Implementing a High-level Tuning Language on the Open Community Runtime

[Experience Report]

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

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

ABSTRACT

The Open Community Runtime (OCR) is designed as a testbed for future exascale software technologies and techniques. The runtime is architected for modularity and extensibility, allowing researchers to create new components for testing new runtime strategies. One aspect of this design choice is reflected in the OCR hints API, which allows application code to communicate extra tuning information to the runtime. This extra information is optionally analyzed by runtime components and heuristics.

In this paper, we present our experience with building a high-level tuning abstraction on top of OCR, as well as with modifying OCR to add a small set of tuning hints. We describe the process of generating hint-annotated OCR code from a higher-level CnC application using the CnC-OCR framework. We also describe the process of modifying the runtime to support new hints, and new components to act on the tuning hints during program execution. Finally, we show preliminary results from our proof-of-concept implementation.

1. INTRODUCTION

The Open Community Runtime (OCR)[1] is an open source software project with the purpose of building a task-based runtime for future exascale systems. A major consideration in the runtime design was its modularity and extensibility, since the project aims to allow the community to contribute to the project by creating new components and testing new functionalities.

A *hints API* was added in the April 2015 workshop release of OCR, which allows an application programmer to annotate OCR objects (e.g., datablocks and EDTs) with additional information that might be useful for runtime heuristics. Shortly after the April workshop release, we began a project to add support to OCR for a small set of hints, which would be used by a high-level language on top of OCR; namely, CnC-OCR.[2]

This paper describes our experience with modifying the runtime to support new hints, and with generating the hint-annotated OCR code for CnC-OCR applications. Section 2 provides a brief overview of our tuning language for CnC, and the tuning hints we implemented for CnC-OCR application support. Section 3 outlines the process of generating OCR code, including hint API annotations, from our higher-level language. Section 4 discusses the changes we made to OCR to support the set of hints introduced in section 2. Section 5 provides some preliminary results from applying the hints to a few kernel applications. Section 6 explains the limitations we encountered when attempting to express additional hints in OCR. Section 7 summarizes the conclusions we drew from this overall experience.

2. HIGH-LEVEL TUNING LANGUAGE

CnC is a graph-based dependence programming model. The application is described as a graph of tasks and data, with dependence edges implicitly restricting the parallelism among the task nodes. In CnC-OCR, the application programmer writes a textual graph specification that describes the static dependence relationships among the dynamic task and data instances in the application. Each dynamic task and data instance in CnC-OCR is identified by a unique integer tuple value called a *tag*. Tunings are just ways of expressing additional constraints on the graph, and are declared in a separate tuning specification file. The graph and tuning specification files are used to generate a skeleton project, including scaffolding code to interface the programmer's task code with OCR. Tuning specification declarations are automatically translated into corresponding OCR hint API code within the scaffolding layer.

We implemented support for two classes of tuning hints in CnC-OCR: affinity hints, and task scheduling hints. The affinity tunings include explicit distribution functions, and a more high-level task-data affinity declaration. The distribution functions allow mapping instances of data and tasks (via their tag values) onto the set of available compute locations (via an internal OCR abstraction that currently corresponds to an MPI rank). The task-data affinity allows the programmer to constrain a task to be colocated with one of its dependences (input data instances).

We also implemented two task scheduling tuning hints: task priorities (which should be self-explanatory), and la-

[1] `https://xstackwiki.modelado.org/OCR`

[2] `https://habanero.rice.edu/cnc-ocr`

RESPA '15 Austin, Texas USA

beling of stoker/quencher tasks. *Stokers* are tasks that create more work (more tasks), whereas *quencher* tasks should mainly just do work. Any task not labeled as a stoker is assumed to be a quencher. By labeling *stoker* tasks as such, the runtime can throttle task creation when it already has plenty of work to do (by only scheduling *quencher* tasks). If the runtime is using a work-stealing scheduler, then thieves can give preference to *stoker* tasks when stealing, which should create more local work when run and thus delay the need to attempt more steals.

3. OCR CODE GENERATION

The CnC-OCR toolchain automatically generates OCR scaffolding code based on the dependences declared in the CnC graph specification, and the additional declarations in the tuning specification. For example, an EDT is generated to wrap each CnC computation task, and the generated code automatically sets up the EDT's dependences based on the declarations in the CnC graph specification. Since there is a 1:1 mapping between most of the basic concepts in CnC and OCR, making the process of generating scaffolding code between a CnC application and OCR pretty straightforward. Similarly, declarations in the CnC tuning specification are used to automatically generate hint-annotated OCR code.

Note that whereas OCR hints are specified inline throughout the application code, the CnC-OCR tunings are specified in a completely separate file. We believe this provides a better separation of concerns during the development process, and also makes it easier to get a big-picture view of an application's tunings (since all the tunings are in one place). We believe this is an important advantage available through higher-level abstractions built on top of OCR.

The CnC affinity tunings are implemented using OCR's affinity API extension. An affinity for an EDT or a datablock can be set directly via the *affinity* parameter when creating the EDT or datablock. This is equivalent to, but more succinct than, setting the *AFFINITY* hint on the EDT or datablock through the hints API; therefore, we bypass the hints API when generating affinity hint code. Since the tunings map directly to the affinities provided by the affinity API, these tunings are supported directly by the existing API (i.e., they do not require any runtime modifications).

Since the OCR hints API already includes a priority hint for EDTs, we use that hint to store our corresponding tuning hint value. However, the hint was only included in the API as an example of the types of information that can be passed through hints to the runtime, and the priority hints are simply ignored by the default scheduler. Setting an EDT's priority value requires four separate function calls to the hints API (in addition to the normal calls to `ocrEdtCreate` and `ocrAddDependence`).

The stoker tuning hint generation is very similar to the priority tuning; however, there is no existing hint corresponding to stoker tasks in the OCR hints API, so we had to add our own. The hint code is generated along with the EDT-creation code for a given CnC task, attaching the *STOKER* hint to a particular EDT instance if it is labeled as a stoker in the CnC tuning specification. Although this adds some verbosity to OCR code when hints are used, this is not a significant factor for our generated scaffolding code, since it is separate from the CnC-OCR application code and the application programmer can express the hints more succinctly in the CnC tuning specification.

Our experience with OCR code generation for tuned CnC-OCR applications should generalize to any higher-level programming abstraction on top of OCR, whether compiler-supported or library-based. This is because any abstraction built on top of OCR will use the same API, and thus will need roughly the same code to communicate tuning hints to the underlying runtime.

4. RUNTIME SUPPORT FOR HINTS

In order for our generated code to compile, we had to modify the OCR hint definitions to include our new hints. We also needed to add features to the OCR scheduler to observe and react to hints set in our generated code, and finally, we needed to change the runtime configuration to use the new features that we added to the scheduler.

4.1 Adding Hint Types

The hints API is very well documented, even explaining the exact steps to follow to add a new hint type.[3] Adding the *STOKER* hint type required changing just four lines of code across four different runtime source files.

4.2 Modifying the Scheduler

In addition to the hints API, the April 2015 workshop release of OCR also included a new and improved scheduler framework. The new scheduler is more modular, and the API is designed for better support of complex scheduler heuristics. We used this new scheduler to add runtime support for our two schedule-related hints.

The core scheduler behavior is handled by scheduler *objects* and *heuristics* (other types of components exist as well, but we only needed these two). Scheduler objects are mostly passive data structures, whereas the heuristics encapsulate the decision-making logic. Although a high-level document exists on the wiki, giving a sketch of the overall scheduler design, not much exists in the way of documenting the design of the individual scheduler components.[4] Fortunately, the default work-stealing scheduler provides a decent template for building a custom scheduler. Our strategy for implementing new scheduler functionalities was simply to copy the existing scheduler components and then modify the copied code just enough to get our desired behavior.

4.2.1 Stoker/Quencher Scheduler

The stoker/quencher hint requires partitioning tasks into these two groups, giving preference to quencher tasks over stoker tasks when doing local work, and giving the opposite preference for work stealing. We implemented this behavior by coping the existing work-stealing scheduler heuristic, and simply doubling the number of deque scheduler objects in the *deqs* array, where the deques at indices $2i$ and $2i + 1$ are the quencher and stoker deques (respectively) for the ith worker. We then modified the *EDT ready* action to examine the *STOKER* hint, and place new tasks into the proper deque accordingly. Finally, we modified the *get work* action to query the preferred deque first when looking for work locally or trying to steal work globally.

[3] `https://xstack.exascale-tech.com/wiki/index.php/OCR_Module_Hints`

[4] `https://xstack.exascale-tech.com/wiki/index.php/OCR_Module_Scheduler`

Since this update was implemented using existing scheduler objects (deques), the update was pretty straightforward and only required creating a new heuristic. The new stocker/quencher scheduler heuristic adds about 400 lines of code (LOC) to OCR; however, when compared against the default work-stealing heuristic, we see that we only inserted or modified about 50 LOC from the copy. In other words, if the existing heuristic were patched to support this hint (rather than creating a new heuristic), then the difference would only be about 50 LOC.

4.2.2 Priority Scheduler

The priority hint requires that tasks are acquired by workers in priority order. Supporting this hint required modifying both the heuristic and the underlying scheduler object data structures. We created a binary max-heap structure to store our prioritized tasks, a work-sharing *domain object* for centralized prioritization, and a priority scheduler heuristic. The binary heap scheduler object reads a task's *PRIORITY* hint, which is used as the weight for heap insertion. The work-sharing domain directs all requests for scheduling new tasks, or for acquiring tasks for execution, to the binary heap object. The priority heuristic similarly directs requests to the domain object.

The priority scheduler's heuristic is less complex than the default work-stealing heuristic because all of the logic is offloaded to the backing max-heap data structure. This actually required stripping out a lot of the copied component code in order to get the desired behavior. The priority scheduler components add about 1200 LOC to OCR, with about 230 LOC in the binary heap implementation. If this were applied as a patch to the default scheduler (rather than creating a new scheduler type), then the change would add/modify about 460 LOC. Note that these numbers include both the priority queuing logic as well as the work-sharing heuristic logic, whereas the previous example only contained partitioning logic, and thus required a much smaller change to the original code.

4.2.3 OCR Configuration

Interestingly, the most difficult part of the runtime modification process was generating an OCR configuration file to load our new scheduler components. In order to avoid the need to rebuild the runtime library every time you want to swap out a runtime component, OCR dynamically chooses its components during startup based on a configuration file. This is an implementation choice, and is not part of the standard. The configuration file structure is not well documented, and neither is the usage of the file in OCR's bootstrap process.

Initially, OCR seemed to stubbornly ignore the new configuration and continue to use the original settings—but we eventually deduced that the scheduler components declared in the configuration file were being loaded *positionally* (based on their declaration order in a header file) rather than using the component name from the configuration to look up the corresponding scheduler object. After reporting this strange behavior, the configuration file parser was eventually patched to be less rigid in the ordering requirements for scheduler component types. However, since the configuration process is not thoroughly documented, it is not clear if the patch was actually a bug fix or just an "improvement."

5. EXPERIMENTS AND RESULTS

We verified that our generated code and runtime modifications resulted in the expected behavior by tuning one small benchmark corresponding to each of our hints.

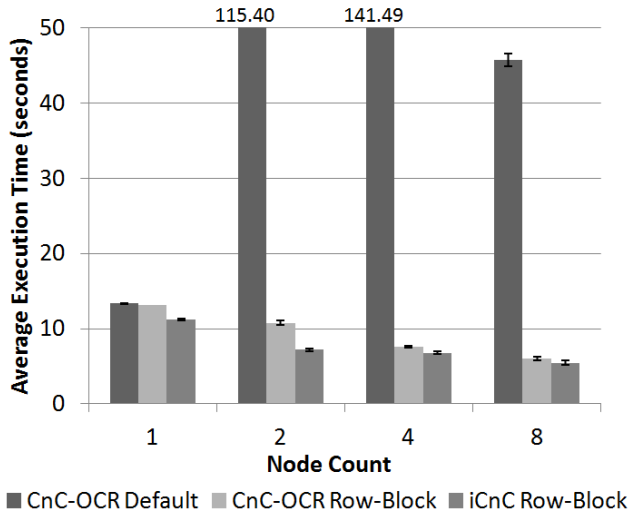

Figure 1: Smith-waterman distributed scaling results. Each input sequence contains about 200k nucleotides, and each tile comprises about 175×150 entries of the dynamic programming matrix.

To test distribution and affinity, we tuned the Smith-Waterman sequence alignment kernel. We ran the application with the default generated affinities—which corresponds to a column distribution in this case—and a row-block distribution, with 16 rows per block. The results are shown in figure 1. The default distribution showed negative scaling due to excessive communication overheads, whereas the tuned version showed positive (but still sub-linear) scaling. This is a significant improvement, especially considering that changing to a row-block distribution only required three short lines of code in a standalone tuning specification. When we ran the same application (with the same row-block tuning) on the Intel CnC (iCnC) runtime, we observed a similar scaling trend, suggesting that the suboptimal scaling is not an OCR-specific problem for this application and tuning configuration.

To test the priority scheduler, we used the N-queens kernel, which searches for a given number of solutions to placing N queens on an $N \times N$ chess board such that no queen threatens any other queen. The queens are placed row by row, and the search only continues to the next row if the current row's placement is legal. By prioritizing search tasks that are deeper in the search tree (i.e., attempting to place a queen on a higher-numbered row), we increase the likelihood of completing the search sooner. In contrast, if the shallow search tasks are prioritized (i.e., attempting to place a queen on a lower-numbered row), then we force the application to search almost the entire problem space before finding any solutions, which noticeably degrades the performance. Figure 2 shows the results. Again, each of these tunings is obtained by adding just one short line of code in a standalone tuning specification.

We built a custom "task-bomb" kernel to test the stoker

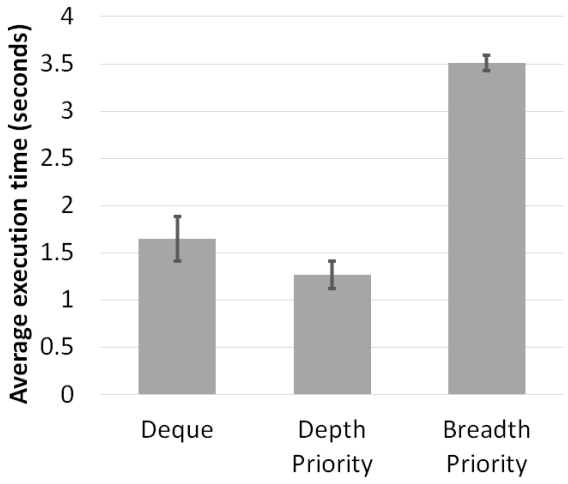

Figure 2: Performance results for N-queens kernel, searching for 5k solutions on a 13×13 board. Deque refers to the default (non-priority) scheduler. Depth refers to the case when $priority = row$. Breadth refers to the case when $priority = -row$.

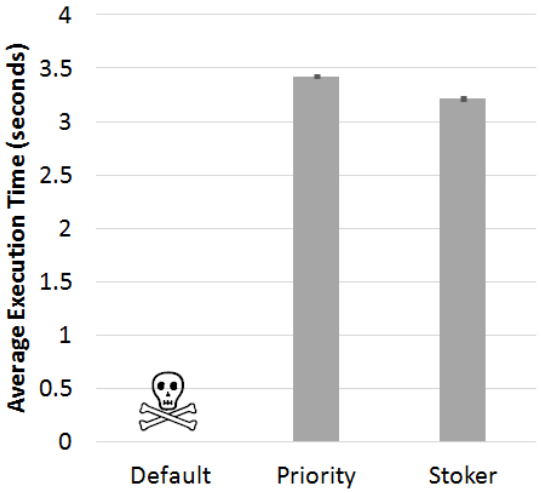

Figure 3: Performance results for the task-bomb kernel. Scheduler configurations include the default default deque scheduler, the priority-weight scheduler, and the stoker-quencher deque scheduler. The deque-based scheduler segfaulted on every run due to too many live tasks.

hint. This application creates 32 *stoker* tasks initially, each of which create 100 *quencher* tasks (which just busy loop), plus one more *stoker* task. This repeats for each of the initial *stoker* tasks 200 times. The results for this kernel are shown in figure 3. In the default deque-based scheduler, all of the *stoker* tasks execute before any *quencher* tasks (due to the LIFO behavior of the deques), which causes the scheduler to quickly exceed its maximum resource limits due to all of the live *quencher* tasks. If the *stoker* tasks are marked as such using our tuning—again, one line of tuning-spec code—then our modified scheduler can prefer running the *quencher* tasks, which allows the kernel to complete without running out of resources. Note that similar performance is achieved on this benchmark using our priority scheduler by setting the *quencher* tasks priorities higher relative to the *stoker* tasks. In a benchmark where work-stealing played a bigger role, we might see a bigger performance difference due to the special treatment of *stoker* tasks when stealing.

6. HINT LIMITATIONS

At the beginning of this project, we planned for another tuning hint related to data. The hint would allow a task to specify that it would only access some subrange of bytes from an input, and the runtime could optimize accordingly. However, we soon found that there was not a clean way to express this hint within the bounds of the current hints API. As a result, we have not yet implemented this hint.

The main issue with this hint is that it would be most naturally associated with an EDT slot in OCR, but the hints API does not allow hints to be attached to slots. An alternative solution would be to store the slot number along with the hint value, but then we would be limited to declaring this hint for only one input per EDT. This points to another limitation of the hints API: only a single scalar value can be associated with a given hint. For example, if we want to specify multiple options for the affinity of a task, that is not possible with the current hints API.

We considered that an acceptable workaround might be to statically declare multiple explicit copies of the hint to accommodate a bounded number of values for the same hint (e.g., *AFFINITY1*, *AFFINITY2*, ...). However, the runtime would need to check each of those separate hints individually when making EDT scheduling decisions, which might add an unacceptable overhead.

7. CONCLUSIONS

We were able to implement a high-level tuning language for CnC-OCR by generating code using the OCR hints API. Although the current hints API has limitations, it was expressive enough to implement a variety of interesting tuning options that demonstrated the ability to obviously influence the performance of the selected benchmarks. Using a high-level tuning language on top of OCR also provided a cleaner separation of concerns for application development than directly writing the inline OCR hints.

Much of the existing runtime code (especially the newer code) is well-designed and provides a good template for creating new runtime components; however, some of the older pieces of the runtime are not well documented, making them difficult to understand and work with. Although more complete documentation would have been helpful, overall we found it relatively easy to implementing new OCR scheduler components to support useful tunings for CnC-OCR. We believe that these findings, although currently limited to CnC on OCR, will also transfer to other abstractions built on top of the Open Community Runtime.

Acknowledgments

We would like to thank the OCR team at Intel for their guidance and support, especially Sanjay Chatterjee for his work on the hints API and CnC tuning. We would also like to thank Vivek Sarkar, Kath Knobe and Zoran Budimlić from the Habanero research group at Rice University for their feedback, as well as their contributions to other research on CnC programming model.