A. F., Hemmert, K. S., Barrett, B. W., Kersey, C., Oldfield, R., Weston, M., Risen, R., Cook, J., Rosenfeld, P., CooperBalls, E., and Jacob, B. The

Structural Simulation Toolkit. *SIGMETRICS Perform. Eval. Rev. 38*, 4 (Mar. 2011), 37–42.

[26] ROSENFELD, P., COOPER-BALIS, E., AND JACOB, B. DRAMSim2: A cycle accurate memory system simulator. *IEEE Comput. Archit. Lett. 10*, 1 (Jan. 2011), 16–19.

[27] SINGLER, J., SANDERS, P., AND PUTZE, F. MCSTL: The multi-core standard template library. In *Proceedings of the 13th International Euro-Par Conference on Parallel Processing (Euro-Par)*. 2007, pp. 682–694.

[28] SITCHINAVA, N., AND ZEH, N. A parallel buffer tree. In *Proceedings of the Twenty-Fourth ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)* (2012), pp. 214–223.

[29] STEINKE, S., WEHMEYER, L., LEE, B., AND MARWEDEL, P. Assigning program and data objects to scratchpad for energy reduction. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)* (2002), pp. 409–415.

[30] http://insidehpc.com/2014/07/cray-wins-174-million-contract-trinity-supercomputer-based-knights-landing.

[31] VALIANT, L. G. A bridging model for multi-core computing. In *Proceedings of the Sixteenth Annual European Symposium on Algorithms (ESA)*. 2008, pp. 13–28.

[32] VITTER, J. S., AND SHRIVER, E. A. Algorithms for parallel memory, I: Two-level memories. *Algorithmica 12*, 2-3 (1994), 110–147.