Automatic Code Generation for an Asynchronous Task-based Runtime

Muthu Baskaran, Benoit Meister, Tom Henretty, Sanket Tavarageri, Benoit Pradelle,
Athanasios Konstantinidis, and Richard Lethin
Reservoir Labs Inc.
632 Broadway Suite 803
New York, NY
baskaran@reservoir.com

ABSTRACT

Hardware scaling considerations associated with the quest for exascale and extreme scale computing are driving system designers to consider event-driven-task (EDT)-oriented execution models for executing on deep hardware hierarchies. Further, for performance, productivity, and code sustainability reasons, there is an increasing demand for auto-parallelizing compiler technologies to automatically produce code for such asynchronous EDT-based runtimes. However achieving scalable performance in exascale systems with auto-generated codes is a non-trivial challenge. Some of the key requirements that are important to achieving good scalable performance across many exascale execution models and systems are: (1) scalable dynamic creation of task-dependence graph and spawning of tasks, (2) scalable creation and management of data blocks and communications, and (3) dynamic scheduling of tasks and movement of data blocks for scalable asynchronous execution. In this paper, we develop capabilities within R-Stream - an automatic source-to-source optimization compiler - for automatic generation and optimization of code targeted towards Open Community Runtime (OCR) - an exascale-ready asynchronous task-based runtime. We demonstrate the effectiveness of our techniques through performance improvements on a benchmark and a proxy application that are relevant to the exascale community.

1. INTRODUCTION

New processor and system architectures are being investigated and designed for exascale computing to address the principal constraints in reaching exascale, namely, power, performance, and resilience. An important characteristic of envisioned exascale hardware to improve power efficiency is near threshold voltage (NTV) – lowering supply voltage near threshold – that causes variations in device performance in addition to the intrinsic imbalance from the application itself. This means that mechanisms for dynamic load balancing increase in importance for near threshold computing (NTC).

Another aspect of exascale architectures is that the hierarchy of the hardware, which is currently at 3-5 levels (core, socket, chassis, rack...) will extend to 10 or more levels (3-4 levels on chip, and another 6+ levels of system packaging). In current systems, execution models address these levels through loop blocking or tiling [8], and stacked heterogeneous execution models (e.g., MPI+OpenMP). Such approaches will become cumbersome (program size, etc.) with the envisioned 10+ levels.

Event-driven task (EDT) execution models are emerging as an effective solution for new exascale architectures. Especially, EDT execution models are a very attractive choice for NTC and for seamlessly addressing deep hierarchy of processors and memories.

The EDT model supports the combination of different styles of parallelism (data, task, pipeline). At a very high-level, the EDT program expresses computation tasks which can: (1) produce and consume data, (2) produce and consume control events, (3) wait for data and events, and (4) produce or cancel other tasks. Dependences between tasks must be declared to the runtime, which keeps distributed queues of ready tasks (i.e., whose dependences have all been met) and decides where and when to schedule tasks. Work-stealing is used for load-balancing purposes [2].

It is impractical to expect programmers to write directly in EDT form; the expression of explicit dependences is cumbersome, requiring a significant expansion in the number of lines of code, and opaque to visual inspection and debugging. Direct EDT programming might be done in limited circumstances by some hero programmers, or in evaluation and experimentation with this execution model.

A high-level compiler and optimization tool is a key component of an exascale software stack, primarily, to attain performance, programmability, productivity, and sustainability for application software. R-Stream is a source-to-source automatic parallelization and optimization tool targeted at a wide range of architectures including multicores, GPGPU, and other hierarchical, heterogeneous architectures including the exascale architectures. Without automatic mapping, the management of extreme scale features will require longer software programs (more lines of code) to be written, thus require more effort to produce software, will be less portable, and may be error-prone. R-Stream provides advanced polyhedral optimization methods and is notable for features that can transform programs to find more con-

currency and locality, and for features that manage communications and memory hardware explicitly as a way of saving energy.

There are multiple EDT-based runtimes being developed in the community for exascale systems such as Open Community Runtime (OCR) [5], Concurrent Collections (CnC) [4], SWift Adaptive Runtime Machine (SWARM) [6], Realm [9], Charm++ [7], and others. In a previous work [10], we had developed a hierarchical mapping solution using auto-parallelizing compiler technology to target three different EDT runtimes - OCR, CnC, and SWARM. Specifically, we had developed (1) a mapping strategy with selective trade-offs between parallelism and locality to extract fine-grained EDTs, and (2) a retargetable runtime API that captures common aspects of the EDT programming model and uniformizes translation, porting, and comparisons between the different runtimes.

In this paper, we describe a dedicated backend in R-Stream for OCR to automate the synthesis of "scalable" OCR codes from simple sequential codes. To this end, we use advanced code analysis and transformation strategies of the R-Stream compiler.

2. BACKGROUND

2.1 R-Stream Compiler

R-Stream is a high-level automatic parallelization tool, performing mapping tasks, which include parallelism extraction, locality improvement, processor assignment, managing the data layout, and generating explicit data movements. R-Stream take sequential programs written in `C` as input, automatically determines the mapping based on the target machine, and emits transformed code. While R-Stream handles high-level transformations, the resulting source code still needs to be compiled via a traditional low-level compiler.

R-Stream works by creating a polyhedral abstraction from the input source. This abstraction is encapsulated by a generalized dependence graph (GDG), the representation used in the R-Stream polyhedral mapper.

R-Stream explores an unified space of all semantically legal sequences of traditional loop transformations. From a statement-centric point of view in the polyhedral abstraction, such a sequence of transformations is represented by a single schedule (i.e. a rectangular parametric integer matrix). The R-Stream optimizer adds capabilities to express the mathematical link between high-level abstract program properties and variables in this unified space. These properties include parallelism, locality, contiguity of memory references, vectorization/SIMDization and data layout permutations.

2.2 OCR

OCR is an open-source runtime system that presents a set of runtime APIs for asynchronous task-based parallel programming models suited for exascale systems. The main paradigms in OCR are: (1) Event-driven tasks (EDTs), (2) Data Blocks (DBs), and (3) Events. All EDTs, DBs, and events have a global unique ID (GUID) that identifies them across the system.

EDTs are the units of computation in OCR. All EDTs need to declare a set of dependencies to which DBs or events can be associated. An EDT does not begin execution until all its dependencies have been satisfied. EDTs are intended to be non-blocking pieces of code and they are expected to communicate with other EDTs through the DBs (which are the units of storage) and events. All user data needs to be in the form of DBs and to be controlled by the runtime since the runtime can relocate and replicate DBs for performance, power, or resilience reasons.

Events provide a mechanism for creating data and control dependencies in OCR. An event can be associated with a DB or empty. An event with a DB can be used to pass data to the EDTs waiting on the event (control+data dependence) and an event without a DB can be used to trigger EDTs waiting on the event (control dependence). Pure data dependence is encoded by attaching a DB in a dependence slot to an EDT.

The programmer (or compiler) creates an OCR program by constructing a dynamic graph of EDTs, DBs and events.

3. TECHNICAL APPROACH

We implement a backend to the R-Stream compiler that takes stylized C loop codes and generates parallel and locality-optimized OCR versions of the code. The approach involves polyhedral compiler optimizations to transform the code for exploiting data locality and exposing concurrency that is suited for asynchronous EDT-based execution in a (deeply) hierarchical architecture, and to identify and generate minimal communications in the code between different levels of memory. R-Stream creates "tiles" of computations and data that are then turned in to EDTs and data blocks, respectively (i.e. each computation tile is turned in to an EDT and each data tile is turned in to a data block). These tiles are created after applying optimal polyhedral transformations. R-Stream also captures the dependence between tiles in a concise internal representation, namely, dependence polyhedra. This tile dependence information is embedded in the generated code that dynamically creates the task-dependence graph needed by the runtime and sets the dependence between different EDTs (and data blocks) during execution. In further discussion, we use the terms "task" and "EDT" interchangeably.

3.1 Scalable spawning of tasks

R-Stream supports automatic generation of OCR code with on-the-fly scalable creation of EDTs. Creating all the EDTs at the beginning of the execution leads to non-scalability and adds a huge sequential "startup" overhead. R-Stream statically identifies (whenever possible) the set of EDTs that do not depend on any other EDT (i.e. that do not have a "predecessor" EDT) and generates code to populate and spawn them at the beginning. Each EDT is generated with an additional piece of code that embeds the necessary dependence information to create its successors (if they are not created already) and spawn them dynamically. This dynamic on-the-fly creation of EDTs is key for scalable execution of OCR code on large number of cores.

We briefly describe the technique in R-Stream to avoid the sequential startup overhead in task spawning. The overhead arises primarily due to the absence of a viable way to statically determine a unique predecessor task for a successor task that has multiple predecessors. We implement a technique called "autodecs" that dynamically resolves this problem. We represent the number of unsatisfied input (control) dependences of a task using a "counted dependence" and we

use polyhedral counting techniques to scan or enumerate the task dependence polyhedra to create the count. We let each predecessor of a task to decrement the count upon completion. The main idea of autodecs is that the first predecessor task to be able to decrement the counter of a successor task becomes the creator of the successor task. Unique creation of a counted dependence and hence a unique successor task creation is ensured through an atomic operation that avoids the race condition when two or more predecessors complete at the exact same time and become ready to create the same task.

3.2 Scalable creation and management of data blocks

R-Stream has the capability to generate local arrays for holding data within a computation tile or task and to generate explicit communication or data movement needed to transfer data between remote and local arrays. These features are exploited to generate data blocks for OCR code generation. The prior support for OCR code generation in R-Stream involves creating one large data block for each array, then create smaller data blocks within EDTs that fit in the local memory based on the data accessed within an EDT, and generate communications between the data blocks.

We extend and improve R-Stream's data block generation capability to provide scalable OCR data blocks support. The extended capability takes in a data partitioning (aka data tiling) specification from the user and creates data blocks (data tiles) according to the specification. R-Stream automatically identifies the data blocks that each EDT needs and creates an input slot for each data block. Within an EDT, the data needed by the EDT from each of its input data blocks is automatically copied on to temporary local arrays that collectively fit in the local memory (scratchpad or cache) attached to the processing element. This capability eliminates the need to create one large data block for each array and provides a pathway to achieve scalable performance.

3.3 R-Stream runtime layer

As mentioned above, R-Stream automatically identifies the dependence between different EDTs and the dependence between EDTs and data blocks, and automatically generates code for optimal on-the-fly EDT and data block creation. R-Stream has a light-weight runtime layer that operates on top of OCR and assists these capabilities, namely, on-the-fly creation of EDTs and data blocks, and dynamic spawning of EDTs. The runtime layer keeps track of active EDTs and data blocks, and implements (non-OCR-based) race-avoidance mechanisms to enable dynamic race-free creation of EDTs and data blocks. Currently these mechanisms are implemented to run on a shared memory system for proof-of-concept validations. In future, the runtime layer will be using one of the latest OCR features, namely, "GUID labeling", to implement the race-avoidance mechanisms. The R-Stream OCR backend will support GUID labeling once the APIs for GUID labeling are stabilized in the next public OCR release.

The R-Stream OCR backend has to be constantly kept in synchronization with the OCR release. This requires constant changes to the backend as the OCR APIs evolve. Once the R-Stream OCR backend supports GUID labeling we can also run optimized distributed OCR versions. Otherwise, we would have to resort to running non-optimized OCR codes on distributed memory nodes.

4. EXPERIMENTAL RESULTS

We present the results that we produced with R-Stream's automatic optimized OCR code generation capability in this section. These results clearly highlight the performance and productivity benefits that R-Stream compiler offers to an exascale software stack. We used R-Stream v3.15.0.1 for our experiments. We ran our experiments on a 48 core (96 thread) quad socket Intel Xeon (Ivy Bridge) server. We generate OCR code through R-Stream for a wide variety of kernels and benchmarks spanning multiple domains and application areas - linear algebra, multi-linear algebra (tensor computations), space-time adaptive processing (STAP), Synthetic Aperture Radar (SAR), and so on. In this paper, we discuss our code generation and optimization experiments on a benchmark (HPGMG) and a proxy application (CoSP2) that are relevant to the exascale community.

4.1 HPGMG: Chebyshev kernel

The HPGMG benchmark [1] consists of two different code bases - a finite volume code and a finite element code. We examined the timing of HPGMG benchmark and identified that the performance critical sections include smoothing, restriction, interpolation, and ghost zone exchange operations.

For our experiments, we focused on optimizing coarser regions of the HPGMG "Chebyshev" smoother kernel to exploit the opportunities in executing these regions in a more asynchronous fashion in an EDT-based runtime such as OCR. We isolated a coarse grain Chebyshev kernel representing the entire smooth function and parallelized with multiple methods, namely, hand parallelized OpenMP, R-Stream-generated OpenMP (with and without fusion of different sweeps of smoother), and R-Stream-generated OCR (with and without fusion of different sweeps of smoother), as shown in Figure 1. R-Stream-generated OCR code enables a scalable asynchronous EDT-based execution. As mentioned earlier, R-Stream has a lightweight runtime layer on top of OCR that enables on-the-fly just-in-time creation of EDTs and data blocks, and enables dynamic creation and handling of dependence events between EDTs. This avoids any unnecessary runtime overhead and enables scalable performance. This turned out be key for the Chebyshev kernel.

Further, R-Stream applies key compiler optimizations and generates OCR and OpenMP codes. For the Chebyshev kernel, the optimizations included - (1) smart fusion of Chebyshev smoother loops, (2) tiling across multiple smooth steps, and (3) autotuned tile dimensions. Due to the afore-mentioned compiler and runtime optimizations, R-Stream OCR code turned out to be the fastest among all parallelized codes when the number of threads is greater than 2. R-Stream OpenMP version turned out to be the fastest for the single thread and two thread runs, primarily due to the compiler optimizations. R-Stream OCR code was slower than the OpenMP versions for the single thread and two thread runs. For the other cases (number of threads > 2), R-Stream OCR code was up to 5x faster than hand parallelized OpenMP code, and further, R-Stream OCR code was also faster than fused, tiled and autotuned R-Stream OpenMP code.

Our experiments also showed super-linear speedups when scaling R-Stream OCR versions of the Chebyshev smoother from 1 to 32 OCR workers, as shown in Figure 2. We gener-

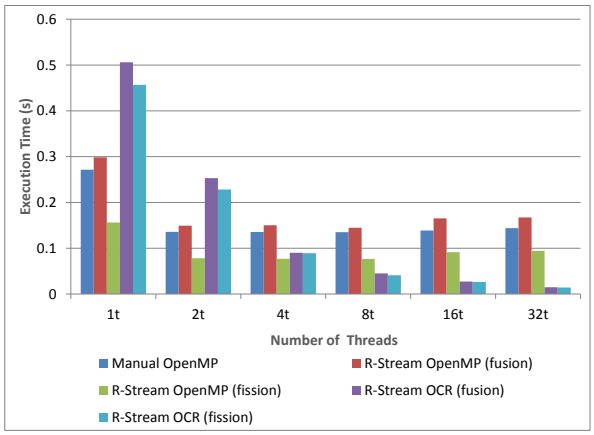

Figure 1: Chebyshev smoother performance on 64^3 array using multiple parallelization techniques for 1 to 32 threads. R-Stream OCR shows up to 5x performance increase vs hand parallelized OpenMP.

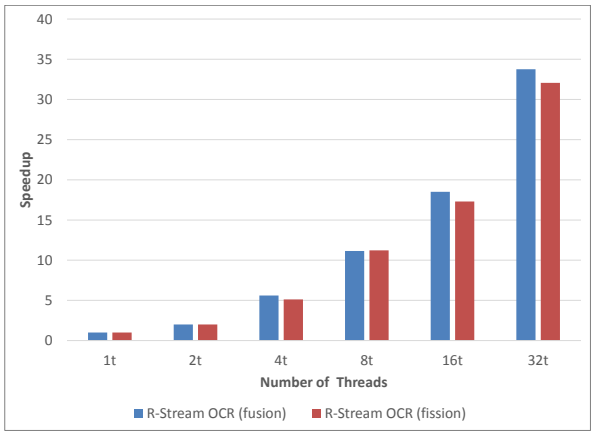

Figure 2: R-Stream OCR scalability: R-Stream OCR shows over 33x performance increase at 32 worker threads; the super-linear speedup is due to the fact the code is tuned for each worker count with a tile size that leads to better cache utilization and performance for that worker count.

ated tiled code and autotuned the tile sizes for each worker count. Super-linear speedup was observed at 4, 8, 16, and 32 workers. This is due to better cache utilization of the tiled code tuned for each worker count. The number of EDTs and the number of floating point operations per EDT depend on the tile size. The number of floating point operations per EDT in codes that gave high performance was typically between 128 K and 256 K. In total, using R-Stream we generated approximately 8.75 million lines of OCR Chebyshev smoother code consisting of approximately 3500 variants with 2500 lines of code each. The Chebyshev smoother code inputted to R-Stream has less than 100 lines of code.

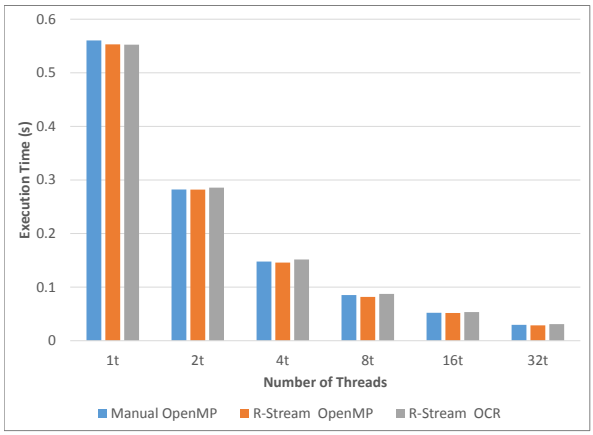

Figure 3: Performance of manual OpenMP version and R-Stream generated OpenMP and OCR versions of spmm code. R-Stream OCR version shows comparable performance with respect to the OpenMP versions.

4.2 CoSP2: sparse matrix matrix multiply kernel

We experimented with R-Stream's OCR code generation and optimization capabilities on the sparse matrix matrix multiply (spmm) kernel of the CoSP2 proxy application [3] from ExMatEx co-design center. The spmm kernel has indirect array accesses that make the computations irregular and thereby pose additional challenges to the compiler.

R-Stream successfully exploited the available concurrency in spmm code and generated a locality-optimized parallel OCR code. We ran the different versions of spmm code using a large sparse matrix of size 12288 x 12288 that has 196608 non-zeros. The performance results of manually parallelized OpenMP version, R-Stream generated OpenMP version and R-Stream generated OCR version are shown in Figure 3. The R-Stream OCR version exhibits comparable performance with respect to the OpenMP versions. The R-Stream OpenMP version is slightly better in most of the cases. This kernel has an outermost "doall" (synchronization-free parallel) loop that leaves the asynchronous runtime with no specific advantage to exploit. The proportion of runtime overhead in the overall execution time of this kernel is a possible reason for OCR version's slightly lower performance.

5. CONCLUSION AND FUTURE WORK

We have developed capabilities within R-Stream parallelizing compiler for automatic generation and optimization of code targeted towards OCR, an exascale-ready asynchronous task-based runtime. Through these capabilities we have demonstrated the following: (1) automatic code generation and data management enables high productivity, (2) the ability to find more concurrency and data locality, and generate different versions of locality-optimized parallel code improves performance (and energy efficiency), and (3) the ability to parallelize to an asynchronous EDT model in a scalable manner provides the basis for seamlessly scaling to adaptive exascale architectures.

6. REFERENCES

[1] M. F. Adams, J. Brown, J. Shalf, B. V. Straalen, E. Strohmaier, and S. Williams. HPGMG 1.0: A Benchmark for Ranking High Performance Computing Systems. *LBNL Technical Report, 2014, LBNL 6630E.*

[2] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, August 25 1996. (An early version appeared in the *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '95)*, pages 207–216, Santa Barbara, California, July 1995.).

[3] ExMatEx. CoSP2 Proxy Application. http://www.exmatex.org/cosp2.html.

[4] Intel. Concurrent collections. http://software.intel.com/en-us/articles/intel-concurrent-collections-for-cc/.

[5] Intel Open Source Technology Center. Open Community Runtime. `https://01.org/projects/open-community-runtime`.

[6] E. International. SWift adaptive runtime machine. http://www.etinternational.com/index.php/products/swarmbeta/.

[7] 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.

[8] M. S. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *ASPLOS-IV Proceedings - Forth International Conference on Architectural Support for Programming Languages and Operating Systems, Santa Clara, California, USA, April 8-11, 1991.*, pages 63–74, 1991.

[9] 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.

[10] N. Vasilache, M. M. Baskaran, T. Henretty, B. Meister, H. Langston, S. Tavarageri, and R. Lethin. A tale of three runtimes. *CoRR*, abs/1409.1914, 2014.