Characterizing Application Execution using the Open Community Runtime

Zoran Budimlić
Rice University
Houston, Texas, USA
zoran@rice.edu

Vincent Cavé
Intel
Hillsboro, Oregon, USA
vincent.cave@intel.com

Sanjay Chatterjee
Intel
Hillsboro, Oregon, USA
sanjay.chatterjee@intel.com

Romain Cledat
Intel
Hillsboro, Oregon, USA
romain.e.cledat@intel.com

Vivek Sarkar
Rice University
Houston, Texas, USA
vsarkar@rice.edu

Bala Seshasayee
Intel
Hillsboro, Oregon, USA
bala.seshasayee@intel.com

Rishi Surendran
Rice University
Houston, Texas, USA
rishi.surendran@rice.edu

Nick Vrvilo
Rice University
Houston, Texas, USA
nick.vrvilo@rice.edu

ABSTRACT

Exascale and extreme-scale systems impose fundamental new requirements on software to target platforms with severe energy, data movement and resiliency constraints within and across nodes, and large degrees of homogeneous and heterogeneous parallelism and locality within a node. These challenges have led to the exploration of a diverse range of many-core processor architectures and memory hierarchies for future systems, that differ quite dramatically from current systems. As a result, there is still a lack of consensus as to what the main tenets should be for the execution model and low-level runtime system in future architectures. The Open Community Runtime (OCR) was created to engage the broader community of software and hardware researchers in identifying these underlying principles. While there is broad support for including dynamic task parallelism as one of the pillars of future execution models, there is currently little agreement on what else should be included. OCR proposes an approach to complete this picture by adding events and relocatable data-blocks as two additional pillars to build on, and shows how the three concepts (tasks, events, data blocks) can be combined in very general ways to support a wide range of higher-level programming constructs.

In this paper, we focus on the use of OCR for application characterization. Despite the fact that the development of OCR is still at an early stage, we are fortunate that a large number of applications have already been implemented using the OCR APIs. We study the behavior of OCR implementations of these applications to gain insights into task durations, data block sizes, and access patterns. These insights can provide feedback to both application developers (on how to redesign algorithms for OCR-like execution models) and to hardware designers (to optimize for common characteristics of tasks, events and data blocks).

RESPA '15 Austin, Texas USA

1. INTRODUCTION

Exascale and extreme-scale systems impose fundamental new requirements on software to target platforms with severe energy, data movement and resiliency constraints within and across nodes, and large degrees of homogeneous and heterogeneous parallelism and locality within a node. It is widely agreed that major innovations in runtime systems will be necessary to address these challenges of extreme-scale computing. Further, these innovations need to be coupled so that light-weight computations and data transfers can be dynamically orchestrated across processor cores, customized accelerators, memory hierarchies, and network interfaces with unified approaches to parallelism, energy and resilience. The runtime framework must also help bridge the wide semantic gap between high-level programming models and low-level exascale hardware.

The Open Community Runtime (OCR) [7] was created to engage the broader community of application, compiler, runtime, and hardware experts in designing this critical component of the future exascale software stack so as to best address these challenges and requirements. The OCR framework allows for multiple implementations of common runtime APIs, with different trade-offs in different dimensions of resource usage. OCR is not intended to be a standardization effort, but instead an open substrate that helps accelerate innovations in the exascale software stack. As an example, the creation of the OCR specification enabled researchers at the University of Vienna to create a new implementation for OCR that is completely independent of the OCR reference implementation.

There are many differences between the proposed OCR approach and existing solutions. Many current runtime systems have either been optimized for dynamic parallelism that is oblivious of locality (e.g., OpenMP, Intel Thread Building Blocks, Cilk) or for locality in the absence of dynamic parallelism (e.g., MPI, shmem, UPC). Further, MPI and PGAS programming models only express two levels of locality — local and remote — which are typically accessed through a SPMD parallel execution model without dynamic task parallelism. HPCS languages such as Chapel and X10 mitigate this limitation by allowing the programmer to express task parallelism in conjunction with "locales" and "places", but their locality model is also restricted to two levels. In contrast, OCR supports unbounded amounts of dynamic parallelism with the potential for hierarchical locality control that can includes support for heterogeneous accelerators, while also allowing for tight integration with communication runtimes (e.g., MPI, GASNet, UCX) and runtime co-optimization of performance, energy, and resilience. The most basic OCR primitives (summarized in Section 2) include relocatable data-blocks and non-preemptive event-driven tasks, and result in a fundamentally different execution model from past runtimes. In addition to running on current hardware platforms, a functional simulator has also been developed to map OCR on to a representative future extreme-scale computer architecture [1].

Despite the fact that the development of OCR is still at an early stage, we are fortunate that a large number of applications have already been implemented using the OCR APIs, including CoMD, HPGMG, LULESH, miniAMR, SNAP, Tempest, RSBench, and XSBench, as well as a number of kernels (Stencil1D, Cholesky, FFT, Fibonacci, Global Sum, SAR, Smith Waterman, Synch_p2p, triangle). Given the fundamental differences between OCR APIs and other parallel runtime APIs, the focus of this paper is on understanding the characteristics of current applications implemented in OCR. These characterizations can provide insights for both hardware and software designers. Hardware designers can learn more about the typical ranges for task execution times and data-block sizes. Software designers can explore opportunities for future work on compile-time and run-time code/data transformations (e.g., splitting of data-blocks or fusing of tasks) to improve the overall execution characteristics.

The rest of the paper is organized as follows. Section 5 goes over some related work. Section 2 provides a brief summary of OCR, while referring the reader to the OCR specification for details [7]. Section 3 describes the experimental methodology used in our work to study application characteristics. Section 4 contains the results obtained for application characteristics. Section 5 summarizes related work, and Section 6 contains our conclusions.

2. OCR SUMMARY

The OCR effort seeks to develop, formalize and implement a task-based programming model for future exascale systems. In this section, we focus on briefly describing the philosophy behind the OCR programming model and its embodiment in the current set of APIs and reference implementation.

2.1 The OCR programming model

Given that, in future machines, the cost of data movement will vastly overshadow the cost of computation, OCR's programming model is similar to a data-flow model in the sense that data is explicit and "flows" between *tasks*. It is not, however, a traditional data-flow model as, unlike in pure data-flow, the data graph is built dynamically.

2.1.1 OCR principles

OCR has several goals aimed at ensuring that it is a valid model for future exascale systems: allow the expression of the maximum amount of parallelism, insulate the programmer from the specificities of the hardware, and allow the programmer to fully and accurately represent his knowledge of the application.

Parallelism.

In future exascale machines, given the trend towards more and more computing resources, whether this is in the form of a GPU or chips like the Xeon Phi, a large amount of parallelism will be required to keep the machine busy. It is therefore important to provide a way **a)** for a programmer to express the parallelism in his application and **b)** to have light-weight tasks since increasing parallelism may come at the expense of more traditional coarse grained division of work.

Separation of concerns.

Traditionally, programmers seeking to eek out the last bit of performance out of a particular machine will fine tune their code to that particular architecture taking into account cache sizes, instruction latency, etc. Given the constraints on exascale machines, particularly in terms of power, future large machines will be far more unpredictable and therefore make it harder for a programmer to tweak his application to a particular machine. Furthermore, given that the effort to tweak an application to a particular machine increases with the complexity of the machine, continuously adapting an application to each new generation of machine will become unsustainable.

An automated way of tuning the application to the machine it is being run is therefore required. Traditionally, compiler-driven static scheduling has been used and is an efficient technique provided the compiler has an accurate view of the hardware. As previously mentioned, on future hardware, this may become more challenging. OCR therefore takes the approach that scheduling decisions are best done dynamically by a runtime that is aware of the exact execution conditions (i.e. the runtime is introspecting its own execution). In OCR, the programmer is therefore mostly insulated from the exact details of the hardware. The exact "platform" exposed to the user is still an active area of research but OCR currently takes the approach that the user should not be able to directly reason about a machine.

Transfer of information.

Naturally, if more decisions are left to the runtime, the programmer needs to be able to express as much of his knowledge about an application as possible (access patterns, affinity between tasks and data, etc.). The OCR API therefore strives **a)** to allow the programmer to express all of his knowledge and **b)** to avoid having the runtime make assumptions about the programmer's intent. In other words, we seek to provide an API that allows all information through but does not lose or assume any.

2.2 OCR concepts

Three concepts are central to the OCR programming model: **a)** tasks, which we call event-driven tasks or EDTs, **b)** data-blocks and **c)** events. Tasks are units of computation and operate on explicit input data in the form of data-blocks. Events are linked together (in a data-flow sense) through events. All OCR objects are uniquely addressable using a Globally Unique Identifier (GUID). An overview of these concepts and the relationships between them are shown in Figure 1.

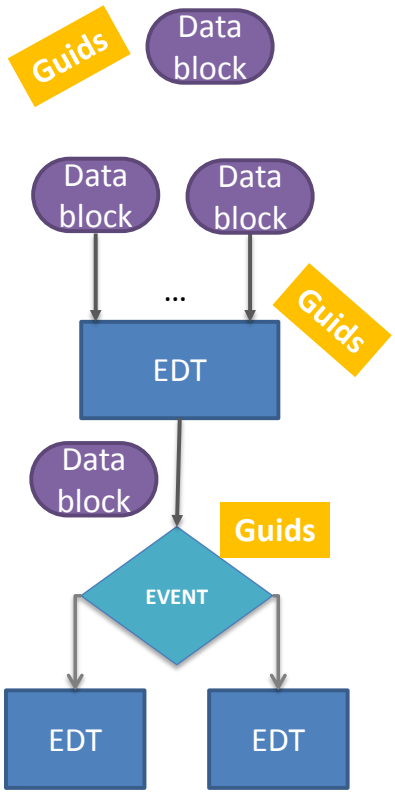

Figure 1: Basic Objects in OCR

OCR also introduces a number of secondary concepts that we do not describe in this paper. A detailed description of all these concepts is beyond the scope of this paper and we restrict ourselves to basic information on the central concepts; more information can be found in the OCR specification [7].

2.2.1 Event driven tasks

EDTs are OCR's abstractions for a unit of work. The name "event driven" comes from the fact that the start of an EDT's execution depends on the satisfaction of its dependences (events). A satisfied dependence may carry data (a data-block) with it. This data can be operated on by the EDT during its execution.

The lifetime of an EDT is as follows:

Creation Another EDT will create the EDT to execute. Upon creation, the programmer will supply the code to execute, the number of dependences that the EDT has as well as supply optional parameters.

Setup The dependences for the EDT will then be setup. An EDT can depend on an event (and will wait for that event to be satisfied prior to executing), a data-block (in effect, a pure data-dependence) as well as the completion of other EDTs.

Execution Once all dependences are defined and satisfied, the EDT will be scheduled by the runtime and execute *at some point* in the future. Note that OCR does not force EDTs to execute as soon as their dependences are satisfied; the satisfaction of the last dependence is only the lower-bound on the start of execution. During execution, an EDT may create other EDTs, data-blocks and events and satisfy events but may not wait on anything or synchronize with another EDT; in other words, once an EDT starts executing, it is guaranteed to complete execution irrespective of the actions of other EDTs.

Cleanup Once an EDT has executed, it will be cleaned up and its resources will be reclaimed. An EDT executes at most once.

2.2.2 Data-blocks

A data-block is OCR's abstraction for a chunk of data that is operated on by EDTs. A data-block is a contiguous chunk of memory which can be entirely accessed from a base address. The base address may, however, change over the execution of a program as the data-block migrates across the memory system (as directed by the runtime; this is transparent to the user). An EDT can only operate on data-blocks that it has a base address for and it can only get a base address if **a)** it creates the data-block or **b)** the data-block is passed in as a dependence to the EDT.

2.2.3 Events

Events are the synchronization mechanism in OCR. An event is initially in an "unsatisfied" state and transitions to a "satisfied" state when the programmer makes a special API call. Upon satisfaction, the user may optionally pass along a data-block which will be passed to the sink of the event (the "waiters").

2.3 Traleika Glacier — a strawman exascale architecture

Although OCR is currently usable on existing platforms and architectures, its design has been heavily influenced by the *Traleika Glacier* (TG) strawman architecture [1, 3] of a future exascale architecture. Note that OCR currently runs on the FSim functional simulator developed for this architecture. This section explains some of the reasoning that went behind OCR based on this architecture.

2.3.1 Overview of the TG architecture

The TG architecture is a hierarchial, non-coherent NUMA machine with heterogeneous cores. As seen in Figure 2, the system consists of a few power-efficient specialized cores (termed Execution Engines or XEs) controlled by an x86-based general purpose core in the smallest collection of cores called a block. Each core has access to its own scratchpad at very low latency in addition to a larger portion of memory common to the block. This pattern of hierarchical collection is extended all the way to the machine level. Being a shared memory system, any portion of any memory in the system (including scratchpads) is addressable by any core using a *canonical* address, albeit with different latencies depending

on the topological distance from the accessing core to the accessed portion of the memory. Additionally, they can also be addressed using a *self-referential* alias to its address that is core-relative, for ease of programmability.

2.3.2 TG influences

Several design choices of the TG architecture are reflected in the design of OCR, a few of which are outlined below.

GUIDs.

Data residing on the TG machine is bound to change address both due to the use of scratchpads (closer scratchpads are significantly smaller in size) and of its unique addressing scheme (the same location can have multiple addresses). Given the cost of virtual addresses (and therefore its absence in the TG architecture), memory addresses can no longer be used as identifiers for data. OCR therefore relies on GUIDs, an opaque identifier that will, at all times, uniquely identify data and other objects in OCR.

EDTs.

The XEs, being specialized low power cores designed primarily for number crunching, have limited/no interrupt handling support. This property maps well to EDTs, since EDTs, by definition, do not have any external synchronization and can run from start to finish without having to wait for anything. All the data that an EDT needs during its execution is made available and all dependences resolved prior to the EDT starting its execution.

Data-blocks.

As data movement costs are expected to be a significant portion of Exascale systems' overall power consumption, it is important that only the necessary portion of data needed for an EDT is made available to it. The data-block abstraction allows the programmer to partition the application's data into chunks needed by each EDT, so that extraneous data movement can be minimized.

Events.

The massive scale of Exascale applications, together with the task-granular nature of OCR implies that the management overheads can potentially quickly add up. The use of events, however, simplifies the management costs by imposing an implicit ordering of task execution. There are several variants of events (detailed in the OCR specification [7]) to address various types of flow-graphs, reducing the burden on the task scheduler.

3. METHODOLOGY

3.1 Applications

In this paper, we will study the characteristics of 4 applications that have been written using OCR's API. They are CoMD, Cholesky, SAR and StencilID.

CoMD is molecular dynamics application that calculates the force on each atom in a material resulting from other atoms within a cutoff distance. The motion of the atoms under these forces is simulated. The set of atoms being evaluated are spatially decomposed into grids that are first level data-blocks in the OCR application. The interatomic potential and related forces on each atom is calculated using OCR tasks. Since the range of neighboring atom elements span over a radial distance equal to the cutoff distance, the atom's position determines which data blocks will contain the elements that are its neighbors. The granularity of the EDT is determined by the number of atoms evaluated in that task.

Cholesky decomposition is a dense linear algebra application. The OCR version is an iterative tiled application where each tile on every iteration is executed by a unique task. An iteration starts with a sequential cholesky computation on the pivot tile. The pivot tile of an iteration is the highest tile on the diagonal. The subsequent iteration starts with the pivot tile being one step down the diagonal. The application halts when the pivot computation is done on the lowest element of the diagonal. In each iteration, the sequential cholesky on the pivot tile is followed by the trisolve computation of the pivot column tiles. These tasks release the update computations for the rest of the tile space in that iteration.

The Synthetic Aperture Radar (SAR) application is used to construct images of objects based on the motion of a radar antenna over a targeted region. A 3D array of voxels are maintained to represent the probability of an object in that space. The density of each voxel is initially 0. Waveform data sampled for each voxel adds the probability of the existence of the object at that position. The version used in this paper focuses on time-domain backprojection for image construction, which is a computationally intensive step but highly parallelizable using OCR tasks. The granularity of a task is determined by the number of image points constructed.

The Stencil1D application demonstrates the execution of a repeated pattern. The element space is a 1D decomposition in which each element reads the previous iteration data from its neighbor elements to compute the values of its current iteration. This repeated pattern is continued until convergence of values occur in the computation. Each element is represented by an independent task for each iteration and the data exchanges are synchronized using OCR events.

3.2 Runtime Instrumentation

To study the characteristics of an OCR application in a platform-neutral manner, we focus on metrics that are largely invariant across platforms, such as loads, stores and floating point instruction count. We gather these by **a)** modifying the compiler to emit instructions that track the above counts, and **b)** modifying the runtime to associate these counts to OCR objects such as EDTs and data-blocks, in order to obtain per-object, context-sensitive metrics.

We achieve the first step by using the LLVM compiler [6], along with a compiler pass that generates instructions to count floating point operations, loads and stores within each function. For the loads and stores, the address of the memory location and size of the operation are also recorded. While this technique was used on an x86 platform, the same can be applied to any other platform as well — in fact, these metrics can be obtained even from a simulator such as FSim [3] which tracks these internally on a per-instruction basis.

These raw metrics are then associated with OCR objects based on internally managed data structures within OCR. Specifically, each newly created data-block's start address and size are maintained, and similarly, each newly created

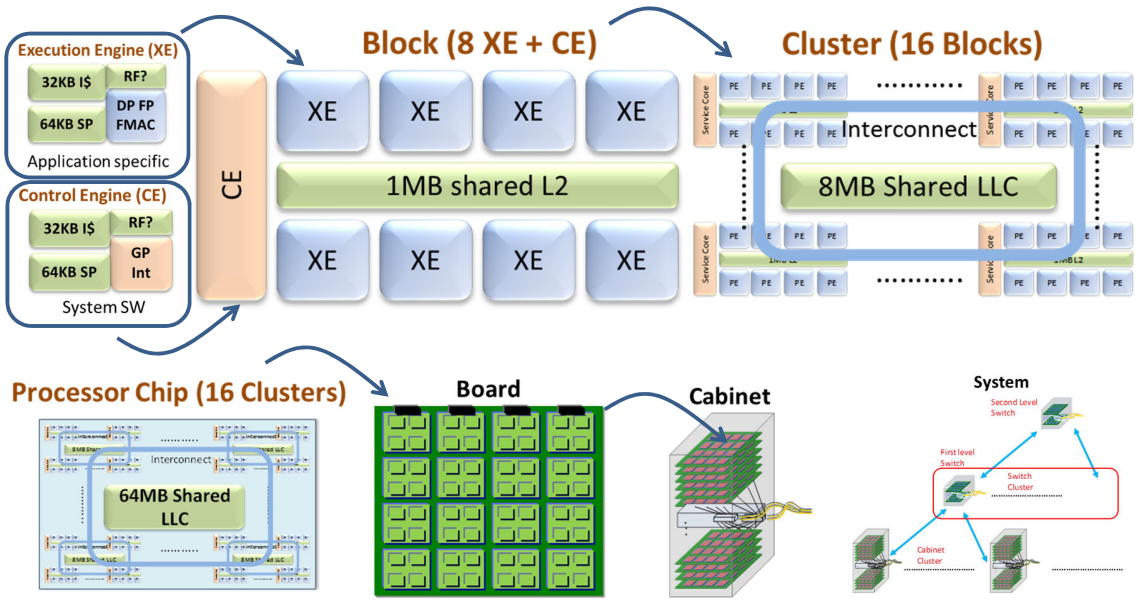

Figure 2: Traleika Glacier Architecture

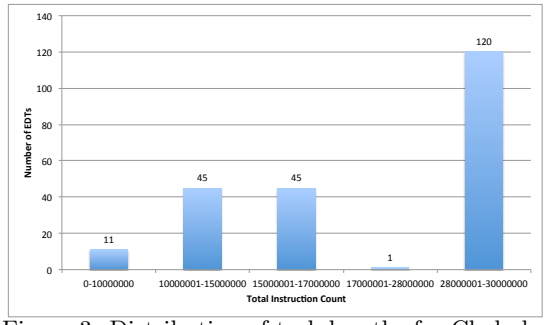

Figure 3: Distribution of task lengths for Cholesky

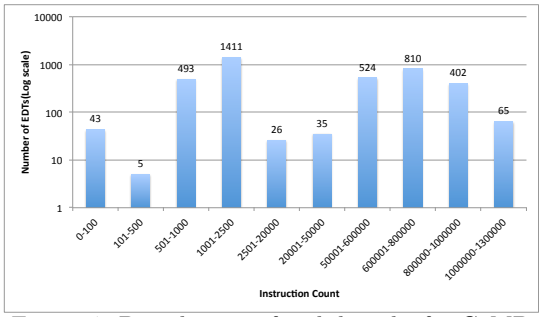

Figure 4: Distribution of task lengths for CoMD

task has its floating point instruction count zeroed. When the computation proceeds and any loads/stores are encountered, the target address is used to determine which datablock the load/store is associated with, as well as the corresponding EDT that is executing. A detailed accounting of each load/store and floating point operation is thus maintained by the runtime. At the end of each EDT's execution, these are printed to the console which is then collected for post-processing by scripts.

4. RESULTS

Figures 3–6 show the distribution of task lenghts (in the total number of instructions executed) for the four applications we have evaluated. This kind of information can be used by the runtime to make informed scheduling decisions, by picking more powerful (and more reliable) cores for the longer running tasks, for example. We note that while some applications (Cholesky and Stencil1D, for example) have only a handful of tasks of different lenghts, others (CoMD) have tasks of many varying lenghts. Discovering this information, possibly with programmer's assistance, would be of very high importance for the runtime.

Figures 7–10 show the distribution of data-block sizes for our four applications. This information can be used by the

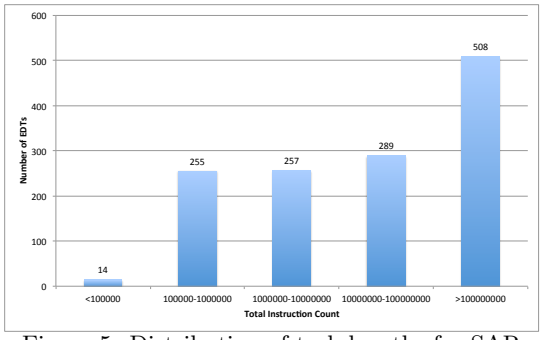

Figure 5: Distribution of task lengths for SAR

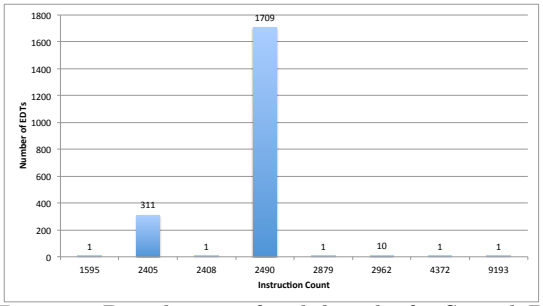

Figure 6: Distribution of task lengths for Stencil1D

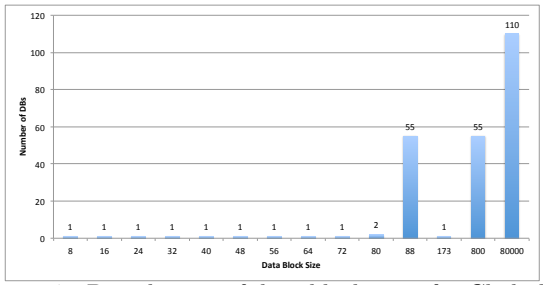

Figure 7: Distribution of data-block sizes for Cholesky

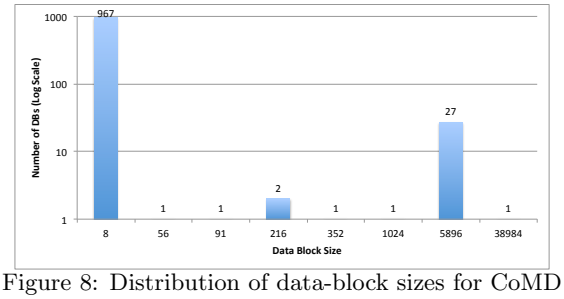

Figure 8: Distribution of data-block sizes for CoMD

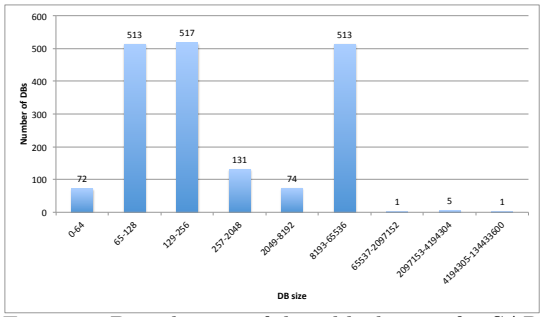

Figure 9: Distribution of data-block sizes for SAR

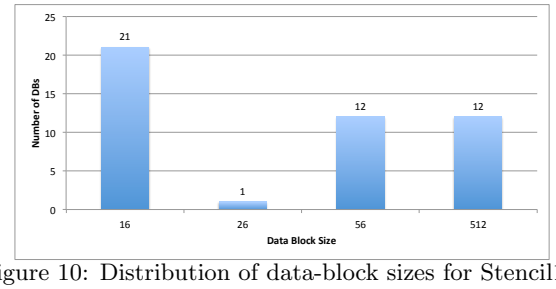

Figure 10: Distribution of data-block sizes for Stencil1D

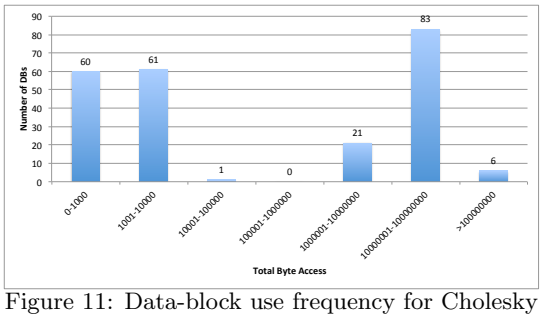

Figure 11: Data-block use frequency for Cholesky

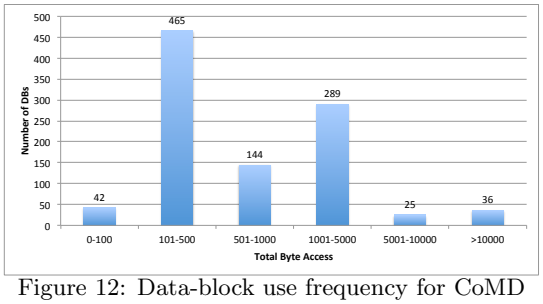

Figure 12: Data-block use frequency for CoMD

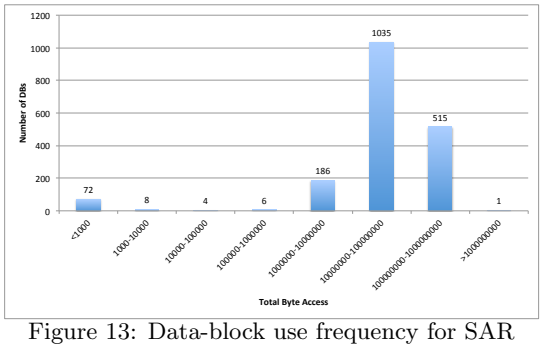

Figure 13: Data-block use frequency for SAR

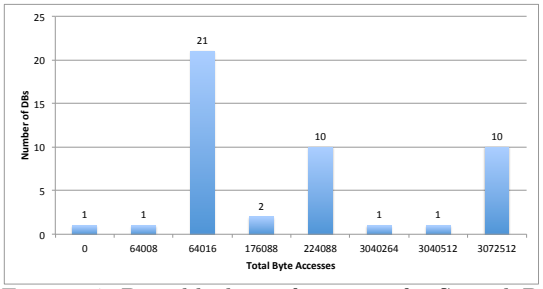

Figure 14: Data-block use frequency for Stencil1D

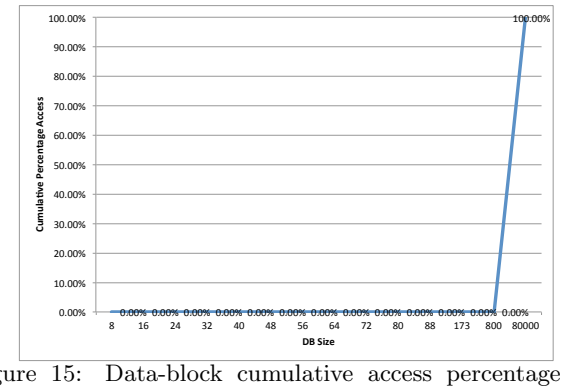

Figure 15: Data-block cumulative access percentage for Cholesky

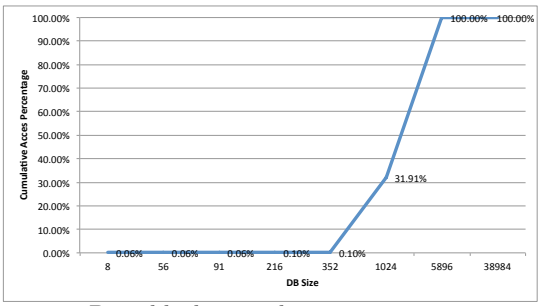

Figure 16: Data-block cumulative access percentage for CoMD

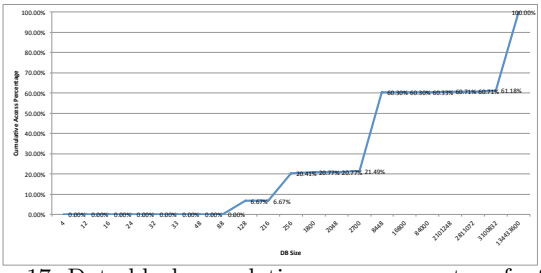
Figure 17: Data-block cumulative access percentage for SAR

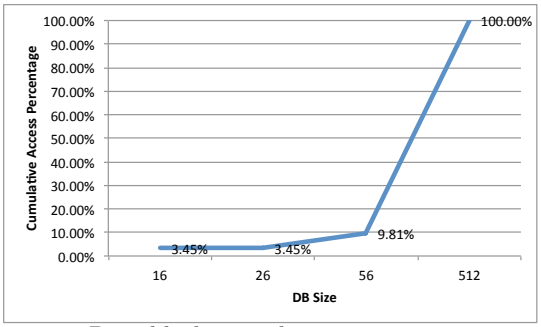

Figure 18: Data-block cumulative access percentage for Stencil1D

runtime to make memory allocation decisions.

Figures 7–10 show the data-block use frequency (in toal number of byte accesses) for the data-blocks in our applications. This information can be used by the runtime to make allocation decisions by placing frequently accessed data-blocks closer to the cores (for example, in block memory, or even in the individual XE's scratchpad on the TG proposed machine).

Figures 15–18 show the percentage of the total number of data accesses to the data-blocks smaller than the size shown on the X-axis. CoMD and Stencil1D applications have mostly small data-blocks, and could potentially benefit significantly from memory hierarchy optimizations, such as allocating the data-blocks in the parts of the memory hierarchy closer to the cores, with the faster access times but smaller memory sizes.

The SAR and Cholesky applications use larger data-block sizes. Such data-blocks would probably not fit into the lower levels of the memory hierarchy for most of the future systems being considered. One possible solution would be to split up the data-blocks, either statically using a compiler, or dynamically at runtime. Another would be to rely on the programmer or compiler to provide parameterized tile sizes that can be fine-tuned by the runtime.

While the measurements we obtained using the code instrumentation are very accurate, they also incur a very large overhead (close to two orders of magnitude on some of the benchmarks) because every memory read and write is instrumented. In order to make such measurements feasible for use during the application execution, the runtime will have to employ several approximations. For example, instead of counting all the memory accesses to the data-block by an EDT, the compiler can peel the first iteration of the loop and instrument only the first memory access, then multiply the access counter with the loop size. Also, the runtime can run the application for a while with a more detailed instrumentation, then after making decisions about the scheduling of certain types of EDTs and allocation of the certain types of data-blocks, it can run a less-instrumented version of the code, or even turn off the instrumentation completely. The instrumentation can be also aided by user annotations on the EDTs and data-blocks.

Another interesting challenge for the runtime will be to strike a correct balance between locality and parallelism. Placing a data-block high in the memory hierarchy will increase the access time but will allow many EDTs that are accessing the data-block and running on the cores underneath that node to do so in parallel. On the other hand, placing the data-block lower in the memory hierarchy will allow for a faster access, but by a smaller number of cores. The information collected from the instrumentation, such as the one presented here, on how many EDTs are accessing the given data-block and with what frequency, would be invaluable to the runtime in making a correct decision.

The runtime could also break up a large data-block high in the memory hierarchy in order to fit smaller pieces into the lower parts of the memory hierarchy, but the cost of breaking it up and reassembling would have to be offset by the benefits of doing so. Read-only data blocks can be disassembled and replicated in the lower levels of the hierarchy, closer to the cores. Naturally, this would only pay off if those blocks are accessed multiple times.

5. RELATED WORK

HPX [4] is another task-based runtime and programming model that is aimed at future exascale systems. HPX implements an active global address space, where globally-addressable objects can be tracked by a unique global ID as they are migrated throughout the system.

Charm++ [5] takes an object-oriented approach to building exascale software. Charm++ programs are decomposed into distributed objects, called *chars*, which are distributed throughout a system by the runtime. Messages are passed throughout the system via method calls on remote *char* objects.

Realm [8] is a fully asynchronous, event-based runtime for task-based computations. All runtime actions in Realm are non-blocking, building on a lightweight event mechanism for dependence management. Realm implements a concept of *physical regions* for shared global data, which provides type information for blocks of data that may be migrated to remote locations by the runtime. The additional type information provided for *physical regions* allows Realm to combine compute operations, such as reductions, with data movement within the runtime.

Carrington et al. at the San Diego Supercomputer Center have been working to develop and characterize the performance of several scientific compute kernels in OCR [2], including the CoMD kernel used in section 4. Their work thus far has focused on comparing the performance and scalability characteristics of applications written in OCR with corresponding implementations in more traditional HPC programming models, such as MPI+OpenMP; in contrast, this work focuses on identifying trends in OCR applications' data and task usage patterns that in turn help to identify potential optimizations.

6. CONCLUSION

In this paper, we reported on results obtained from an initial study to characterize application execution using OCR. Our early results provide insights into task durations, data block sizes, and access patterns resulting from current implementation approaches. These insights suggest the possibility of new approaches for application developers to redesign algorithms for OCR-like execution models, for programming systems to perform optimizations to split/fuse tasks and data blocks, and for hardware designers to optimize for common characteristics of tasks, events and data blocks. We believe that follow-on studies that study the intrinsic properties of OCR-enabled applications in more detail will lead to additional opportunities for software and hardware optimizations. Further, the fact that the OCR specification is publicly available, and is independent of any specific implementation, allows for these studies to be conducted in an implementation-independent manner.

Acknowledgments

The OCR release used for this paper includes contributions of ideas and code for both OCR and accompanying applications from participants at multiple sites, including Rice University, Intel Corporation, Univ. of Illinois at Urbana-Champaign, Univ. of California at San Diego, Reservoir Labs, EQware, Oregon State University, ET International, Univ. of Delaware, and Pacific Northwest National Laboratory. A broader community engagement during the application workshops has also included Sandia National Laboratory, Lawrence Livermore National Laboratory, Los Alamos National Laboratory, Lawrence Berkeley National Lab, Georgia Tech, and others.

This material is based in part upon work supported by the Department of Energy under Award Number DE-SC0008717.

7. REFERENCES

[1] N. P. Carter, A. Agrawal, S. Borkar, R. Cledat, H. David, J. Fryman, I. Ganev, R. A. Golliver, R. Knauerhase, R. Lethin, B. Meister, A. K. Mishra, W. R. Pinfold, J. Teller, J. Torrellas, G. Venkatesh, and J. Xu. Runnemede: An architecture for ubiquitous high-performance computing. In *High Performance Compute Architecture*, pages 198–209. IEEE, 2013.

[2] P. Cicotti, M. Shantharam, and L. Carrington. UCSD XStack. Presented at the X-Stack TG Retrospective Meeting, Aug. 2015.

[3] R. Clédat, J. Fryman, I. Ganev, S. Kaplan, R. Khan, A. Mishra, B. Seshasayee, G. Venkatesh, D. Dunning, and S. Borkar. Functional Simulator for Exascale System Research. In *2013 MODSIM workshop*, 2013.

[4] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey. Hpx: A task based programming model in a global address space. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, PGAS '14, pages 6:1–6:11, New York, NY, USA, 2014. ACM.

[5] L. V. Kale and S. Krishnan. Charm++: A portable concurrent object oriented system based on c++. In *Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '93, pages 91–108, New York, NY, USA, 1993. ACM.

[6] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, International Symposium on*, pages 75–86. IEEE, 2004.

[7] T. Mattson, R. Clédat, Z. Budimlić, V. Cavé, S. Chatterjee, B. Seshasayee, R. van der Wijngaart, and V. Sarkar. The Open Community Runtime Interface, 2015.

[8] S. Treichler, M. Bauer, and A. Aiken. Realm: An event-based low-level runtime for distributed memory architectures. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 263–276, New York, NY, USA, 2014. ACM.