ACDT: Architected Composite Data Types
Trading-in Unfettered Data Access for Improved Execution

Andres Marquez*, Joseph Manzano*, Shuaiwen Leon Song*, Benoît Meister†
Sunil Shrestha‡, Thomas St. John‡ and Guang Gao‡
*Pacific Northwest National Laboratory
{andres.marquez,joseph.manzano,shuaiwen.song}@pnnl.gov
†Reservoir Labs
meister@reservoir.com
‡University of Delaware
{shrestha,stjohn,ggao}@capsl.udel.edu

***Abstract*—**

With Exascale performance and its challenges in mind, one ubiquitous concern among architects is energy efficiency. Petascale systems projected to Exascale systems are unsustainable at current power consumption rates. One major contributor to system-wide power consumption is the number of memory operations leading to data movement and management techniques applied by the runtime system. To address this problem, we present the concept of the Architected Composite Data Types (ACDT) framework. The framework is made aware of data composites, assigning them a specific layout, transformations and operators. Data manipulation overhead is amortized over a larger number of elements and program performance and power efficiency can be significantly improved.

We developed the fundamentals of an ACDT framework on a massively multithreaded adaptive runtime system geared towards Exascale clusters. Showcasing the capability of ACDT, we exercised the framework with two representative processing kernels - Matrix Vector Multiply and the Cholesky Decomposition – applied to sparse matrices. As transformation modules, we applied optimized compress/decompress engines and configured invariant operators for maximum energy/performance efficiency. Additionally, we explored two different approaches based on transformation opaqueness in relation to the application. Under the first approach, the application is agnostic to compression and decompression activity. Such approach entails minimal changes to the original application code, but leaves out potential application-specific optimizations. The second approach exposes the decompression process to the application, hereby exposing optimization opportunities that can only be exploited with application knowledge. The experimental results show that the two approaches have their strengths in HW and SW respectively, where the SW approach can yield performance and power improvements that are an order of magnitude better than ACDT-oblivious, hand-optimized implementations. We consider the ACDT runtime framework an important component of compute nodes that will lead towards power efficient Exascale clusters.

I. Introduction

As we strive to bring to bear the Exascale era, we are still confronted with the arduous task to tame the expected power hunger of these machines. The challenge is to get into a regime of ∼20MW [1] whereas current projections consider an order of magnitude higher to be realistic. To get into the required power regime, we expect that massively multithreaded systems with billion way concurrency and HW/SW technologies will reduce or avoid energy costs of data transfers as we move data around. These techniques will complement data transfer reduction approaches associated with improved data locality, obtained through optimized data and computation distribution. In the SW-stack we foresee the runtime system to have a particular important role to contribute to the solution of the power challenge. It is here where the massive concurrency is managed and where judicious data layouts [11] and data movements are orchestrated. With that in mind, we set to investigate on how to improve efficiency of a massively multithreaded adaptive runtime system in managing and moving data, and the trade-offs an improved data management efficiency requires. Specifically, in the context of run-time system (RTS), we explore the power efficiency potential that data compression has to offer under various degrees of opaqueness to the application.

Previous studies [20] [10] [24] suggest that data movement across the memory hierarchy and across nodes not only impact the overall performance significantly but also contribute greatly to overall power consumption. It stands to reason that avoiding or reducing [9] data movement will be advantageous for either power and/or performance efficiency. In order to reduce data movement, this work explores opportunities to occasionally decorate composites – i.e., vectors, structures – with additional type attributes used solely by the RTS for optimization purposes. We call this concept "Architected Composite Data Type" [ACDT] – as reference, we denote an untouched composite as "Unstructured Composite Data Type" [UCDT].

ACDT associates type information to the whole composite that goes beyond element type information. Important attributes of a composite are size and shape (e.g., tiling). Additional attributes are related to data format, residency and lifetime (stays in the memory hierarchy), algebraic and topological matrix properties (value types and distribution), as well as dynamic range distribution and access/reuse pattern. ACDT is known to the runtime. One important aspect of ACDT is that it enables the runtime to reason about composites as a whole, e.g., weighing-in the benefits and detriments of encapsulating a composite: *Hereby, ACDT trades in unfettered access to elements of the composite via primitive operators/load&stores in favor of architected operators/gets&puts that act on the whole composite*. Encapsulated composites are beneficial because their architected operators, compared to their elemental primitive counterparts, better amortize overheads and improve

efficiency in the runtime of data manipulation algorithms such as data compression/decompression [15], reshaping [16], marshaling [21], redundancy and rematerialization [7] to name a few. All these algorithms have a direct impact on data management and movement. In summary, contrary to the *top-down* approach that high-level languages do to bridge the semantic gap, ACDT's *bottom-up* approach is to improve the efficiency of data manipulation algorithms in the runtime system.

Algorithm 1 Matrix Vector Multiply

```
for i = 1 to m do
  for k = 1 to n do
    c[i] += A[i][k]* b[k]
  end for
end for
```

Consider Algorithm 1 as an example, showcasing a simple Matrix Vector Multiply (MVM). The composites are a 2D matrix **A**, an input vector **b** and an output vector **c**. In the first case *I*, we assume the density of the three composites to be dense, meaning the number of non-zero elements for each composite is larger than a specified threshold. In the second case *II*, we assume that matrix **A** is non-dense (i.e., sparse), following the same threshold criterion. We will further assume that the composite sparsity information is an attribute of an ACDT, as is residency, lifetime and access/reuse pattern. This information provides opportunities to the RTS for operator selection, data transformation and data movement that are intended to yield amortized savings across various elements. Determining **A**'s ACDT gives multiple options to the runtime system: (A) Based on **A**'s sparsity, the runtime system could alternatively choose to use an operator implementation that relies on a Vector or SIMD implementation for improved performance; (B) based on scheduling status, the runtime could decide to postpone the execution and/or ship the data to a remote resource; (C) based on (A), (B) and data reuse, the runtime could decide to use compression. However, the right course of action is dependent on the overhead as well as the amortized benefits of the applied transformations and operators. We provide a detailed cost/benefit analysis for MVM in Section V-A.

In this paper, we propose the ACDT framework which addresses the importance of data management/ movement and explores opportunities to improve power and performance efficiency. We showcase that our framework can improve performance and power efficiency of a massively multithreaded adaptive runtime system by an order of magnitude for a selected Cholesky decomposition algorithm. This paper makes the following contributions:

1) We developed an ACDT conceptual framework on a massively multithreaded adaptive runtime system geared towards Exascale systems.
2) We tested the framework with two representative processing kernels - Matrix Vector Multiply and the Cholesky Decomposition, both running sparse matrices. We opted for compress/decompress [C/D] engines as transformations and selected composite specific operators for performance/power improvement.
3) We developed two different ACDT approaches based on transformation opaqueness in relation to the application. We demonstrated that the two approaches show promise in hardware and software respectively, where the software approach can yield performance and power improvements that are an order of magnitude better than implementations without ACDT.

The remainder of the paper is structured as follows: We provide an overview of related work that is tangential or has a direct impact on our work in Section II. Next, we give an overview of our ACDT framework in Section III where we reiterate the concepts of ACDT and explore the creation and maintenance of these composite data types. It follows a framework use case in Section IV where we introduce the concepts and tools necessary to run our experiments. Experimental setup, results and discussion of our compression/decompression approaches and results on Matrix Vector Multiply and Cholesky Decomposition are shown in Section V. We end with future work and conclusions in Section VI.

II. Related Work

A. Massively Multithreaded Adaptive Runtime Systems

Massively Multithreaded Runtime Systems are capable of scheduling thousands of threads, managing memory bandwidth and contention avoidance. For instance, Cilk [4] is a multithreaded parallel programming model with an associated runtime system. Threads are dispatched by a work-stealing scheduler, executing in nonblocking fashion. Cilk's strength lies in its performance predictive capability that accounts for the amount of work and the critical path. The execution model is limited to processing a tree-like task graph that requires "fork-join" synchronization at each level. X10/Habanero [6] relaxes the requirement to "join" at each level. Supporting a more general execution model, the Efficient Architecture for Running THreads (EARTH) [22] schedules DAGs and uses uninterruptable fine grain threads, called fibers, that share activation frames. Under this model, data is either passed around as output from fibers or explicitly moved with asynchronous or synchronous functions. The generality is bought at the expense of limiting performance prediction. In addition, this type of model puts the entire responsibility of expressing parallelism and memory access solely on the fiber programmers. The SWARM framework [8] builds on EARTH concepts, adding support for a type system, direct access shared memory structures, placement information and syntactic sugar. As before, the burden is on the programmer to express parallelism and synchronization. Similar in concept, a recent ongoing development is the Open Community Runtime (OCR) [19] system. OCR has the concept of control- and data-flow event driven tasks. Data is passed between tasks with the support of memory data blocks. OCR is an open platform to experiment with adaptive runtime systems at extreme scale. Neither of the above mentioned RTS offers performance and power efficiency adaptation due to composite data properties. This work offers a remedy.

B. Compression & Decompression

1) Compression Algorithms: Besides powerful compression techniques used in commercial software, like dictionary based methods (Lempel-Ziv and others), a few selected compression algorithms are used for online compression of scientific data. These algorithms are characterized by an affinity

for small blocks, an ease to interface with synchronous devices (busses, memory controllers, etc), being lossless, and with fast and efficient methodologies that could be easily implemented in hardware.

The XMatch with Run Length (XRL) encoding algorithm [13] depends on the construction of a dynamic dictionary that assigns shorter codewords to frequently appearing patterns. Before the XMatch phase starts, the data stream is preconditioned with a run length encoder that is sensitive to long chains of zeroes. An entry in the dictionary (with an associated codeword) is especially reserved for these zero sequences. XRL is used in the selective compressed memory system [15]. In this system, the level 1 caches are uncompressed and the level 2 caches uses the XRL for each data block. Due to its ease-to-implement, we selected the run length encoding part of this algorithm as a compression engine for one of the ACDT types in Section V-A.

The floating point compression algorithm (FPC) was introduced by Burtscher and Ratanaworabhan [5]. In this algorithm, when a new floating point value arrives, two predictors are used to guess this number. Afterwards, a quick comparison to the original value takes place and the smallest residual is encoded. The predictor used, the residual and the number of leading zeroes are encoded in the final compressed data. The use of the predictor tables allow this algorithm implementations to catch a wide range of patterns. For this reason, we selected this algorithm to showcase a more sophisticated compression engine in support of the ACDT type presented in Section V-A.

2) Compression formats: The application designers can tailor their algorithms and data structures when they have the knowledge of the sparsity of the workloads. These techniques usually take the form of compressed format representations for the structures of interest. One of the famous methods is the Compressed Sparse Row (CSR) format [2]. In this format, the sparse matrix is stored in three arrays: **val**, **ptr** and **col**. Array val is dedicated to save the non-zero elements of the structure in row major order. The other two arrays contain positional information. The ptr array contains the location on val of the first non-zero element of each row. The second array, named col, contains the column indexes of the elements of val in the original array. The Block Compressed Sparse Row [18] approach takes the idea of CSR but instead of values, block of data are saved in the val array. A disadvantage of this approach is that, depending upon the shape of the matrix, extra padding might be required. Finally, some tiles might be mostly composed of zeroes since the internal structure is not exploited. Compressed Sparse eXtended [14] format exploits the inner structures of non-zero elements inside a sparse matrix. However, all of these formats are best used as static (read-only) formats. In this work, we modified the Block Compressed Sparse Row representation to make it amenable to online usage in Section V-B.

C. Power Efficiency

Optimizing data movement of Exascale applications for high performance and energy efficiency has been defined as one of the major challenges and key research areas by DARPA [3]. There has been some recent work focusing on characterizing energy data movement consumption across different memory hierarchies. Molka et al. [17] have characterized the energy efficiency of arithmetic and memory operations across multiple cache levels. Based on micro-benchmarks, they built a basic model to approximate the energy consumption of data transfers from cache levels to main memory. Kestor et al. [12] have also proposed a more elaborate energy model that not only approximates the energy consumption but also considers the energy cost of stalled cycles and the impact of prefetching on energy waste. Unlike our approach, none of the work above has proposed an adaptive runtime system level framework to manage various data-movement optimization techniques that hides non-trivial architectural details from users.

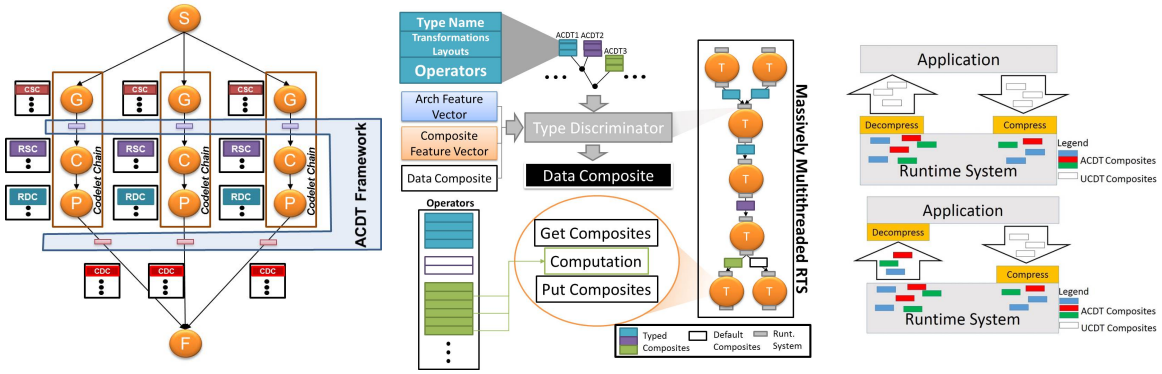
III. Framework Overview

The overview of the conceptual framework is shown in Figures 1(a),1(b). In Figure 1(a) we depict the MVM dataflow graph with threads as vertices and dependencies as edges. **S**tartup and **F**inalize threads are located top and bottom, respectively. The **A** matrix is tiled according to size and shape and distributed to multiple thread chains. Each thread chain is comprised of a **G**et, a **C**omputation and a **P**ut phase. In the Get phase the tile is read in its entirety into the thread chain context. The Computation operates on the tile and the Put phase discharges the whole tile out of the thread context. Notice how the ACDT framework introduces additional threads into the graph (shown as small bars) that operate on the tiles – in this example, transparently to the application. The contractual agreement between the ACDT framework and application requires that the application follow the abstract thread model of Get/Put phases that transfer whole tiles. The ACDT framework now has the freedom to view these tiles as composites, associating feature attributes encoded as meta-data at compile or runtime derived from size, shape, algebraic or topological properties, constant values, expected lifetimes and access/reuse pattern. In our MVM example, we use size and shape in conjunction with architectural features to guide tiling. The topological properties are used to transparently guide compression as discussed further in the experimental Section V-A.

Figure 1(b) depicts the inner details of the framework. On the right side we have an excerpt of a dataflow graph as seen by the runtime system. Each actor in the graph is represented by a thread chain (yellow circle with a 'T') that communicates with other thread chains via runtime mediation (grey rectangles). ACDT composites and UCDT (Unstructured Composite Data Type) default composites are passed between threads in dataflow fashion. Center and top left in the figure showcase the role of the runtime system: a type discriminator samples the stream of feature vector meta-data associated with a composite and the composite itself. The discriminator determines type changes and appropriate operators that are suitable for respective ACDT data layouts.

IV. ACDT Framework Use Case

For the remainder of this paper we will consider opportunities that ACDT offers for compression as a data manipulation technique on a massively multithreaded runtime system. We will consider two use cases, Matrix Vector Multiply and Cholesky Decomposition to guide the reader through the methodology. Each use case will expose a different compression approach, Balanced- and Unbalanced-Compress/Decompress, whose mechanisms will be described next.



(a) MVM in ACDT framework (b) Overview of ACDT framework (c) balanced (top) & unbalanced C/D

Fig. 1. ACDT framework

TABLE I. COMPOSITE INVARIANT ALGEBRAIC OPERATORS

CIAO	Composite
x	diag(-1), diag(1)
+, x	matrix(0)

A. ACDT for Compression

1) Balanced Compress/Decompress [bC/D]: With "Balanced Compress/Decompress" (*b*C/D) we denote a class of compression algorithms where both parts, compression and decompression, run at the *same level* of the software stack. With *b*C/D it is possible to subject composites to data transformations completely transparent to the application. In that case, any resulting ACDT operators will not be visible to the application since the application is oblivious to the underlying transformations. Figure 1(c)(top) exemplifies this method. Compress and decompress engines are not located with the application. ACDT composites (colored bricks) are transformed into their UCDT composite counterparts (uncolored bricks) as they are read from memory and shipped to the application. On their way back to memory, the UCDT composites are converted back into ACDT. Section V-A showcases the use of this approach.

2) Unbalanced Compress/Decompress [uC/D]: With "Unbalanced Compress/Decompress" (*u*C/D) we denote a class of compression algorithms where both parts, compression and decompression, run at *different levels* of the software stack. The part that runs at application level will have access to ACDT operators, if applicable. Figure 1(c)(bottom) showcases this method. Only the Compress engine is located with the runtime system, in order to run in the background. The Decompress engine is located with the application. Information associated with the ACDT composites is visible to the application and can be used advantageously. Section V-B demonstrates the use of this approach.

3) Composite Invariant Algebraic Operators [CIAO]: The *u*C/D approach is attractive because decompressing at the application level makes ACDT operators available to the application as seen in Figure 1(c). Composite algebraic invariants can be detected with almost no overhead since repeating values are a fundamental compression resource of any C/D engine. Tab. I showcases the composite invariants and their associated algebraic operators. Section V-B exploits CIAOs to obtain performance and power improvements by eliminating redundant computation [25].

B. C/D Latency Model

In order to help determine first order benefits of one C/D implementation over another, we developed a simple latency model tracking first order effects of DRAM memory and a set of compress/decompress engines commonly found in literature.

The DRAM latency model assumes symmetric read/write access and no contention: The memory holds a large matrix with square tiles s.t. each new tile vector of the matrix's leading dimension is placed and fits in a different memory row. Furthermore, we will assume that the number of tile elements N is a large multiple of the memory burst size B_{col}. The resulting memory access latency Lat_{mem} to move a tile into the cache is a function of the refresh C_{ref} and the row C_{row} and column C_{colb} (per element amortized burst mode) costs respectively (Eq. 1-2).

$$B_{col} \ll N; f_{ref}(N) \ll N \tag{1}$$

$$Lat_{mem} \cong f_{ref}(N) * C_{ref} + \sqrt{N} * C_{row} + N * C_{colb} \tag{2}$$

$$Overhead_{dec} = Fix + N_c * D \quad with\ Fix \ll N_c * D \tag{3}$$

$$Overhead_{dec} \cong N_c * D \tag{4}$$

$$\overline{Overhead_{dec}} = Overhead_{dec}/Reuse \tag{5}$$

$$Lat_{mem\_u} > Lat_{mem\_c} + Overhead_{dec} \tag{6}$$

The compression and decompression engines under this framework are designed to support high throughput, amenable to SW or HW engines, relying on a few simple ALU operators such as *shifts* or *xor*. Hence, the decoding overhead $Overhead_{dec}$ for a tile is $\Theta(N_c)$ in the number of compressed tile elements N_c under the assumption that we size our tiles such that the fixed cost Fix can be neglected in comparison to the decoding cost D (Eq. 3-4). Reusing the tile $Reuse$ times per single decode further reduces the overhead yielding an amortized overhead $\overline{Overhead_{dec}}$ (Eq. 5).

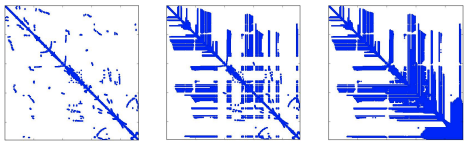
Armed with these formulas, we are now able to formulate the condition under which it is beneficial to engage a C/D encoding scheme for our ACDT framework: Let N_u be the number of elements of an uncompressed tile and as before, N_c the number of compressed elements for the same tile. Memory access latencies are denoted by Lat_{mem_u} and Lat_{mem_c} respectively for the uncompressed and compressed cases. Benefits start to accrue when Eq. 6 is met. We use the C/D latency model in our discussion section to reason about trade-offs.

C. Runtime System

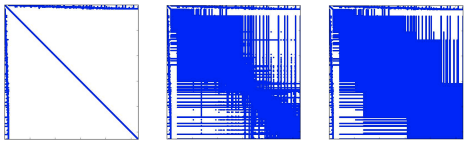
In order to explore Exascale software stacks whose initial implementation we expect in the 2017 timeframe, we choose SWARM [8] as X in MPI + X and extend it with the ACDT framework. In this programming paradigm, the inter-node communication is handled by MPI calls and the intra-node aspects are handled by the X programming/execution model. Like several other recent advanced runtimes (including the Open Community Runtime (OCR), Intel CnC, and OpenStreams), SWARMs multithreaded execution model supports asynchronous task spawning, dynamic load balancing as well as point-to-point task synchronization. These features are necessary to expose all the task parallelism in programs, a strict requirement at Exascale. One advantage of SWARM which made us originally choose it over other runtimes is that it offers a rich set of synchronization mechanisms. Nevertheless, we expect results of these studies to impact currently more popular, yet more restrictive execution models as well e.g. Cilk and OpenMP.

D. ACDT Discriminator

This section provides an overview of an ACDT discriminator with the ability to select between different compression techniques. It would run in the background and provide guidance for selection based on algorithmic and architectural state throughout the execution process. It requires detecting different data patterns at runtime and selecting an efficient compression technique based on available feature vectors. Here we conduct analysis on the determination of the topology of two initially sparse matrices (bcsstk36 and bundle1) from Table II to demonstrate how various sampling techniques affect the partial knowledge of the composite state in the ACDT framework.



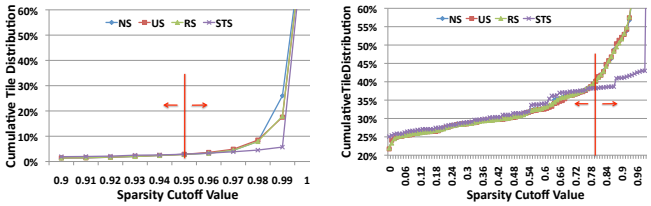
(a) I=128(99.78%) (b) I=8K(98.7%) (c) I=22K(96.8%)



(d) I=0(99.3%) (e) I=1K(63%) (f) I=10K(36.1%)

Fig. 2. Bcsstk36 and Bundle1 sparsity evolution

Using uniform sampling (US), random sampling (RS) and sub-composite sampling (STS), we ran our experiments with different sampling ratios and composite sizes and compared them against the non-sampled (NS) data. Our results show very close approximation of cumulative composite distribution based on sparsity for the UFL matrices [23] as well as intermediate matrices produced during execution. For instance, Figures 2(a), 2(b) and 2(c) show gradual progression of sparsity for matrix bcsstk36 at iteration 128, 8K and 22K for the Cholesky Decomposition algorithm used in our experiment (Section V-B). Sparsity in this example reduces very slowly over time (99.7%, 98.7%, 96.8%). However, such progression of sparsity can vary per application. Figures 2(d), 2(e) and 2(f) show sparsity progression of matrix bundle1 at iteration 0, 1K and 10K, for which sparsity reduces very rapidly (99.3%, 63%, 36.1%). Change in sparsity can have direct impact on the compression ratio for different compression techniques as data patterns can dynamically change during runtime. Such changes affect number of CIAOs, cache hits/misses and used bandwidth.



(a) Bcsstk36 (b) Bundle1

Fig. 3. Cumulative composite distribution over different cutoff value (vertical red line) , composite size=256, SR=0.01

For simplicity, we are not interested in getting close sparsity approximation for every sampled composite but instead we classify composites into two broad categories where composites with patterns (in this case sparse composites) are compressed for memory and bandwidth savings. The cutoff value to decide such classification can be predetermined or calculated during runtime. Figure 3(a) and 3(b) show cumulative percentage of composites at different cutoff values (vertical red lines) for intermediate matrices in Figures 2(c) and 2(e) respectively. The cut off line shown at 95 percentile and 80 percentile can be moved on the x-axis in Figure 3(a) and 3(b) to create different classifications based on certain thresholds. The height of the last entry before the cut off line represents the percentile of composites left untouched by the compression engine.

Here we applied static analysis as an example, however our eventual goal is to provide a dynamic system that is capable of guiding classification based on data patterns, composite sizes, compression ratios and architectural states. We leave further optimization of the discriminator for future work.

V. Experiments And Results

In this study, we report power consumption in Watts [W], energy consumption in Joules [J], and timing (performance) in seconds [s]. Data points in the graphs are the average records of multiple runs. We conduct our experiments on two multicore HPC nodes to show the usage of our ACDT framework: A large Intel based Xeon SMP (for Cholesky Decomposition Analysis) and a 16-core Intel Xeon E5-2670 with 64 GiB of main memory (for Matrix Vector Multiplication). The large SMP Intel node is an Intel Westmere-E7 8860 multiprocessor (32nm) and it contains 8 sockets with 10 cores/socket. The default core clock speed is 2.27GHz and turbo boost is supported. L2 cache (256k) is shared by all the cores in each socket and L3 cache (24MB) is shared by all the sockets. The memory size on the SMP node is 2TB, useful for scalability studies and in line with demands of future Exascale systems. To maximize performance, we use the maximum number of threads supported by the architecture (e.g. 80 threads for the SMP node) to run applications.

After an analysis of our introductory Matrix Vector Multiply example using our ACDT framework in Section V-A, we

TABLE II. UFL DATASET FEATURES

Name	row	$columns$	non-zeros	Description
bundle1	10,581	10,581	770,811	3D vision
bcsstk17	10,974	10,974	428,650	Pressure Vessel
bcsstk36	23,052	23,052	1,143,140	Shock Absorber
QY case9	14,454	14,454	147,972	Power Network
pdb1HYS	36,417	36,417	4,344,765	Protein
raefsky4	19,779	19,779	1,316,789	Structural
Trefethen	20,000	20,000	554,466	Combinatorial

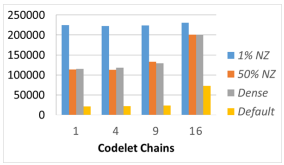
showcase a power-efficient, ACDT based, Cholesky Decomposition (Section V-B). We select several popular real-world datasets (e.g., structural problem, computer graphics/vision, etc.) from University of Florida Sparse Matrix Collection [23] to cover a wide spectrum of data pattern and degrees of sparsity. Their features are shown in Table II. All datasets in the table are real and symmetric.

A. MVM Analysis

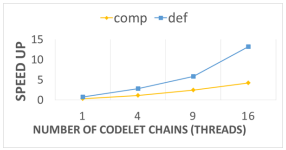
The matrix vector multiply (MVM) is a good didactic example to showcase Architected Composite Data Types (ACDT). It features a regular control and memory access pattern. Moreover, it is a kernel that is featured in many linear algebra operations. The serial implementation used as baseline is presented in algorithm 1. Starting from this version, we implemented a tiled version in which the matrix and the vectors are equally divided in tiles and padded if the division is not exact.

Each chain of SWARM codelets is associated to a thread and binds to it. Due to the usage of matrix **A** in the algorithm, each tile of matrix **A** was selected to be transformed to a balanced Compress/Decompress (*b*C/D) ACDT composite. In the current experiments, the *b*C/D composite can take advantage of two underlying compression algorithms. They have different tradeoffs between compression ratios and overheads depending on the composition of the workload. The first algorithm is the Burtscher and Ratanaworabhan FPC [5] described in Section II-B. It was selected for its floating point workload affinity and its high compression ratio. However, its high complexity results in a high overhead when no hardware support is provided. The second algorithm is the Run Length Encoding (RLE) phase of the X-Match algorithm [13]. RLE was selected because it is a simpler algorithm and it has a lower compression overhead. Nevertheless, the compression ratio of this approach is lower than the FPC algorithm. The *b*C/D ACDT composites are compressed at the beginning of their lifetime based on the compression ratio, and are decompressed before being used inside a codelet chain.

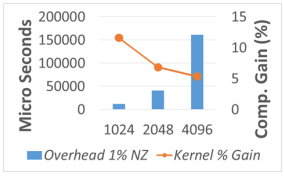
Both implementations were highly optimized for the architectural testbed. In the case of FPC, the predictor tables were reduced in size to ensure that they fit in the cache and vector instructions were used for their main operations. In the case of RLE, the main loops were enhanced with pre-fetching instructions. Using these algorithms, we ran Matrix Vector Multiply with *b*C/D ACDT composites. The following analysis showcases the compression overhead of both algorithms, the number of last level cache misses, the gain on the computational kernel and the general speedup of the application. Each test case was run with synthetic workloads to ensure a uniform distribution of non zero elements on each composite.



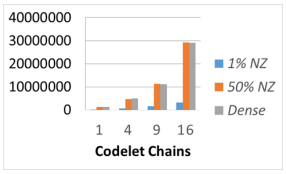
(a) Performance 4096



(b) Speedup 4096



(c) Overhead with 1% NZ (left axis) and Performance Gain (right axis)



(d) LLC Misses for 4096

Fig. 4. FPC: Collected performance data for the MVM algorithm. In the Figures, comp stands for compressed and def stands for the baseline implementation

1) The FPC Analysis: Figure 4(a) shows the relationship between overall performance and sparsity for the FPC-based *b*C/D ACDT. The y-axis represents application's time to completion in microseconds with different sparsity levels. The bars are grouped by the number of codelet chains. The bars are 1% non-zero values (the blue bar), 50% sparsity (the red bar), and dense (the gray bar)[1]. Finally, the default bar (the yellow bar) represents the most optimized version of the application in the SWARM runtime that does not utilize compression. As depicted in the Figure, FPC increases in complexity when dealing with compressible patterns as shown by the 1% bar. The 50% sparsity bar shows that it has negligible improvements over the non-compression scheme (i.e. dense).

Due to its compression overhead, FPC does not scale as well as the best optimized Matrix Vector running in SWARM in the current testbed. The speedup curves in Figure 4(b) shows a decrease to a quarter in speedup when compared to the optimized version.

The overhead of this algorithm is shown in Figure 4(c)'s left axis and appears to be substantial, showing a maximum value of 160K for composite sizes of 4k by 4k. However, there is a computational kernel gain of bringing data into the cache using compression (shown in Figure 4(c)'s right axis). Due to its disproportionate high overhead, the benefits of this software-based compression method are offset by its overhead. However, if we separate the overhead from the application, the data tells a different story. When considering the computational kernel runtime, there is a reduction of overall runtime from 11% to 6% when increasing the composite size from 1024 by 1024 to 4096 by 4096. The reduction of performance gain when increasing the composite size is due to cache eviction effects since the composite sizes rapidly overcome the cache sizes. Moreover, there is an aspect of thread interference that is aggravated by this increase. Nevertheless, this trend shows performance gains in the computational kernel. Compression helps in this aspect since it effectively decreases the cache footprint and allows bigger size composites to amortize the cost of the compression and decompression algorithms. This experimental sweep also showcases that the cache behavior

[1] with no compression but with a pre-fetching scheme to bring data to the cache

is an important aspect that must be considered when selecting the right composite type with the discriminator. Otherwise, the performance gain will be lost due to detrimental side effects.

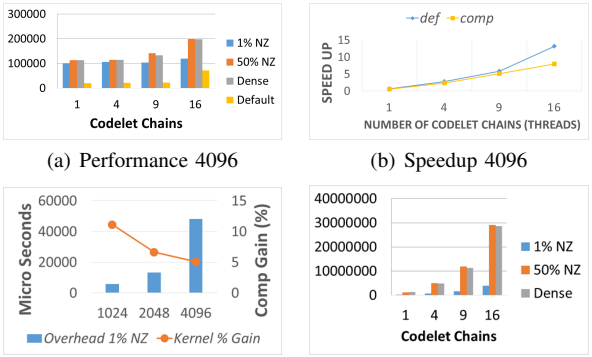
Finally, the FPC algorithm shows promise when dealing with the last level cache misses. Shown in Figure 4(d), last level cache misses for FPC are on average 13 times lower than the dense version. This trend shows that in this case, compression helps to bring data sets into the core's caches with fewer memory transactions from main storage.

Such trends help us make a case for reducing the overhead of *b*C/D type's algorithms to better utilize the memory bandwidth and to improve the orchestration of resources in the computer system.

2) The RLE Analysis: The RLE algorithm results present a case in which the overhead of the underlying algorithm is lower and how a lower compression overhead affects the performance. The runtime shown in Figure 5(a) presents the RLE overhead. When compared with the FPC algorithm numbers, there is around 2x to 3x reduction in overhead. This reduction translates into an overall improvement in scalability (around twice) over its FPC counterpart, as shown in Figure 5(b). Meanwhile, RLE maintains similar computational kernel gains (shown in Figure 5(c)'s right axis) and reduction of level 3 cache misses (shown in Figure 5(d)). Although this is not enough to overcome the gap in performance between the compressed and default approaches, the speedup curves and runtime numbers show that this gap is greatly reduced (around half).

The reason behind the similar behavior between the compression algorithms is that the synthetic data sets present very simple and similar compressible pattern to both algorithms. This means that the extra features of FPC do not have an effect on the compression ratio of the final composite for this set of experiments.

As the data has shown, having hardware support for these operations can greatly increase performance because of the computational kernels improvements due to a reduced number of main memory transactions. However, there are other key aspects that need to be considered in order to exploit algorithmic and architectural optimization opportunities. One of these aspects is the lifetime of a composite. Matrix Vector multiply is a simple example to showcase the abilities of the framework. However, due to the limited lifetime of its composites, the advantages that could be harnessed from the ACDT framework are limited. Thus, hardware support alone is not enough and adaptive runtime support is needed to select the best possible composite candidate based on aspects such as composite lifetime, cache sizes, access latency, etc. For this reason, the ACDT's discriminator in Sec. IV-D would need to periodically sample composites about their feature vectors in order to select an adequate compression/decompression action. Moreover, since the *b*C/D ACDT version is transparent to the application, benefits from operating directly on compressed data cannot be exploited. If the application can be made aware of these opportunities, chances are that performance and power efficiencies will greatly increase. We will showcase these benefits using the *u*C/D ACDTs for sparse data workload as presented next.



(a) Performance 4096

(b) Speedup 4096

(c) Overhead with 1% NZ (left axis) and Performance Gain (right axis)

(d) LLC Misses 4096

Fig. 5. RLE: Collected performance data for MVM algorithm. In the Figures, comp stands for compressed and def stands for the baseline implementation

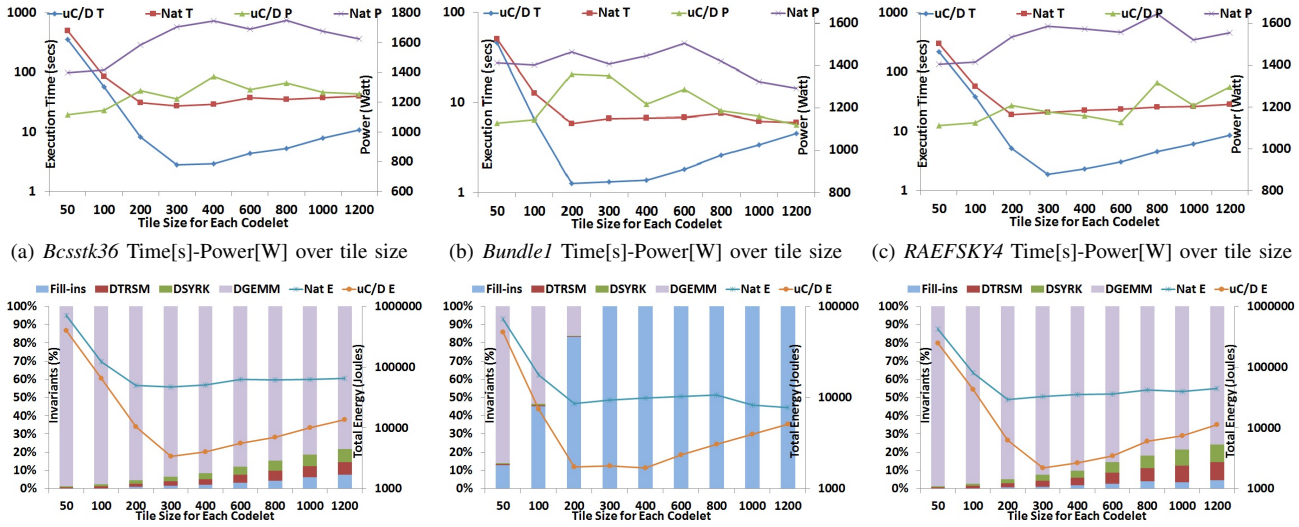
B. Cholesky Decomposition Analysis

To understand how our ACDT framework performs on more sophisticated algorithms, we decided to evaluate performance and power of a SWARM Cholesky decomposition – *"nativeCholesky"* for short. The existing reference implementation (to be found at [8] website) is highly hand optimized for dense real, positive-definite matrices. Our challenge is to apply the code to sparse matrices with no or slight modifications and derive improvements in performance and power – as an added bonus, we get memory capacity savings as well. We will call this modified code *"ucdCholesky"* for short. As the Section V-A showcased, performance improvements in SW without HW support are difficult to attain for the *b*C/D case (see Sec. IV-A1). Some "data transformation awareness" in the application is desirable to improve the performance and power efficiency outcome. This insight led us to develop an ucdCholesky for the *u*C/D case with only slight modifications to the original optimized code.

For the *u*C/D compress/decompress engine we chose a simple Block Sparse Row (BSR) [18] representation for the sparse matrix square tiles served to the math kernels. BSR has the advantage that the original interfaces to the underlying math kernels (potrf, rrsm, syrk, gemm – in BLAS and LAPACK nomenclature) do not need to be modified. The tiles in these experiments conform to the composites of our ACDT framework. BSR is the unbalanced *u*C/D approach in our framework.

BSR is a static matrix representation and as such, is not amenable to dynamic matrix changes without recompressing the matrix after each algorithmic step; a process that would be prohibitively time consuming. For this purpose we extended BSR to *dynamic BSR* (dBSR). If a tile is not found in the original BSR structure, a secondary key/value store structure addressed by a chain-hashing function is queried. This secondary store holds the matrix fill-ins that are generated as the algorithm progresses.

BSR's decompressing time complexity is $O(\sqrt{T})$ where $T = n/N$ is the total number of square tiles after matrix partitioning. n and N are the number of elements in the matrix and tile, respectively. For the nominal case, this complexity tends to be much smaller and is a function of the non-zero-element (NZE) distribution in the matrix. dBSR's decoding



(a) *Bcsstk36* Time[s]-Power[W] over tile size (b) *Bundle1* Time[s]-Power[W] over tile size (c) *RAEFSKY4* Time[s]-Power[W] over tile size

(d) *Bcsstk36* CIAO[%]-Energy[J] over tile size (e) *Bundle1* CIAO[%]-Energy[J] over tile size (f) *RAEFSKY4* CIAO[%]-Energy[J] over tile size

Fig. 6. Cholesky factorization Native (Nat) vs. unbalanced Compression/Decompression (uC/D)

time complexity is $O(\sqrt{T}+T)$ as we consider probing the secondary key/value store. Yet for the nominal case, avoiding most bucket collisions, we end up again with $O(\sqrt{T})$. dBSR's decompressing time complexity in ucdCholesky contrasts with the overhead that the original, optimized SWARM Cholesky decomposition (nativeCholesky) expends on tile addressing in the order of $O(1)$. In practice, our $Overhead_{dec}$ in formula 6 in Sec. IV-B tends to be 3-5x the addressing overhead of the original implementation. According to formula 6 it is clear that we can only expect improvements if we have latency access savings; in other words, without HW modifications, if we substantially reduce the number of memory accesses.

The latency aspect is tilted in favor of ucdCholesky by extensive use of Composite Invariant Algebraic Operators (CIAO) (Sec. IV-A3): Probing dBSR inherently signals Zero tiles (*"SigZero"*) during decompression[2]. We use this signal to enable optimizations at the decompression engine as well as at the application logic: Upon SigZero, the decompression engine delivers to the application either a Read-Zero tile or a Write-Zero tile. Read-Zero tile requests are aliased to a unique reserved memory address with the consequence that the tile values will be served with high probability from the cache, reducing memory access traffic and hereby improving performance and power. Write-Zero tiles are served from a memory pool and registered in dBSR's key/value store.

In addition, CIAO benefits the application in two ways: 1) Tile decompression is evaluated in a lazy fashion, ordered by the algebraic importance of the operator's constituent right-hand tiles. A SigZero of a dominant tile obliviates decompression of the dominated tiles. 2) SigZero can lead to an arithmetic intensity reduced operator, in most cases either a tautology or no operator at all. As with the decompression engine optimizations, these two application optimizations have a positive impact on performance and power as well.

As an example, Figures 6(a),6(d) depict performance (T), power (P) and energy (E) over varying tile sweeps of Boeing's Bcsstk36 matrix running nativeCholesky (Nat) or ucdCholesky (*u*C/D). Recall from discussion in Sec. IV-D, (Figure 2) that this matrix maintains a high level of sparsity during program evolution. We notice from the left-y-axis half-logarithmic Figure 6(a) that the performance of ucdCholesky over native-Cholesky is an order of magnitude better at the stationary point located at tile size $\sim 350^2$. Moving farther left from the minimum, the ratio of computation vs. memory movement becomes unfavorable, as smaller tiles yield less work and addressing and decoding overhead starts to dominate. Since the $Overhead_{dec} = O(\sqrt{T}) > O(1)$ we see ucdCholesky rapidly loosing its advantage. From the stationary point on to the right, overhead due to addressing or decompression starts to loose importance in comparison to computation. Moving farther right from the minimum, we notice ucdCholesky gradually ceding benefits: This behavior is explained by Figure 6(d). It shows left-y-axis accumulated CIAOs per tile size run in % for Cholesky's potrf, trsm, syrk and gemm math kernels. As tile sizes increase, invariably the number of SigZero signals decreases as a function of the probability mass function (pmf) of NZE over the matrix's topology. For ucdCholesky, the most prominent CIAO contributor is related to gemm.

The average power consumption per Cholesky run over tile size depicted in Figure 6(a) right-y-axis corroborates the performance findings. The power consumption improvement in general is $>\sim 20\%$ if we compare ucdCholesky vs. nativeCholesky. In fact, at tile size $\sim 350^2$ we see an uptick in power consumption due to increased arithmetic intensity. Even at the performance sweetspot we still experience 22% power improvement. The extensive use of CIAO reduces power consumption without sacrificing performance by not wasting power on memory movements that do not change the algorithmic outcome. Substantial improvements in performance and power consumption reflect positively on energy use as well as seen on the half-logarithmic right-y-axis Joules axis in Figure 6(d). A $> 10x$ improvement translates into savings of 90kJ.

Figures 6(b), 6(e), 6(c) and 6(f) all showcase similar trends, albeit with different personalities. Table II indicates that we ran more test cases which confirm our results. In terms of

[2]Diagonal 1,-1 tiles that enable CIAO could be encoded into dBSR as well but is out of scope for this paper

performance, ucdCholesky always wins by varying degrees of margin – recall that the performance axis is logarithmic. Power consumption is more erratic. As the addressing/decompression becomes less relevant, nativeCholesky and ucdCholesky curves approach each other. Nevertheless, even selecting the best tile sizes for nativeCholesky, ucdCholesky still wins. Overall, this translates into one order of magnitude ($> 10x$) in energy savings measured in kJ.

To check how ucdCholesky would fare with matrices that loose their sparsity quickly during program evolution – 36% sparsity at the end – we exercised ucdCholesky with the Lourakis Bundle1 matrix discussed in Sec. IV-D (see Figures 2(d), 2(e) and 2(f)). As it turns out, even in this case ucdCholseky wins. In Figure 6(b) we see that performance gains have melted to single digits but still constitute respectable several seconds savings, despite the fact that CIAOs disappear quickly and fill-ins dominate (see Figure 6(e)). The power gains are tight at the sweet spot but are still discernible. Overall, energy savings are still 8kJ.

In summary, in this section we have shown how the unbalanced Compress/Decompress (*u*C/D) approach under the ACDT framework can yield significant improvements in performance, power and energy for an important set of sparse matrices. Furthermore, these composites can exhibit significant changes in their attributes – e.g., change in sparsity of Lourakis Bundle1. A discriminator at runtime would be able to detect these changes, update the feature vector and discern a new type that would assign modified transformation rules to the composite.

VI. Conclusions and Future Work

We showed that our ACDT framework can improve efficiency of a massively multithreaded adaptive runtime system by managing composites in different compressed formats. With our *b*C/D approach, we showed better reuse of memory within different level of caches by reducing data movement. Similarly, with our *u*C/D, we showed improvement in power and performance by an order of magnitude utilizing ACDT operators and taking advantage of ACDT invariants. Next steps will examine ACDT on other adaptive runtime systems such as OCR and expand ACDT capabilities to support resiliency.

Acknowledgment

This research was supported in part by DOE ASCR XStack program under Awards DE-SC0008716, DE-SC0008717.

References

[1] S. Ashby et al. The Opportunities and Challenges of Exascale Computing. Technical report, US Department of Energy Office of Science, 2010. Summary Report of the Advanced Scientific Computing Advisory Committee (ASCAC).

[2] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. M. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. V. D. Vorst. Templates for the solution of linear systems: Building blocks for iterative methods. In *SIAM*, 1994.

[3] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, and K. Yelick. Exascale computing study: Technology challenges in achieving exascale systems peter kogge, editor & study lead. Technical report, DARPA, 2008.

[4] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. *SIGPLAN Not.*, 30(8):207–216, Aug. 1995.

[5] M. Burtscher and P. Ratanaworabhan. Fpc: A high-speed compressor for double-precision floating-point data. *Computers, IEEE Transactions on*, 58(1):18–31, 2009.

[6] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 519–538, New York, NY, USA, 2005. ACM.

[7] J. Demmel, M. Hoemmen, M. Mohiyuddin, and K. Yelick. Avoiding communication in sparse matrix calculations. Apr. 2008.

[8] ETI. Swarm (swift adaptive runtime machine) http://www.etinternational.com/index.php/products/swarmbeta/, 2012.

[9] E. Georganas, J. González-Domínguez, E. Solomonik, Y. Zheng, J. Touriño, and K. Yelick. Communication avoiding and overlapping for numerical linear algebra. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 100:1–100:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[10] R. Gonzalez and M. Horowitz. Energy dissipation in general purpose microprocessors. *Solid-State Circuits, IEEE Journal of*, 31(9):1277–1284, Sep 1996.

[11] C. Jung, S. Rus, B. P. Railing, N. Clark, and S. Pande. Brainy: Effective selection of data structures. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 86–97, New York, NY, USA, 2011. ACM.

[12] G. Kestor, R. Gioiosa, D. J. Kerbyson, and A. Hoisie. Quantifying the energy cost of data movement in scientific applications. In *Workload Characterization (IISWC), 2013 IEEE International Symposium on*, 2013.

[13] M. Kjelso, M. Gooch, and S. Jones. Design and performance of a main memory hardware data compressor. In *EUROMICRO 96. Beyond 2000: Hardware and Software Design Strategies., Proceedings of the 22nd EUROMICRO Conference*, pages 423–430, 1996.

[14] K. Kourtis, V. Karakasis, G. Goumas, and N. Koziris. Csx: An extended compression format for spmv on shared memory systems. *SIGPLAN Not.*, 46(8):247–256, Feb. 2011.

[15] J.-S. Lee, W.-K. Hong, and S.-D. Kim. Design and evaluation of a selective compressed memory system. In *Computer Design, 1999. (ICCD '99) International Conference on*, pages 184–191, 1999.

[16] S. Leung and J. Zahorjan. *Optimizing data locality by array restructuring*. Citeseer, 1995.

[17] D. Molka, D. Hackenberg, R. Schone, and M. Muller. Characterizing the energy consumption of data transfers and arithmetic operations on x86 64 processors. In *Green Computing Conference, 2010 International*, pages 123–133, 2010.

[18] Y. Saad. Sparskit: a basic tool kit for sparse matrix computations - version 2. Technical Report. Computer Science Department. University of Minnesota, Minneapolis, MN, 55455, 1994.

[19] V. Sarkar, B. Chapman, W. Gropp, and R. Lethin. Building an Open Community Runtime (OCR) framework for Exascale Systems, nov 2012. Supercomputing 2012 Birds Of A Feather session.

[20] J. Shalf, S. Dosanjh, and J. Morrison. Exascale Computing Technology Challenges. In *Proceedings of the 9th International Conference on High Performance Computing for Computational Science*, VECPAR'10, pages 1–25, Berlin, Heidelberg, 2011. Springer-Verlag.

[21] M. A. Suleman, O. Mutlu, J. A. Joao, Khubaib, and Y. N. Patt. Data marshaling for multi-core architectures. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 441–450, New York, NY, USA, 2010. ACM.

[22] K. Theobald. *Earth: An Efficient Architecture for Running Threads*. McGill University, 1999.

[23] Y. H. Tim Davis. The university of florida sparse matrix collection. http://www.cise.ufl.edu/research/sparse/matrices/, 1994.

[24] V. Tiwari, S. Malik, and A. Wolfe. Power analysis of embedded software: A first step towards software power minimization. In *Proceedings of the 1994 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '94, pages 384–390, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[25] H.-W. Tseng and D. M. Tullsen. Data-triggered threads: Eliminating redundant computation. In *HPCA'11*, pages 181–192, 2011.