Final Scientific/Technical Report

Award #: **DE-SC0008717**
Recipient: **Intel Federal LLC**
Project Title: **TRALEIKA GLACIER X-STACK**
PI: **Shekhar Borkar**
Report Date: Sep 1, 2015

Acknowledgment: This material is based upon work supported by the Department of Energy [Office of Science] under Award Number DE-SC0008717.

Contents

Executive Summary 3
1. XStack Traleika Glacier (XSTG) Project Summary and Highlights....4
2. UC San Diego / San Diego Supercomputing Center (SDSC) 17
3. Pacific Northwest National Labs Applications and Runtime Work 27
4. Intel Applications and Kernels 43
5. University of Illinois, Urbana Campus - Applications and Productivity....49
6. Rice University 61
7. Reservoir Labs....69
8. Intel's Distributed Runtime OCR....81
9. Intel's Strawman Architecture and Simulator 92
10. University of Illinois, Urbana Campus - Architecture and Sensitivity Analysis 101

Executive Summary

The XStack Traleika Glacier (XSTG) project was a three-year research award for exploring a revolutionary exascale-class machine software framework. The XSTG program, including Intel, UC San Diego, Pacific Northwest National Lab, UIUC, Rice University, Reservoir Labs, ET International, and U. Delaware, had major accomplishments, insights, and products resulting from this three-year effort.

The XSTG project was formed to explore several fundamental challenges facing Exascale-class machines and their usability. Based on Intel's projections for what a severely power-constrained Exascale machine might require from architectural and network design changes, we identified major research challenges spanning the entire software and hardware stacks, which would require extensive co-design to resolve. These challenges included:

- Extract parallelism from applications
- Develop productivity tools for programmers
- Separation of concerns: system vs. user software
- Energy efficiency and scalability
- Implement and study task-based execution ecosystem
- Explore and assess architecture strawman prototype
- Co-Design with DOE Groups around all challenge areas

The combined efforts of the XSTG team tackled all of these challenges, each group using both similar and at times quite different approaches. The fundamental, central thesis of the XSTG project was that future machines would be forced to adopt throughput-optimized simpler accelerators for energy efficiency, vastly increasing core count to reach expected levels of performance. Under a profile study of expected fault models and upset events during a typical "uptime" requirement, it was also clear that the machine would need a self-adaptive distributed runtime layer to manage the large number of tasks, dynamically changing application and machine behaviors, and significant introspection and self-analysis features to reach the energy goals. A side effect of this design, if the computational tasks were relatively small, would be to provide programmer-transparent application resiliency during runs. Therefore, the XSTG built the central piece of the research project around an asynchronous, task-based execution model implemented as a portable, open-source distributed execution layer called the Open Community Runtime (OCR). All other work was based on or around the core OCR philosophy and design.

The collected body of work provided from the XSTG project includes the following major accomplishments:

- A formal, open specification for an asynchronous task-based execution model, named the Open Community Runtime
- An open-source OCR reference implementation that runs on x86 machines, single nodes and clusters
- Tools and infrastructure to debug and profile OCR applications and the OCR reference implementation
- Multiple programmer productivity tools to simplify programming in task-based models for OCR
- Many DOE mini-applications re-factored into OCR and productivity frameworks, all released open-source
- A viable strawman architecture and ISA for future Exascale class machines
- An open-source, massively parallel, high-performance functional simulator of the strawman architecture
- All tools needed to program and use the strawman architecture, also released as open-source

1. XStack Traleika Glacier (XSTG) Project Summary and Highlights

Introduction

The XStack Traleika Glacier (XSTG) project was a three-year research award for exploring a revolutionary exascale-class machine software framework. From the period of Sep 1, 2012 through Aug 31, 2015, the XSTG program, led by Intel, included major research contributions from Intel's partners at UC San Diego, Pacific Northwest National Lab, UIUC, Rice University, Reservoir Labs, ET International, and U. Delaware. This report summarizes the major accomplishments, insights, and products resulting from this three-year research program.

Organization of this report

This report is organized into 11 sections, using the outline below. While this first section provides a cross-section highlight of all the contributors, each major group has contributed their own section to deep-dive on their accomplishments and key learnings.

- Section 1: Project Summary and Highlights
- Section 2: UC San Diego / San Diego Supercomputing Center (SDSC)
- Section 3: PNNL
- Section 4: Intel Applications
- Section 5: UIUC Applications and Productivity
- Section 6: Rice
- Section 7: Reservoir Labs
- Section 8: Intel's Distributed Runtime OCR
- Section 9: Intel's Strawman Architecture and Simulator
- Section 10: UIUC Architecture and Sensitivity Analysis
- Section 11: XStack Contract Materials

Each section explores what technical research challenges that group was working on, the technical approach(es) taken to each challenge, key results or findings, and future work opportunities. This organization of the whole document and each section is meant to provide easy accessibility to reviewers for the content they will be most interested in understanding.

XStack Challenges

The XSTG project was formed to explore several fundamental challenges facing Exascale-class machines and their usability. Based on Intel's projections for what a severely power-constrained Exascale machine might require from architectural and network design changes, we identified major research challenges spanning the entire software and hardware stacks, which would require extensive co-design to resolve. These challenges included:

- Extract parallelism from applications
- Develop productivity tools for programmers
- Separation of concerns: system vs. user software
- Energy efficiency and scalability
- Implement and study task-based execution ecosystem
- Explore and assess architecture strawman prototype
- Co-Design with DOE Groups around all challenge areas

The combined efforts of the XSTG team tackled all of these challenges, each group using both similar and at times quite different approaches. The fundamental, central thesis of the XSTG project was that future machines would be forced to adopt throughput-optimized simpler cores for energy efficiency, vastly increasing core count to reach expected levels of performance. Under a profile study of expected fault models and upset events during a typical "uptime" requirement, it was also clear that the machine would need a self-adaptive distributed runtime layer to

manage the large number of tasks and memory regions, dynamically changing application and machine behaviors, and significant introspection and self-analysis features to reach the energy goals. A side effect of this design, if the computational tasks were relatively small, would be to provide programmer-transparent application resiliency during runs. Therefore, XSTG built the central piece of the research project around an asynchronous, task-based execution model implemented as a portable, open-source distributed execution layer called the Open Community Runtime (OCR). All other work was based on or around the core OCR philosophy and design.

Weekly Principal Investigator (PI) meetings were held to keep all efforts synchronized, as well as weekly "deep dive" technical two-hour meetings to explore specific topics or areas in more depth and with a wider audience, including DOE members. A higher-bandwidth forum of a co-design meeting, inviting all XSTG participants and DOE members as well as interested third parties, was held as a mini-workshop style meeting approximately every six months. This close level of interaction between algorithm specialists, application developers, tool makers, runtime developers, and processor/system design architects resulted in many findings, both positive and negative, that this report will review.

Technical Approaches

Applications

For application developers, the primary technical approach was to return to the original algorithm mathematical models and to re-constitute these into a native task-based execution model, in close conjunction with DOE experts at LLNL, LANL, and LBL. Select workloads or applications were further studied in both depth and breadth, trying various implementation strategies and different computational task granularities. These efforts included use of different tools, programming models, parallelization strategies, and task decomposition models. While an initial port might be made from transliterated MPI style programs, these were refined to adjust behaviors and communications to exploit the asynchronous task-based runtime platform that was the central thesis of the XSTG project.

As an example based on the DOE proxy CoMD, the UCSD team focused on the most time-consuming phases of the computation to identify high-value targets for task conversion. This included analysis of the CoMD nature from the very basic algorithm level, as shown in Figure 1.1 below. Further details of how UCSD transformed CoMD can be found in Section 2 of this report.

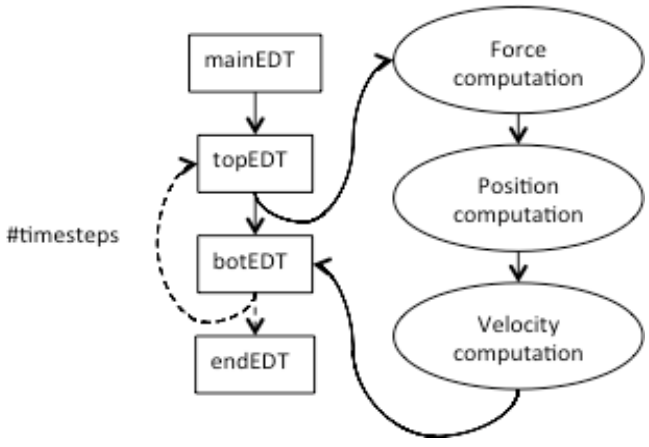


Figure 1.1

The PNNL team instead studied NWChem, LULESH, MiniGmG, HPGMG, and various kernels for their behavior and performance under task-based execution models. As PNNL explains in Section 3 of this report, the bulk of the inner kernel loops are embarrassingly parallel, making them easy targets to convert to task-based models. A complexity was found in the periodic use of reduction operations, however, which can incur some effort to program well in

OCR. Some applications, such as LULESH, were re-factored into alternate forms such as CnC, depicted in Figure 1.2 below.

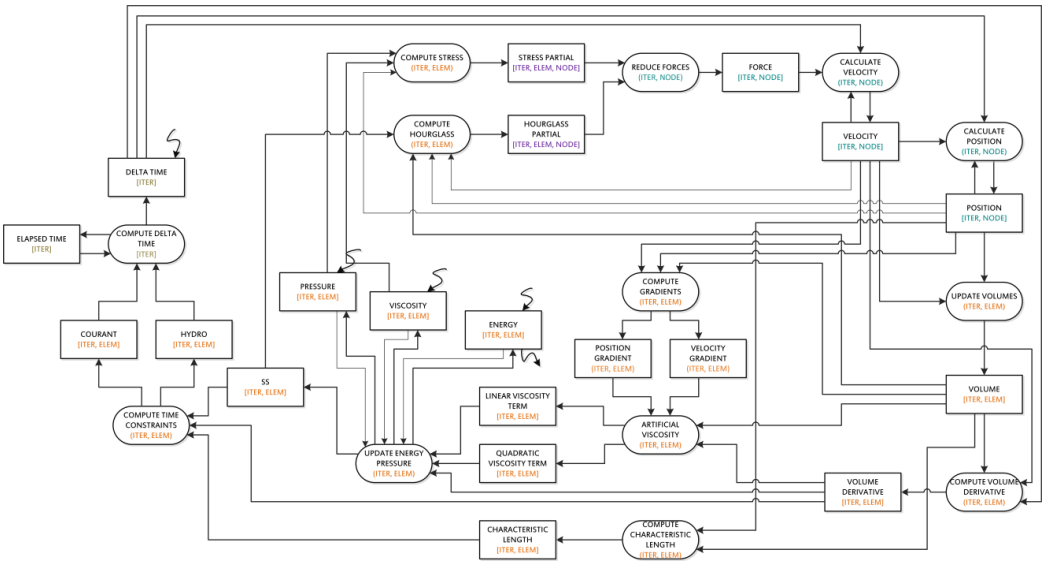


Figure 2.2

Giving primary focus to alternate representations as a productivity mechanism, the UIUC team under Prof. Padua studying Hierarchical Tiled Arrays (HTA) explored how to map data parallel operations onto tiles of large arrays. The underlying insight is a natural alignment of expression between what an application scientist is doing mathematically, and how it is rendered down to task-based execution models such as OCR. The mechanism of this conversion is similar to other Domain Specific Language (DSL) tools, as shown in Figure 1.3 below. The study of the NAS parallel benchmarks and their representation in HTA is explored in Section 5 of this report.

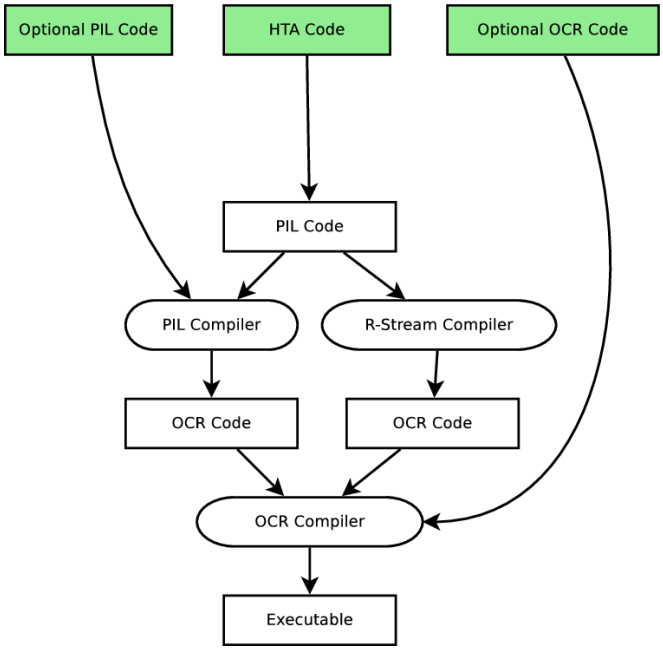


Figure 3.3

Tools and Productivity

While converting applications to OCR is manually possible, there is a significant loss in productivity from this model. While compilers have evolved to provide sophisticated automated support around function calling conventions and the ABI formats for host systems, task-based execution models – particularly those that support distributed execution models – have no such convenience and require the programmer to manually construct per-task heap, stack, and calling frames, complete with entry/exit marshal/unmarshal sequences for operands. Several teams explored productivity models to ameliorate this conversion burden.

As previously mentioned, the PNNL team explored using CnC for re-factoring applications. The advanced features of the CnC framework is that it automatically emits OCR tasks and datablocks from the source code form, since the CnC notation forces programmers to think in tasks and communication patterns. The goal of CnC is to simplify parallelism expression by allowing developers to work at a higher level than traditional parallelization techniques permit. The process starts by building a diagram that graphically describes an application's methods, data structures, and execution. Once the graph is defined, the application development can begin. While the prior CnC flow graph was quite detailed, PNNL evolved that design from a much simpler beginning, as shown in the first version of the CnC flow diagram below.

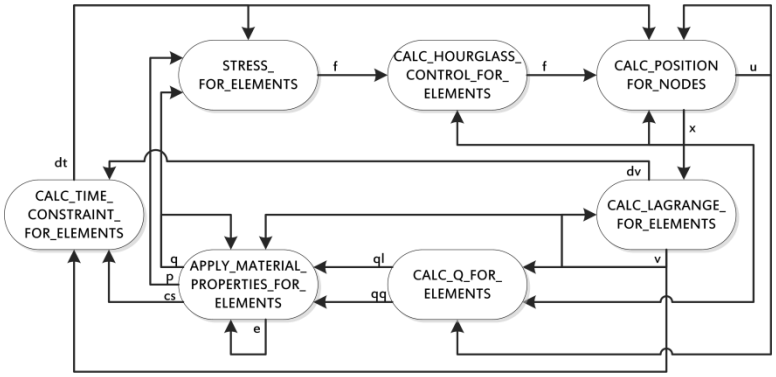


Figure 4.4

Similar to the PNNL team, the UCSD team also explored using CnC to refactor various applications or proxies. In particular, UCSD focused on CoMD, and how the CnC version behaved on top of OCR. The basic design of CoMD in CnC form is shown in Figure 1.5 below. As described in Section 2 of this report, in order to improve the performance of the CnC version of CoMD, the UCSD team had to change their parallelization strategy, which in turn led to modifications to the CnC graph. Working with the team at Rice University that owns CnC for OCR, this turned into an iterative process to eventually reach stable performance levels.

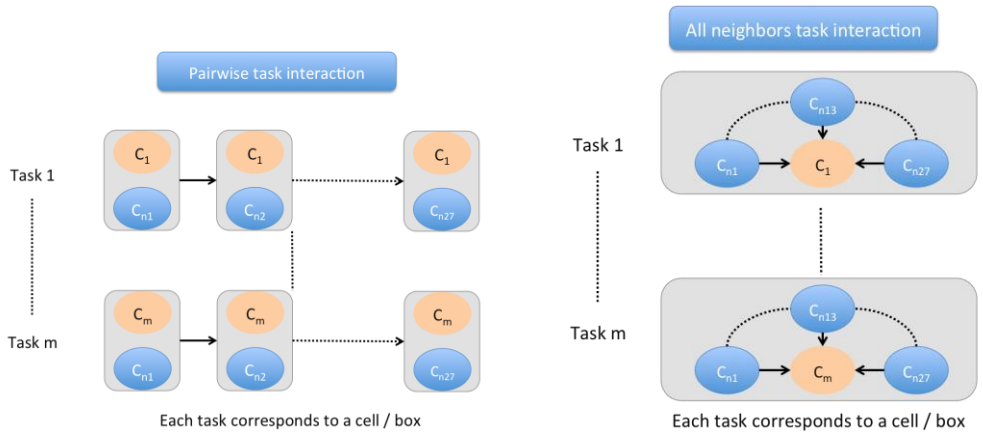


Figure 5.5

The actual work on converting CnC representations to OCR native programs was implemented by the Rice University team. The initial work was simply to build a native construct conversion table, such as the one shown below (Table 1.1) to map CnC idioms into OCR idioms. There was significant work in building this tool and associated profiling and debugging extensions. The work done in productivity frameworks, including CnC and Habanero C, are detailed in Section 6 of this report.

Concept	OCR construct	CnC construct
Task classes (code)	EDT template	Step collection
Task instance	EDT	Step instance
Data classes	All DBs have type void* (keeping track of individual DBs' types is the app programmer's responsibility)	Item collection
Data instance	Datablock	Item instance
Unique instance identifier	GUID	Tag (step tag / item key)
Dependence registration	Event add dependence	Item get
Dependence satisfaction	Event satisfy	Item put

Table 1.1

As another form of productivity tool beyond HTA, CnC, and Habanero, the Reservoir Labs team focused on further extensions to their highly specialized optimizing compiler, RStream. R-Stream takes in loop codes written in stylized sequential C and produces a parallel version of the input program. As detailed in Section 7 of this report, the Reservoir team focused on utilizing R-Stream for demonstrating the ability to automatically map to a deep hierarchical architecture such as XSTG, the ability to parallelize to the asynchronous event-driven task (EDT) runtime model, and the ability to assist the runtime to spawn tasks/computations and create/manage data in a more scalable manner suited for Exascale execution. An example of this productivity transformation is shown in Figure 1.6 below.

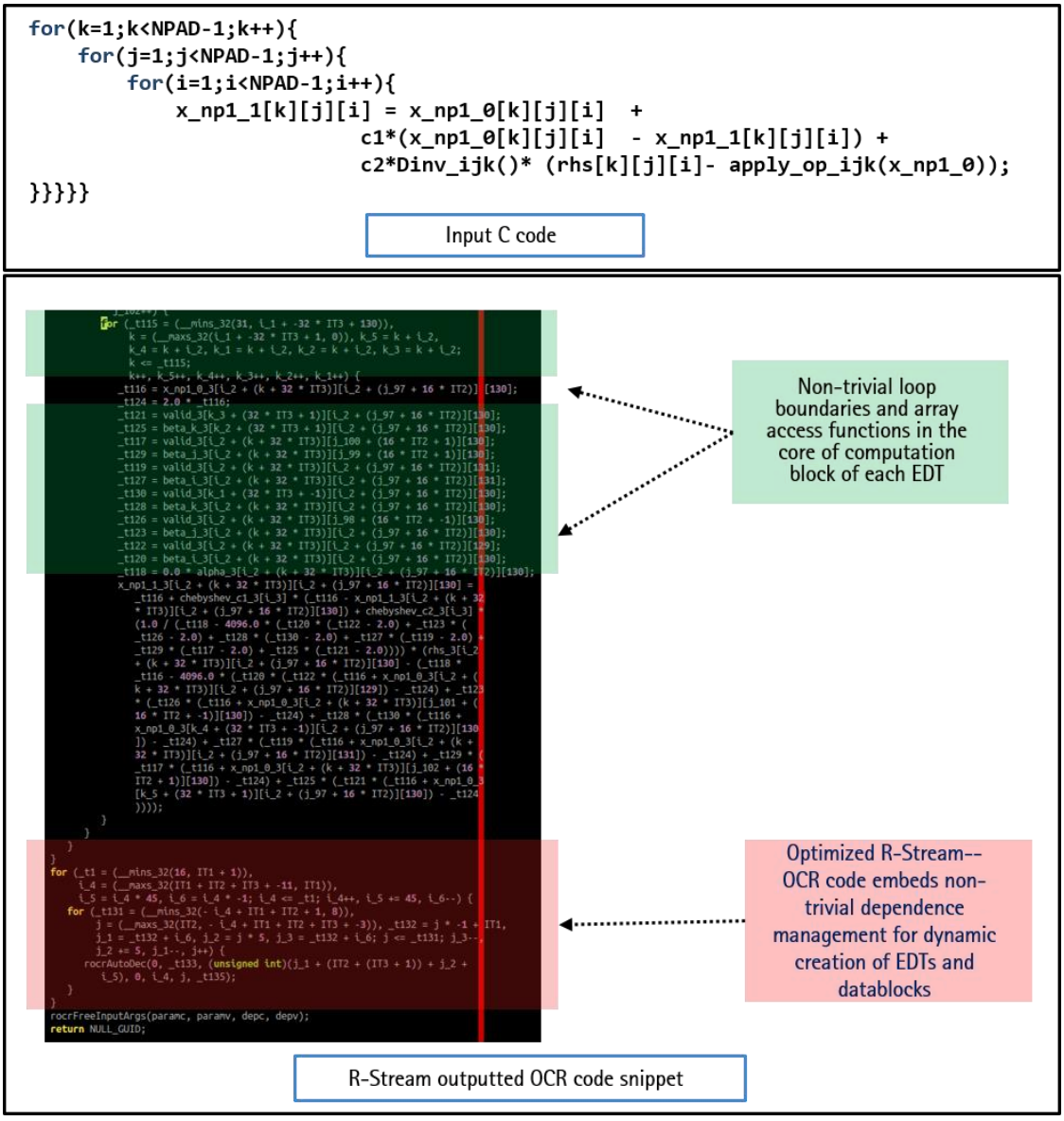


Figure 6.6

Runtime Design

The OCR reference implementation is very modular by design, facilitating various groups to work on different components (schedulers, allocators, etc.) independently while still maintaining compatibility with other components. The modularity also allows easy support for multiple platforms and maximizes code reuse. OCR currently supports x86, distributed x86 clusters, and the Exascale strawman architecture as encapsulated by FSim. This reference implementation is compliant with the formal OCR specification, setting a benchmark for other OCR implementations to both match and validate against. The design and implementation of OCR is detailed in Section 8 of this report. Emphasis on the initial reference implementation focused on performant individual modules (scheduler, data allocator, etc.) that were created within the hierarchical, modular OCR framework. This limits the performance losses OCR will incur in critical paths to the modularity and hierarchy, which can be optimized later. An example of this sub-component capability is shown in Figure 1.7 below, comparing the well-known DLmalloc routines against the OCR "Quick" allocator routines in execution time for random-sized objects.

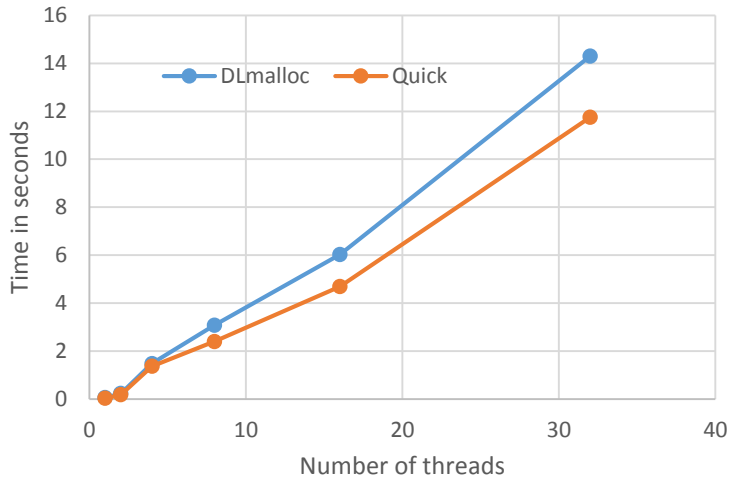


Figure 7.7

PNNL built their own prototype of a dynamic self-aware lightweight system, the Architected Composite Data Type Framework (ACDTF) – originally called the Rescinded Primitive Data Type (RPDT). This system was initially running on top of the OCR interfaces (ACDTF-OCR). The prototype features real time decision-based compression and algebraic invariant operations that use real-time sampling. The prototype has been implemented across a Linux based cluster and a shared memory node.

Results and Analysis

Applications

The UCSD team continues to test and assess their suite of applications and kernels against multiple platforms and multiple runtimes, including different OCR versions. One example of this exhaustive comparison is the measurement of CoMD implementation execution times, as shown in Table 1.2 below. This table also indirectly depicts how the relatively immature OCR framework is maturing over time, when compared to the multi-decade stable and highly optimized native MPI baseline versions of the program. Section 2 of this report further details these studies.

Cores	MPI	OCR old	OCR current	CnC old	CnC current	Asynchronous current
1	6.8	27	10.7	30.5	26.8	22.1
2	3.7	19	7.2	47.5	16.2	11.2
4	1.9	16	4.7	66.9	9.3	6.4
8	1.1	30	2.8	79.6	5.8	3.6
16	0.7	40	4.9	---	4.7	2.8

Table 1.2

The PNNL team obtained many positive results using their custom version of the OCR platform. Not only did the PNNL team compare against baseline versions, they also checked their results and performance against the reference version of OCR – and demonstrated that their refinements to both the runtime model and applications had strong benefits. As one example, the study of speedups for various small kernels for up to 32 cores can be seen in Figure 1.8 below. These results, and many others, are explained in depth in Section 3 of this report.

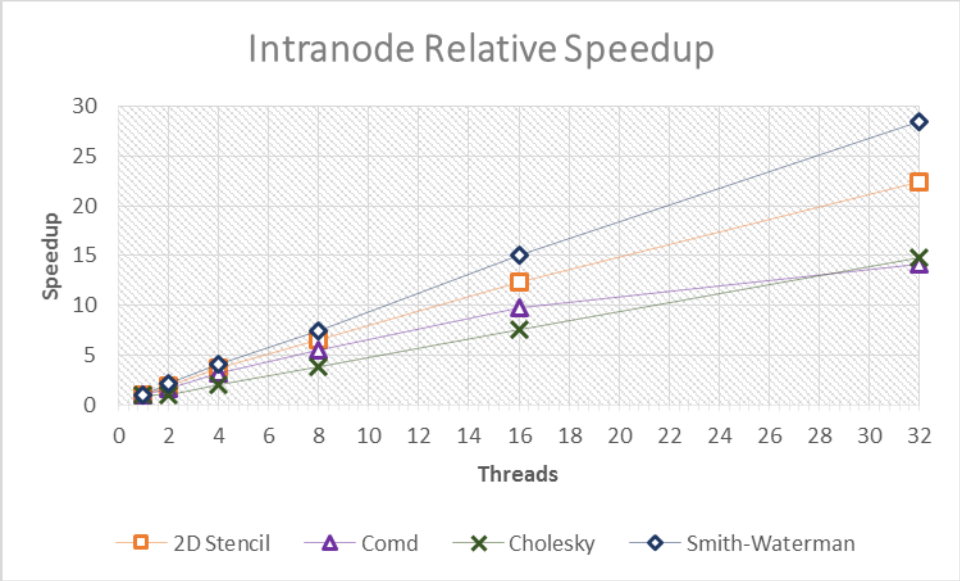


Figure 8.8

Tools and Productivity

The various CnC versions at Rice Univ. also generated widely varying comparisons between baseline versions (without OCR) and the enhanced versions (with versions of OCR). An example of the Cholesky decomposition on a 3000x3000 matrix using 50x50 tiles for the non-OCR native version (iCnC) and two OCR native versions (with and without CnC) is depicted in Figure 1.9 below by execution time. The sensitivity to the tile size parameter is also shown in the figure below. Full details and analysis are in Section 6 of this report.

Figure 9.9

The UIUC team under Professor Padua studied various aspects of task behavior within the OCR execution model, such as variations of task granularity and data accesses from within those tasks. While the full results are in Section 5 of this report, one demonstrative example is the comparison of OpenMP and OCR versions of Cholesky decomposition with 80x80 tiles and random amounts of work per tile, as shown in Figure 1.10 below. In this scenario, the OpenMP and OCR versions perform similarly for two of the three implementation strategies from within the HTA and PIL infrastructure.

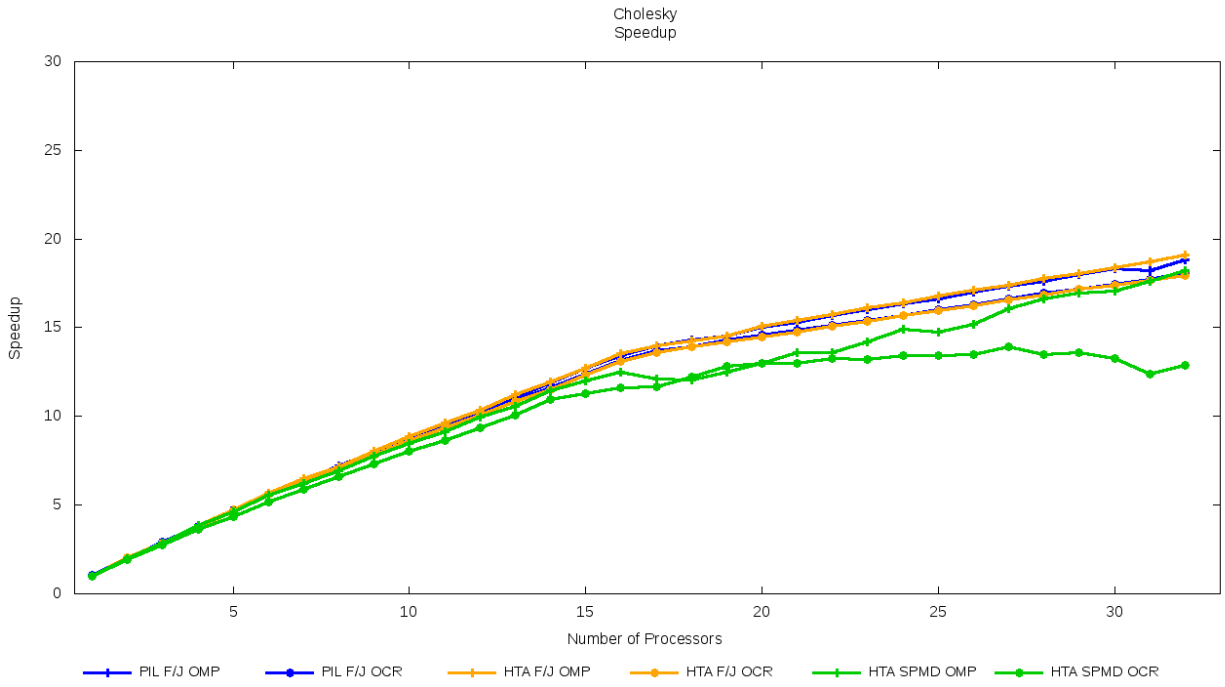


Figure 10.10

The Reservoir Labs team found similar results in their study of the HPGMG inner operations, when comparing against OpenMP and OCR. In Figure 1.11, the problem size was an 83 grid of 643 boxes, where the Y-axis represents the DOF/s (higher is better). The reason why mature OMP performs worse than immature OCR in one phase is explored in Section 7 of this report, along with many other aspects of OCR's performance.

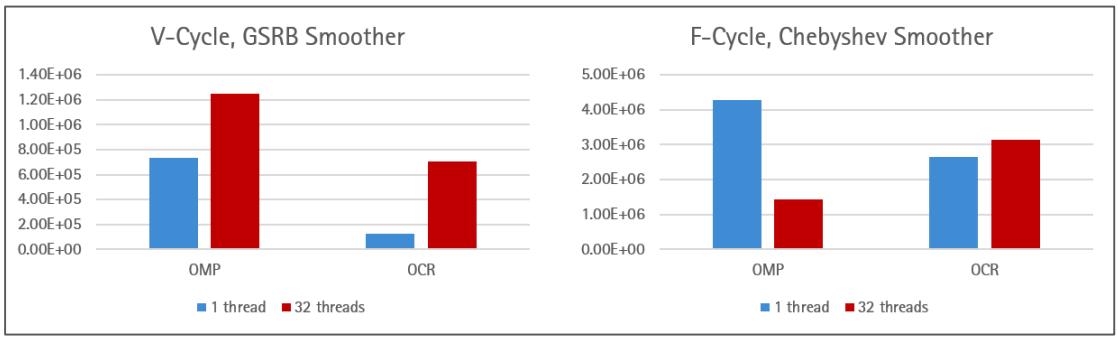


Figure 11.11

Runtime Design

Beyond implementing the basic, specification-compliant reference version of OCR, the collective teams added many extensions and enhancements to the basic design. One example is a dynamic hints framework, which allows richer semantic content to be passed from the application programmer to the runtime for smarter decision-making policies. When looking at a simple metric of "data hit rate" in a Cholesky decomposition, the use of such hints can drastically increase the efficiency of the execution as shown in Figure 1.12 below. These extensions are detailed in Section 8 of this report.

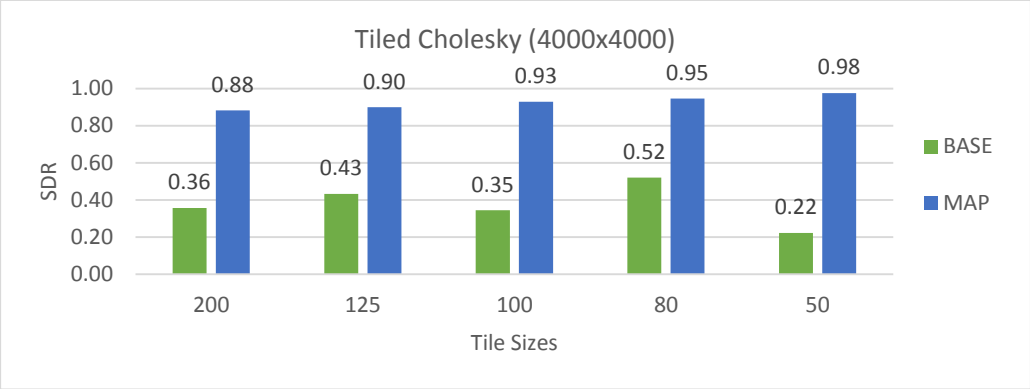


Figure 12.12

The PNNL team version of OCR, enhanced with their additional performance-based features, was compared against the reference community version of OCR in various ways, with one example result shown in Figure 1.13 below. Further details of the PNNL modified design and results are detailed in Section 3 of this report.

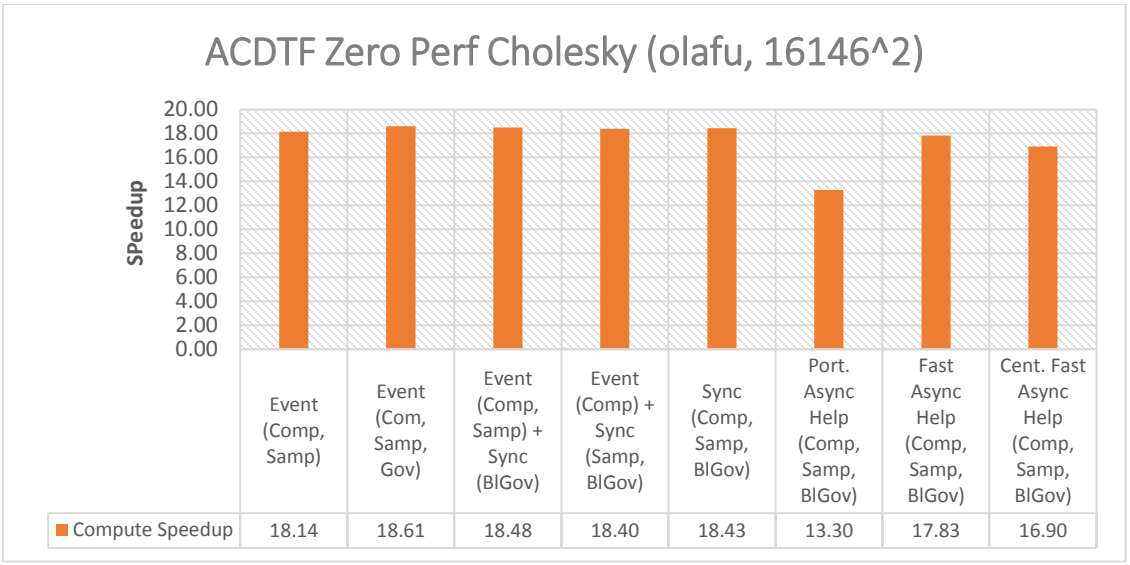


Figure 13.13

Architecture

Part of the research effort went into designing a prototype Exascale-class machine architecture, from the core ISA to the total system design. As detailed in Section 9 of this report, to reach the basic criteria for performance and energy efficiency constrained the architecture to use extreme parallelism via throughput accelerator cores that were highly customized. As depicted in Figure 1.14 below, the Intel architecture team designed a “block” model to replicate across a die, where each block had a control engine (CE) for running system software, and multiple execution engines (XE) for running user code. Each block or major memory subsystem is connected to the network via a next-level network interface (NLNI) component.

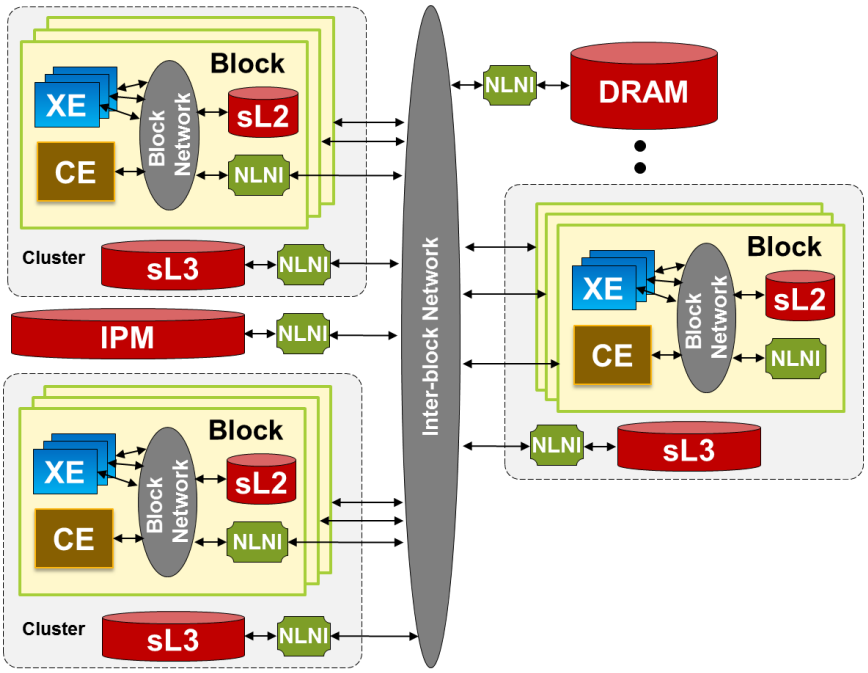


Figure 14.14

The UIUC team under Prof. Torrellas studied this proposed architecture and its evolution through the course of the XSTG project, as detailed in Section 10 of this report. These studies encompasses performance, behavior, and energy breakdown for various kernels and mini-applications. A study of the proposed XSTG strawman architecture for the energy breakdown across four small workloads and up to eight architecture "blocks" is shown in Figure 1.15 below. The dominate cost in most workloads is the movement of bits in and out of far-away memory structures.

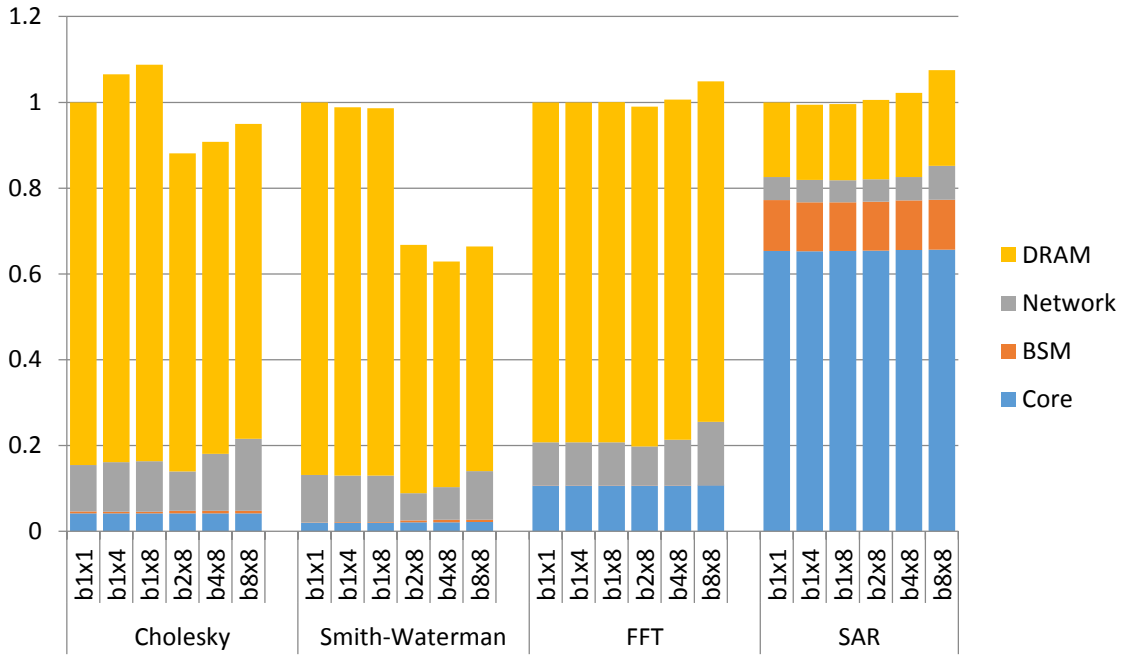


Figure 15.15

Significant Products

Several products, in the form of open-source code, were generated in the course of this project. Each of the major components is detailed below. All of these products are available online, with world-readable access, via the “git" repository located at: https://xstack.exascale-tech.com/git/public/xstack.git. There is abundant information, including guides, tutorials, and mailing list information available online at: https://xstack.exascale-tech.com.

Exascale-class Runtime Prototype

Open Community Runtime

The Open Community Runtime (OCR) was created to be an open-source, freely available reference implementation of a task-based execution model and related technologies. In order to provide a stable baseline for implementations, evaluations, and for community contribution, an OCR design document and formal specification was created and made available.

As part of an open reference implementation, Intel and its partners created a BSD-license based open source reference implementation to be a starting point for community involvement and evaluation. A public git repository of the source code, tools, and associated technologies was made available as part of the initial release.

Software Tools

R-Stream Translator

R-Stream is an advanced polymorphic source-to-source translator created by Reservoir Labs. In the course of the Intel XStack contract, Reservoir modified the R-Stream tool to understand and generate code for the OCR platform. This tool is freely available for evaluation and testing, but requires a paid license for production use. You can find more about R-Stream and examples here in the XSTG project code repository. More information about R-Stream from our project as well as downloading instructions are available online.

CnC

Concurrent Collections (CnC) is a technology pioneered by Kath Knobe and released as open source from Intel. The baseline version tries to make it easy to write C++ programs which take full advantage of the available parallelism. For the Intel XStack project, CnC was adapted to support OCR as a platform. Additional details on the use and design of this CnC version are available publicly. The licensing terms for this version of CnC are also available online. The full source code is available from the git repository.

Habanero C

Habanero-C is a set of extensions to the C language to support asynchronous communications and task-based execution models from Rice University. A reference implementation called HC-lib implements the Habanero-C components in a C library form. As part of the Intel XStack project, HC-lib was retargeted to use OCR as the runtime layer. The source code is hosted at the public github repository. The OCR repo contains only basic information publicly available and defers source access to the primary site. The HC-lib license terms are also available.

HTA

Hierarchically Tiled Arrays (HTA) is a project that comes from UIUC. HTA is a class designed to facilitate the writing of programs based on tiles in object-oriented languages. HTAs allow the programmer to exploit locality as well as to express parallelism with much less effort than other approaches. A port of HTA, with additional enhancements,

was part of the Intel XStack work. You can find the OCR-version of HTA online. Additional details about this version of HTA are available as well.

C Library and C support

Native OCR hand-written code is fairly restrictive on what the application programmer is allowed to do with violating the OCR memory and compute models by design. To facilitate easier programmer start-up, the XSTG project was extended by providing libC support on top of OCR for both x86 plaforms as well as the simulated next-generation XSTG platforms. This enables all OCR developers to start from a familiar baseline, and not worry about the initial conditions and restrictions. Full benefits require adopting of the new OCR features; this port is a productivity aid for programmers.

C++ Support

Similar to the porting of the libC support to run on top of OCR natively, preliminary work is further extending the libC support to include a C++ environment. This will allow all OCR programmers to benefit from C++ abstractions to increase productivity. For those using the DOE productivity frameworks built on top of C++ such as Charm++, RAJA, Kokkos, or Legion, this is a necessary precursor step to port those frameworks to the OCR environment. The initial support for C++ on OCR will be released during 2015. Current versions are available for comments and study in the gerrit code review system.

MPI Support

To increase productivity and decrease the getting-started costs for legacy software code libraries, a native MPI implementation on top of the OCR environment has been provided for the common subset of MPI operations. This will allow existing MPI programs to be recompiled, without modification, and run in the OCR environment. While this will not provide the full benefits of a re-factored OCR application, the goal is to enable fast start of applications in the OCR environment. This enables developers to focus on the most important code portions first to convert to the OCR model, leaving the surrounding MPI code unchanged. Source code for the MPI-lite implementation is available in the OCR code repo. This is an ongoing release, where additional features are added on a regular basis.

ROSE Translator

While a native MPI library on top of OCR will facilitate faster ramp-up on OCR programming, the MPI abstraction model itself is not radically dissimilar to task-based execution models. Academic research projects such as UCSD's Bamboo have explored mechanical transformation of MPI programs into task-based execution programs. The key insight is between MPI calls, the code is very similar to a "task" while the MPI calls themselves represent explicit communications. As part of the ongoing work around the XSTG project, the ROSE translator tool is under evaluation for creating such an automated tool for production use. If successful, this will be part of a future open-source release that will further enhance the XSTG project by adding another type of productivity tool for application developers to bootstrap their task-based execution model conversion. Preliminary tools based on the ROSE framework are available in the public git repository already.

Execution Visualization & Profiling

OCR is bundled with tools to aid the programmer in visually understanding the correctness of, and performance bottlenecks, in their programs.

- The Visualization Tool presents the application as a flow graph and the task execution visually as a timeline. Example visualizations can be found online.
- The instrumentation-based runtime profiler is useful to identify bottlenecks in the runtime. It provides a more accurate measurement of overheads than general-purpose sampling-based tools.

Applications and Examples

DOE Proxy Apps and Select Kernels

Many of the DOE Proxy Apps (CoMD, HPGMG, MiniAMR, etc.) as well as various computational kernels (DGEMM, Stencil1D, SmithWaterman, etc.) were converted to be OCR-native programs. An overall table of these re-worked applications can be found online, in a summary table format: Available Applications, Proxies and Kernels (Refactored and Original). This table also includes links the actual source code, both modified and the unmodified original, for both illustrative purposes and performance/capability assessments.

Prototype Exascale Future Architecture Simulation (FSim)

FSim

As part of the Intel XStack project, a key design goal was to put together a "strawman" future Exascale machine architecture. This architecture had to reflect the expected properties of memory, interconnect, compute, power, reliability, and related issues. The OCR design emerged as a natural way to target such a future architecture, but required an evaluation platform to model that architecture. A fast functional simulator, FSim, was created to model the Intel XStack Traleika Glacier strawman architecture. An initial version of FSim was released open-source on August 31st 2015. Several additional modules were developed or extended to create this back-end simulator, as detailed below, and these tools are also part of the open-source release with FSim.

LLVM

A port of LLVM 3.6 was implemented to target the new ISA of the simulated architecture depicted by FSim. The LLVM 3.6 framework is the current version targeting FSim, to provide better code generation and feature support.

GNU Binutils

Similar to LLVM, a port of GNU Binutils 2.24.51 to the XSTG platform was created. The GNU binutils version 2.24.51 targets the current FSim platform, and is used for the GAS, LD, and associated other tools (objdump, etc.).

QEMU

The strawman architecture simulated by FSim includes two distinct ISA forms: x86 cores for runtime management and policy, with a different ISA for Execution Engines (XE) accelerators. Rather than re-implement an x86 simulator, the FSim effort adapted the open-source QEMU system and turned it into a plug-in module for FSim use. The QEMU version 2.2.1 used in FSim supports native 64-bit execution modes and a large number of platform-specific features for FSim's requirements.

GDB

To address an application-debugging problem that the Intel XStack project deferred, Intel is working on a preliminary port of the GNU Debugger (GDB) to the simulated XSTG architecture. This is to provide a significant increase in programmer productivity, since application debugging via simulation machine traces is slow and inefficient.

2. UC San Diego / San Diego Supercomputing Center (SDSC)

By Pietro Cicotti, Manu Shantharam, Laura Carrington

Challenge Focus

SDSC focused on the following challenges areas:

- Implement and study task-based execution ecosystem,
- Extract parallelism from applications,
- Energy efficiency and scalability, and

- Co-Design with DoE Groups.

All these areas are integral part of the co-design process. In our case, this process involved refactoring or completely redesigning CoMD and HPGMG, two proxy applications from the DoE co-design centers ExMatEx and ExaCT. As part of this process we reformulated a traditional Simple Instruction Multiple Data (SIMD) algorithm that used the message passing interface (MPI) to an Open Community Runtime (OCR) task-based model. In doing so, we evaluated the inherent parallelism of the application and its granularity. We evaluated the new implementations for performance, on x86 architecture, and energy on the Traleika Glacier functional simulator (FSIM). Throughout the development we collaborated closely with ExMatEx team members David Richards (LLNL) and Susan Mnizewski (LANL) for CoMD. We have been in contact with Sam Williams (LBL) for the work on HPGMG. The HPGMG design and implementation was presented to us by Sam Williams of LBNL. During this presentation Sam did a code walkthrough and suggested the configuration we could start our OCR implementation with. After the implementation of the HPGMG OCR code, we presented our progress and discussed our implementation with Sam. This meeting was set up by Gabriele Jost.

Technical Approach

Our investigation approach, for the challenges listed above, was to directly address the challenges as part of the design and development of proxy applications using OCR. Inevitably, during the design phase we devised a work decomposition strategy that reflects the inherent parallelism of the numerical method and the domain problem, which for both CoMD and HPGMG was similar to the respective proxy application, but with a finer-granularity approach. Then we compared performance and scaling to the reference implementation. In the case of CoMD, we also explored different variants of the algorithm, the potential used, as well as different programming models. The remainder of this section describes in detail the design of some of the over 10 different versions of CoMD we developed using different programming models, different parallelization strategies, and code restructure to avoid synchronization as well as our development of a version of OCR HPGMG.

CoMD

Molecular Dynamics (MD) is widely used in material science, chemical physics, and the modeling of biomolecules. MD is a computer simulation technique for modeling the physical movement of interacting atoms by numerically solving Newton's equations of motion. Forces between atoms are defined by molecular mechanics force fields or potentials. CoMD is a proxy application developed to represent classical molecular dynamics workloads and is based on the MD codes ddcMD and Scalable Parallel Short-range Molecular Dynamics (SPaSM).

The force computation is the most time consuming phase and it is therefore the focus of most optimizations. A brute force search for neighboring atoms requires N^2 distance calculations (N distances for each of the N atoms) to determine which atoms fall within the cutoff distance; such an approach is extremely inefficient. In order to identify suitable atom pairs, CoMD uses link-cells: the space is partitioned by applying a regular rectangular decomposition, with the largest number of cells such that each cell exceeds in size the cutoff distance, in every dimension; in this way, the neighbors of an atom need only be searched within 27 cells (the cell containing the atom and the 26 neighboring cells). By using link-cells, the computational complexity is reduced to linear in N, since the number of atoms per link cell is essentially bounded.

Forces are symmetric between 2 atoms and need only be calculated once. While this principle presents an opportunity for reducing the computational workload when the calculation is parallelized, as in most multithreaded approaches, race conditions arise if two concurrent pair evaluations try to simultaneously increment the forces and energies on the same atom. A tradeoff has to be made between managing race conditions and duplicating the force calculations.

The MPI reference version uses a 3-D spatial Cartesian domain decomposition to distribute the atoms across processors. The local domain is split into link cells assuming periodic boundaries. Atom data consists of position, velocity (momentum), force, and energy. All local atom data is stored as a structure of arrays (SoA). A halo of link

cells from replicated adjacent ranks is needed when computing local forces. Velocities and positions are advanced per link cell prior to the halo exchange. The local force computation is sequential and symmetric between atoms in each link cell and between atoms in neighboring link cells. Partial results for link cells are produced relative to the local domain atoms.

CoMD: Threaded

To evaluate different variants of CoMD, we developed and compared multi-threaded versions of the code.

MPI+OpenMP There are two OpenMP implementations that we considered: a reference provided by ExMatEx, and one that we implemented. Both are very simple and only differ in that the reference computes the forces between atoms in different cells twice, to avoid concurrent updates. In both, the outer loops in the force calculation are preceded by the *parallel for* directive to partition and assign the set of local cells to threads. A reduction aggregates the total energy from all the threads. In addition, our implementation uses the force symmetry optimization and avoids race conditions preceding updates to the force data with the *atomic* directive.

MPI+Pthreads The PThreads version is similar in design to the OpenMP implementation: a master thread partitions and assigns the set of local cells to worker threads. Since the iteration space does not change between loops (all the outer loops iterate over the local cells), partition and mapping are static; the master thread sets the *body* function and signals the start of the parallel phase. As with OpenMP, for PThreads we compared two variants: one that uses force symmetry and one that doesn't. PThreads is a low-level explicit model that gives more control than OpenMP on implementation details (of course at the cost of more lines of code and added complexity). In this work it enabled comparisons between locking mechanisms, partition strategies, and scheduling policies

A simple division of the work based on the cell ids may not always result in an optimal mapping. The ids of local cells are assigned using a simple enumeration in the three dimensions. The local space is scanned in all the dimensions by moving along one dimension until the boundaries are encountered, and then wrapping around and incrementing the next dimension, and so on. For example, if the number of cells is divisible by the number of threads, the space is partitioned along a single dimension (e.g. divided into identical planes with a certain thickness). In order to experiment with different partitions we implemented a variant that takes parameters to define how to partition the space within a process and assign cells to threads, similar to the way the space is decomposed and assigned to different MPI processes.

Finally, we developed a PThreads variant that implements a work-stealing policy. In this variant, a bitmap is used to represent the local cells. As before, each worker has a range of cells assigned, and starts working on its set of cells. As work progresses, threads flip bits in the bitmap accordingly, to indicate work that has been completed or is in progress. Then, when a thread runs out of work, it starts scanning the bitmap in search of more work, and when available cells are found, that work is claimed. After a complete scan the threads go to sleep waiting for a new parallel phase. This dynamic scheduling version provided a comparison point including the potential benefits of dynamic scheduling (e.g. load balancing) and the overhead of its implementation.

CoMD: OCR

The first step in refactoring an existing application for an event-driven programming model like OCR consists of identifying the basic steps of the computation, and extracting the underlying data and control flow. The granularity of the steps is somewhat arbitrary, and represents a tradeoff between parallelism, and complexity and overhead. With finer tasks, there is more parallelism available but the dependency graph grows larger and becomes more complex, thereby resulting in more overhead – in the form of runtime system scheduling and dependency checking, startup and termination of the tasks, etc.).

Focusing on the most time consuming phases of the computation, functions and loops are ideal candidates to be transformed into tasks. Both functions and loops provide natural boundaries for defining tasks; in addition, the

signature of the function helps in identifying the data accessed and modified, such data corresponds to the dependencies of the task.

The force computation involves concurrent updates; we defined two types of EDTs for the force computation: one that allows concurrency by using atomic updates (implemented with 64bit *test _and_swap*) and one that avoids concurrency entirely at the cost of duplicating the computation for each pair of interacting atoms.

CoMD uses a regular rectangular spatial decomposition to partition the space into the smallest number of cells such that cells exceed in size the *cutoff* threshold in each dimension (in this way, only neighboring cells need to be scanned when computing forces). While in CoMD atoms and cells are stored in global vectors with a simple mapping to determine the boundaries of a cell or the index of an atom, in OCR data structures should reflect the granularity of the dependencies and the EDTs, therefore there needs to be a separation of cells and atoms in different DBs. Cells are individually allocated as DBs, and the whole space is defined by a list of DB identifiers. This partition supports the EDTs defined above such that an EDT can update velocity or position of the atoms of a cell (e.g. such EDT depends on one DB), while a force EDT can compute forces between atoms of neighboring cells. We defined two types of EDTs to evaluate two levels of granularity: the finest is an EDT that computes the forces between the atoms of two neighboring cells, and the coarsest is an EDT that computes the forces between the atoms of a cell and all its neighboring cells.

The resulting graph of EDTs has a pattern of repeated EDTs that make up the main *timestep* loop, which is preceded by initialization EDTs (set up data structures etc.) and followed by termination EDTs (reporting results and de-allocating data). In addition, the EDTs of the main loop fork and join groups of EDTs for the computational phases discussed. Figure 2.16 shows the high level structure of the EDT graph. On the left side, the timestep loop is unrolled as a sequence of topEDT-botEDT pairs, in which the topEDT forks the computation phases. When the computation phases end the botEDT spawns a new topEDT, for a new iteration, or the endEDT to complete the computation.

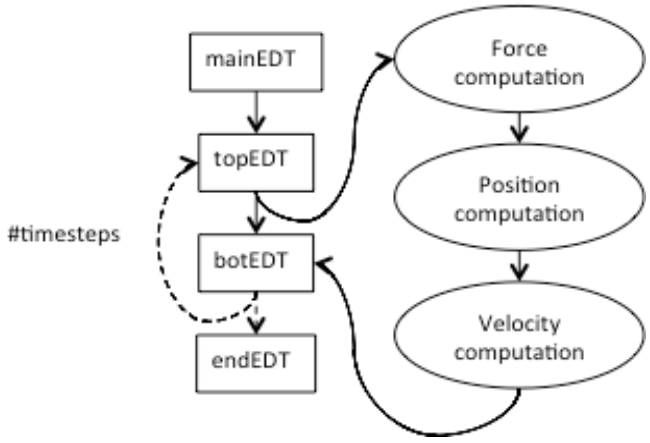


Figure 2.16: Timestep loop in CoMD-OCR.

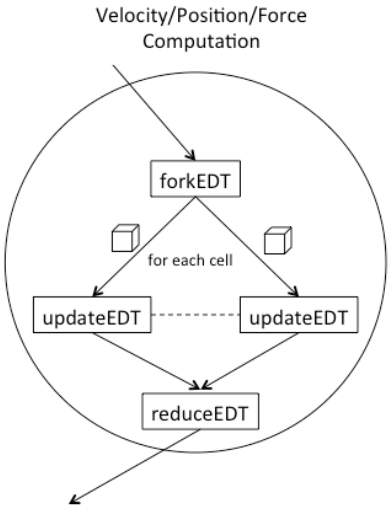


Figure 2.17: Fork/Join computation phase in CoMD-OCR.

Error! Reference source not found..2 shows a high level representation of the computation sub-graphs. The left graph represents the velocity and position updates: in both cases, a *root* EDT forks an EDT per cell, which performs the update on the cell that it receives as input. Each of the update EDTs signals the *continuation* EDT, which in the

case of the velocity update is the forkEDT of the position update, and in the case of the position update it is the forkEDT for the force computation. In the force computation, at least in the simplest case, the graph has the same structure but there is a larger number of EDTs, that is one for each pair of neighboring cells (received as input), and there is a reduceEDT which collects all the potential contributions to update the system potential. Finally, the reduceEDT signal the botEDT for continuation.

A second implementation was later designed and developed using a globally asynchronous approach, rather than a fork/join approach. In the second implementation of CoMD there are *lanes* of EDTs, one per cell, and in each lane EDTs continually generate successive EDTs, and only synchronize with the neighbors' lanes via data dependencies. Figure 2.18 shows the alternating of phases without global synchronizations; the dependencies established between EDTs computing on neighboring cells define localized synchronizations between a small set of EDTs. The only exception is when after a predefined number of iterations, the total energy of the system is computed via reduction and printed for tracking progress and correctness (ensuring that the energy is preserved).

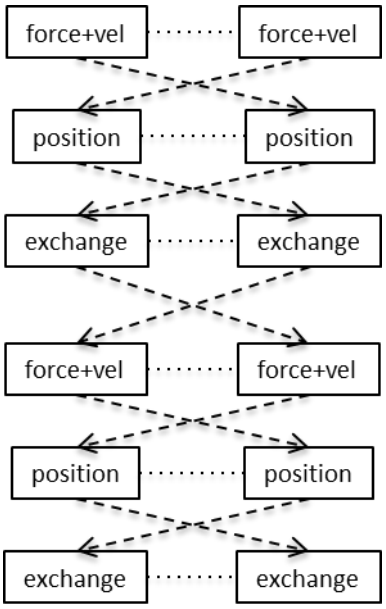


Figure 2.18: Asynchronus CoMD-OCR. Dependencies and EDTs.

CoMD: CnC-OCR

The CnC-OCR version of the CoMD application we implemented supports force computation using Lennard-Jones Potential and Embedded Atom Method Potential. Additionally, we implemented two task decomposition strategies. The first version is based on the pairwise task interaction, wherein each task corresponds to a cell and the force computations for that cell is performed using a cell pair (the cell and one of its neighbors), one neighbor at a time. Figure 2.194 illustrates this version. The second version is based on "all neighbors" task interaction (see Figure 2.19). Here, each task performs the force computations using all 27 neighbors at once. Thus, the parallelism exposed in the second version is coarser than the first version.

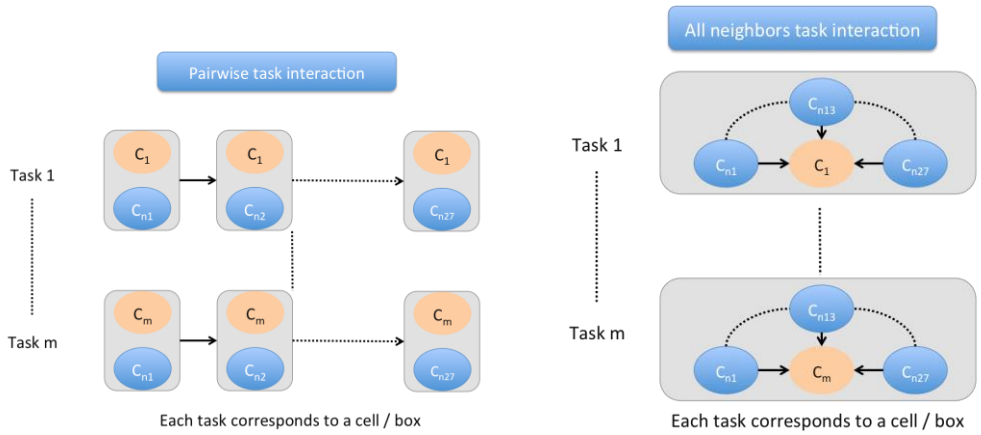


Figure 2.19: Pairwise (left) and all neighbor (right) decompositions.

Implementation challenges and experiences using OCR-CnC:

- Syntax of CnC-OCR was different from Intel CnC. This was initially an issue, as Intel CnC tutorials did not map directly to CnC-OCR, and there was no tutorial for CnC-OCR.
- As a newbie to CnC, coming up with a CnC graph for CoMD was a challenge. But once we had a graph, porting the application to CnC was straightforward.
- In order to improve the performance, we had to change our parallelization strategy, which in turn led to modifications to the CnC graph. But the modified graph did not work due to some underlying CnC-OCR limitations. An iterative design/implementation process between SDSC and Rice resolved this issue.
- Debugging the CnC-OCR code was an issue as there were no easy debugging techniques.

HPGMG

The finite volume version of the HPGMG benchmark (HPGMG-FV) proxies' finite volume based geometric multigrid linear solvers. HPGMG-FV implements a full multigrid solver (FMG), which includes an F-Cycle and a V-Cycle at each level of the FMG solve. The computation of HPGMG-FV is classified into the following operators based on their functionalities: restriction, interpolation, residual, and smooth. The restriction and interpolation operators are used to obtain coarse and fine grids, respectively. The smooth operator is used to solve a system of linear equations at different levels of the FMG solve. In terms of domain and data decomposition, HPGMG-FV creates a hierarchy of levels, with each level partitioned into *boxes*, where a box represents a cubical region of space. These boxes store all the required data to solve the linear system.

HPGMG: OCR

HPGMG-FV supports many flavors of the smooth operator, such as Gauss Seidel Red Black (GSRB) and Chebychev, and solvers such as BiCGStab and CG. As an initial port of this code, based on discussions with Sam Williams and the Reservoir team we specifically selected the following configuration for our OCR implementation: solver – BiCGStab, smoother – Cheby, full multigrid cycle (FMGCycle). It is this configuration that Sam feels will be used on Exascale systems for GMG calculations. The following figure (Figure 2.5) illustrates the overall structure of our OCR implementation. In HPGMG OCR, we have implemented two types of EDTs – level and box EDTs. The level EDTs correspond to each level of the FMGCycle. The main functions of the level EDTs are to spawn box EDTs and to synchronize between the level EDTs (different operator EDTs like restrict_level_edt and smooth_level_edt). The main computation happens in the box EDTs as operators are implemented as box EDTs. For example, the

smooth_edt implements the Cheby smoother. In terms of the important data structures, we used *level* and *box* structures to represent level and box related data. For example, the *level* structure includes a list of box-guids, dimension of each box, and so on. The *box* structure consists of data such as all the variables and box-id.

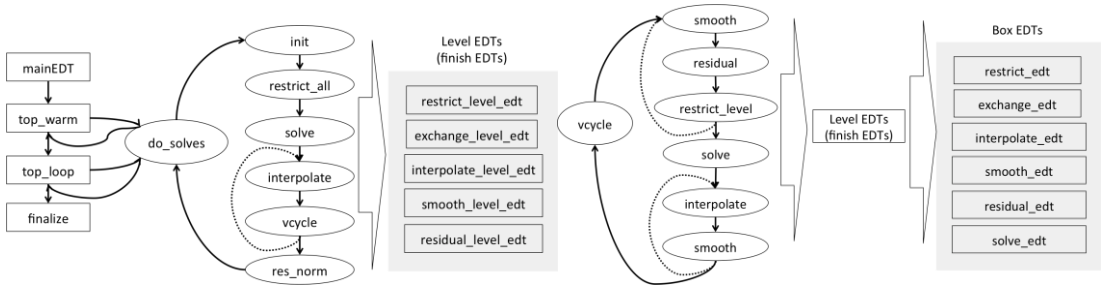


Figure 2.20: Overall structure of HPGMG OCR implementation

Implementation challenges and experiences

- Just coming up with a skeletal code for our OCR implementation of HPGMG took a lot of effort due to the complex nature of the HPGMG code.
- Although we reused some of the original HPGMG code, we had to code from scratch some operations like interpolate and restrict.
- Mapping MPI to OCR in terms of data movement was the most difficult part, especially understanding and deconstruction the communication in the original HPGMG code and then mapping it to EDTs and data dependences.
- Compared to the CnC-OCR port, this was very tedious and error prone. Even as an application developer, one needs to understand the subtlety of the OCR intrinsic operations.

Although there is some debugging capability built into x86 version of OCR, the TG version has no debugging capability, thus making it hard to resolve issues on TG.

Results and Analysis

In this section we report on the experiments with CoMD and HPGMG, and discuss relevant results.

CoMD

CoMD: Threaded Experiments with multithreaded versions of CoMD revealed performance bottlenecks that severely limit performance and scalability. First, it was observed that the reference implementation has near perfect weak scaling and strong scales with almost perfect efficiency. However, in strong scaling, the data overhead due to halo cells quickly surpasses 100%; multithreading reduces the memory footprint and eliminates part of this increasing overhead, a potential performance advantage on future deep memory hierarchies. The cost of concurrent accesses is high, even when using fast atomic operations, due to cache coherency; relying on a dynamic approach in which the scheduler avoids race conditions (and the need for atomic operations) and preserve locality may be beneficial for programming, portability, and performance. Using a dynamic scheduler that automatically load balances will improve performance in future systems in which cores operate at different speeds due to manufacturing variations or power and heat management. However, at fine granularities, which favor load balancing, there is significant overhead from increased surface to volume ratio and from scheduling and work management.

CoMD: OCR

Performance of running the OCR implementation is not comparable to the reference implementation, due to the high overhead in OCR and its limited scalability. This limitation of OCR affected our ability to effectively explore scaling with OCR. Nevertheless, comparing different variants of the OCR implementation offered some insight.

Extracting fine-grained parallelism is possible, but it can incur high overheads. With OCR we observed that defining tasks that update forces between a pair of cells (~18000FLOPs/EDT) has more than 20x the parallelism compared to defining tasks that update forces for a cell and its 26 neighbors (250000 FLOPs/EDT), but it incurs overheads that slows down execution by as much as 10x. DBs are not created and destroyed frequently, to avoid overhead, but are relatively large for the coarse grain case (14-31 DBs of approximately 6KB); This is another limiting factor to how big and EDT can be while avoiding spilling from fast access memory (e.g. L1 caches or scratchpad).

Even with OCR, using force symmetry is advantageous and achieves better performance than duplicating force computations.

Finally, the fully asynchronous formulation of CoMD achieved better performance on 16 cores, despite being slower on 1 core, because it appears to be more scalable than the fork/join approach of generating EDTs, as shown in .

Cores	Fork/Join	Asynchronous
1	10.7s	22.1s
2	7.2s	11.2s
4	4.7s	6.4s
8	2.8s	3.6s
16	4.9s	2.8s

Table 2.1: Runtime comparison between fork/join and asynchronous CoMD-OCR implementations.

CoMD: CnC

shows the results of our experimental evaluation. The numbers within "( )" indicate the slowdown of our CnC implementation on the x86 architecture compared to the MPI version. As can be observed from the above table, for LJ based version, both pairwise and "all neighbors" implementation perform poorly compared to the MPI version. However the latter is almost 2x faster than the former. We believe this speedup is due the coarser granularity of parallelism exposed by the "all neighbors" decomposition. Additionally, note that neither decomposition scaled well. We only implemented "all neighbors" for EAM based CoMD, and observed performance behavior similar to that of LJ Force based CoMD.

	LJ Force		EAM	
#Threads	Pairwise	All Neighbors	Pairwise	All Neighbors
1	30.49 (0.27)	19.80 (0.41)	-	92.87 (0.19)
2	47.47 (0.17)	18.61 (0.44)	-	62.03 (0.29)

4	66.87 (0.12)	21.50 (0.38)	-	57.93 (0.30)
8	78.57 (0.10)	32.80 (0.25)	-	71.30 (0.25)

Table 2.2

The CoMD-CnC version was also much slower than the CoMD-OCR version, with more than a 2x slowdown on 8 cores for LJ and 1.5x slowdown for EAM.

CoMD: Investigating the progress of OCR throughout the TG project

The development and changes of OCR throughout the TG project have been significant. One aspect of the development of OCR-CoMD has enabled the testing of OCR. In addition, having an OCR application that was not a toy kernel helped identify performance and scalability issues within OCR and progress was made to address those. below illustrates the progress of performance and scalability of OCR throughout the project. The performance of the MPI reference version of CoMD is shown in the 2nd column as a comparison. The table shows how the current version of OCR has improved the performance of CoMD OCR by a factor of 20 and 7 for CoMD-CnC on 16 cores. Also the current version of OCR shows good scaling up to 8 cores whereas the previous version only scaled to 4 cores. The asynchronous CoMD-OCR which was designed with scaling in mind shows scaling up to 16 cores using the current version of OCR. The CnC version is similar with scaling up to 16 cores where the older version of OCR only scaled to 8 cores.

Cores	MPI	OCR old	OCR current	CnC old	CnC current	Asynchronous current
1	6.8	27	10.7	30.5	26.8	22.1
2	3.7	19	7.2	47.5	16.2	11.2
4	1.9	16	4.7	66.9	9.3	6.4
8	1.1	30	2.8	79.6	5.8	3.6
16	0.7	40	4.9	---	4.7	2.8

Table 2.3. CoMD OCR & CnC-OCR comparison using current and old version of OCR. OCR old and CnC old refers to branch runtime/master commit 5c1bf712, OCR current and CnC current refer to apps/master commit 248db157.

CoMD: FSIM

Using the functional simulator we compared the energy consumption of 3 variants of CoMD-OCR and CoMD-CnC. The variants of CoMD-OCR were: no force symmetry, force symmetry with atomic updates, and force symmetry with exclusive write access on DBs, which is controlled by the OCR scheduler. Among these, we observed that the no force symmetry variant used approximately 50% more energy than the other two variants, which effectively execute fewer instructions; between the latter two, using exclusive write access to DBs is consuming slightly less energy and is to be preferred for its greater portability. The CoMD-CnC appeared much less efficient, consuming more than twice the resources and energy. Figure 2.212.6 shows the dynamic energy cost of instructions for each XE executing.

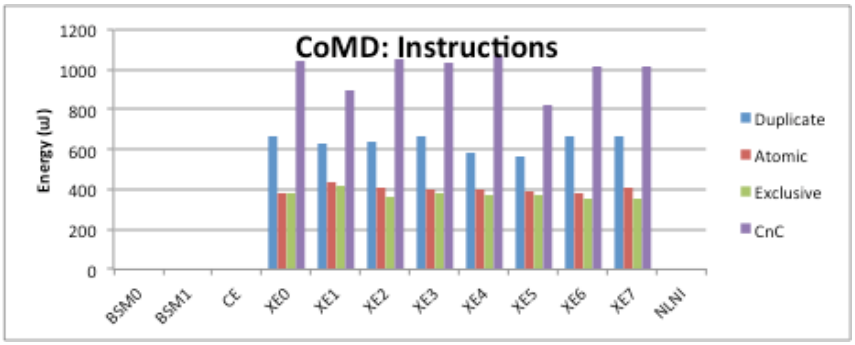


Figure 2.21: Dynamic energy for CoMD-OCR variants and CoMD-CnC.

HPGMG:OCR

Figure 2.227 shows the performance of HPGMG OCR on x86 architecture with respect to the reference MPI implementation. We observed that for a single thread case, both implementations had similar performance behavior, but as we increased the number of threads, the reference implementation performance much better. Our results indicated that OCR implementation does not scale beyond 4 threads. We also observed similar behavior for the smooth operator (which is the most timing consuming part of HPGMG OCR).

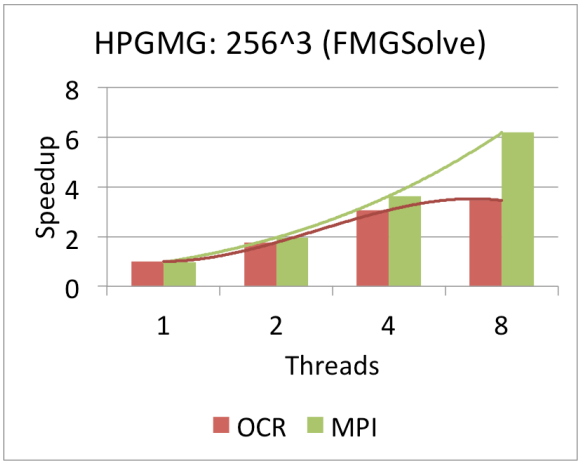


Figure 2.22: Performance of HPGMG OCR vs MPI

Figure 2.23 shows the impact of cache sizes on the energy consumption of HPGMG OCR on the TG architecture. We observed that as we increased the cache size from 32 KB to 128 KB, the dynamic energy decreased. Since HPGMG OCR is memory bound, change cache size did not yield significant benefits (around 5% – 7%) in terms of decreased energy consumption.

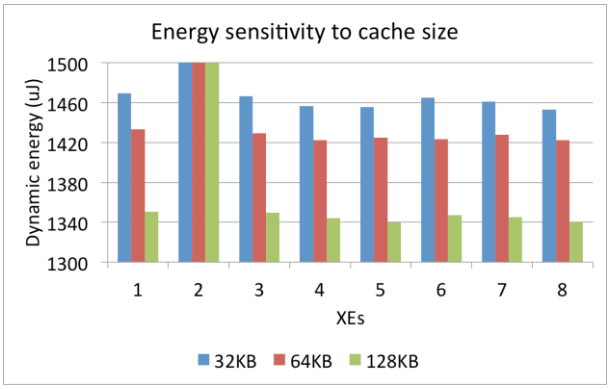


Figure 2.23

Recommendations and Future Directions

Future work should continue exploring the affective performance at scale of OCR. Our ability to effectively conduct research and test performance at scale has been affected by the severe performance bottlenecks in OCR. Freezing the API, and focus on performance should take priority and precede changes to the API, extensions, or any feature addition. In addition, the study of methodology for porting applications should be extended further to verify that OCR can indeed support fine granularity and scaling; in particular, such a study should evaluate the complexities involved with dynamically generating a EDT graph asynchronously.

Higher level programming models should also be carefully reviewed from a performance perspective. While we found that CnC can significantly improve coding productivity by essentially generating OCR for the programmer, the observed performance penalties and cost in terms of resources used (i.e. memory and energy) appears unacceptable for exascale computing.

Products and Bibliography

Publications

- An Evaluation of Threaded Models for a Classical MD Proxy Application, Pietro Cicotti, Susan M. Mniszewski, Laura Carrington. 2014 Hardware and Software Co-Design for High Performance Scientific Computing (Co-HPC'14), held as part of SC14
- An exploratory co-design study: performance and efficiency observations from multi-threaded and event-driven implementations of CoMD. Invited talk. LLNL, December 2014.
- An Evaluation of OCR for a Classical MD Proxy Application, Laura Carrington, at SC14 OCR BOF.

Code

- NPB-CG (available from xtack-tg git repository and distributed with OCR)
- CoMD-OCR (available from xtack-tg git repository and distributed with OCR)
- CoMD-CnC (available from xtack-tg git repository and distributed with OCR)
- HPGMG-OCR (available from xtack-tg git repository and distributed with OCR)

3. Pacific Northwest National Labs Applications and Runtime Work

By Andres Marquez (PI), Joshua Landwehr, Joseph Manzano, Ellen Porter, Luke Rodriguez, Joshua Suetterlein

Challenge Focus

PNNL aims to tackle the challenge of moving data across the system while utilizing resources when operating at Exascale. The main challenge is to achieve an Exaflop of performance at or under 20 MegaWatts. This is quite a challenge, but is attainable when focusing on the main energy bottlenecks. One of the most important bottlenecks is the memory access and the management of the memory resources (which can be more than 50% of the total energy consumed).

Our approach aims to help to manage these resources at Exascale in a two prong strategy. One concentrates on static analysis techniques (under a polyhedral framework), which we call Group Locality, and the other uses dynamic information that will marshal data structures during runtime to adapt them (and their operators) to the ever evolving state of the machine. These latter techniques are called the Architected Composite Data Types. Both approaches are analyzed more in-depth in the next sections.

Technical Approach

Applications

NWCHEM Kernels

We identified two NWChem modules to serve as representative benchmarks for Exascale chemistry applications: 1) the Self-Consistent Field (SCF) method, and 2) the Coupled Cluster (CC) method. We have completed and released both codes to the community. Our release includes source codes, makefiles, input files, output files, and optimization suggestions.

The Self-Consistent Field method (SCF) is often the central and most time consuming computation in ab initio quantum chemistry methods. It is used to solve the electronic Schrodinger Equation, assuming that each particle of the system is subjected to the mean field created by all other particles. The solution to the Schrodinger Equation reduces to the following self-consistent eigenvalue problem

$$F_{\mu\upsilon} = h_{\mu\upsilon} + \frac{1}{2}\sum_{\omega\lambda}\left[(\mu\upsilon|\omega\lambda) - (\mu\omega|\upsilon\lambda)\right] D_{\omega\lambda} \qquad [1]$$

$$F_{\mu\upsilon}\, C_{k\upsilon} = \varepsilon\, S_{\mu\upsilon} C_{k\upsilon} \qquad [2]$$

$$D_{\mu\upsilon} = \sum_k C_{\mu k} C_{\upsilon k} \qquad [3]$$

where F is the Fock matrix, C are the eigenvectors of the system, ε are the eigenvalues, S is the electron force overlap matrix, D is the system density matrix, h are the one-electron forces, and the terms in the square brackets in Equation 1 are the two-electron Coulomb forces and the two-electron Exchange forces, respectively.

Figure 3.1 depicts the control flow of the released code. The upper two leftmost modules initialize the D and C matrices respectively, allowing the first iteration to compute Equations 1 and 2. The “Construct Fock Matrix”, “Compute Orbitals”, and “Compute Density Matrix” modules compute Equations 1, 2, and 3, respectively. The “Damp Density Matrix” module scales the density matrix and finds the greatest changed value between the current and previous density matrix. If the absolute value of the change is less than a threshold value, the method converges; otherwise, a new iteration is started. The method terminates after 30 iterations if convergence is not reached. Figure 3.2 gives the modules’ names, and array inputs and outputs.

The modules comprise simple rectangular, nested for-loops. For the most part, the loops are embarrassingly parallel, but the inclusion of reduction operations in some loops requires concurrent atomic updates. The most computationally intensive routine is “towel” that computes the two electron forces. Guards in the innermost loop impose a cutoff limit and reduce significantly the number of updates computed. The guards can be hoisted to reduce loop overhead as explained in the optimization file that accompanies the code release. We note that the h

values (one electron forces) and g values (two electron forces) used in computing the Fock matrix are constants, so they could be precomputed, saved, and reused. As explained in the optimization file released with the modules, symmetries in g can be exploited to minimize storage and computation requirements.

The second benchmark, the Coupled Cluster (CC) method, is interesting in that the code is generated automatically from tensor equations supplied by the user. Each equation is expressed as a function called from a driver that maneuvers a DAG of the data dependencies. The DAG, in conjunction with the unbalanced data parallel character of the tensor operations and data locality issues, presents a variety of scheduling alternatives and provides a rich environment for optimization and evaluation of emerging Exascale execution models. PNNL has provided versions of both codes in support of the XSTACK projects.

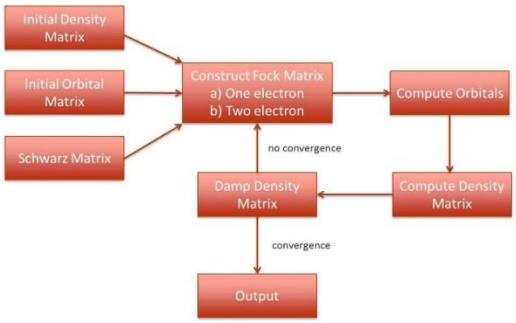


Figure 3.1 – SCF control flow

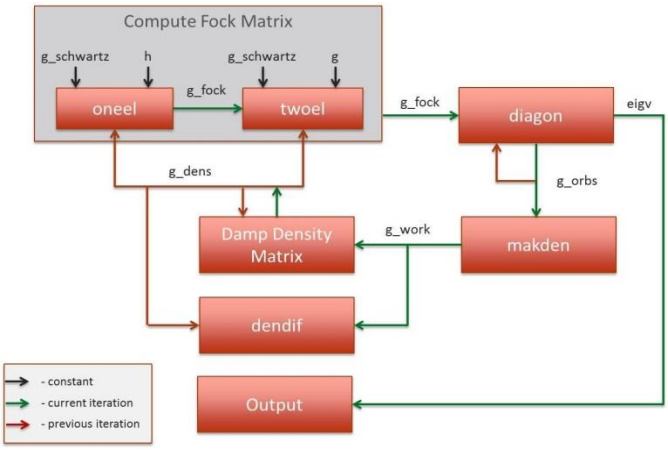


Figure 3.2 – SCF modules, inputs, and outputs

LULESH

LULESH is the first application PNNL has considered as a candidate for conversion according to the Concurrent Collections (CnC) programming model[1]. We chose CnC because it is one of the high level programming platforms that can be converted into the Open Community Runtime (OCR) framework[2]. The goal of CnC is to simplify parallelism expression by allowing developers to work at a higher level than traditional parallelization techniques permit. The process starts by building a diagram that graphically describes an application's methods, data structures, and execution. Once the graph is defined, the application development can begin.

About LULESH

LULESH is a code designed to approximate hydrodynamics equations discretely by partitioning the spatial problem domain into a collection of volumetric elements defined by a mesh[3]. The code has two distinct data structures: nodes and elements, see Figure 3.3. A node is a point on the mesh where the mesh lines intersect, whereas, an element is the volumetric space between nodes. For simplicity, we are testing with a regular Cartesian mesh. Every element has exactly eight neighboring nodes, and every node has eight neighboring elements, except when it is on a boundary.

[1] http://software.intel.com/en-us/articles/intel-concurrent-collections-for-cc
[2] https://01.org/open-community-runtime
[3] https://codesign.llnl.gov/lulesh.php

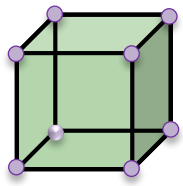


Figure 3.3 A single LULESH element and its neighboring nodes

PNNL worked closely with Intel to design a high-level representation of LULESH in CnC. We started with the control and dataflow diagram composed at one of the hackathons (Figure 3.4).

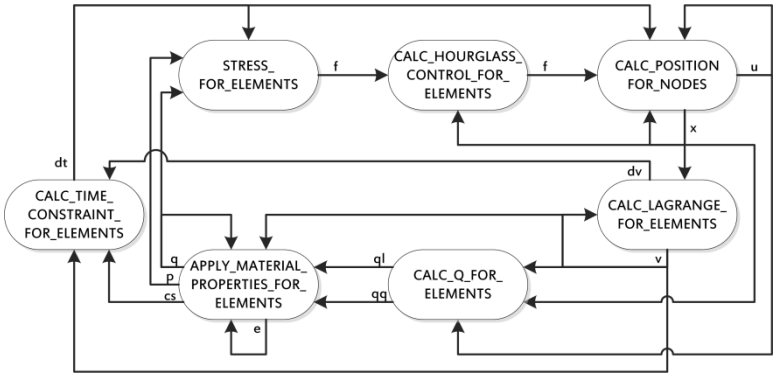


Figure 3.4 - LULESH original: shows data flow

We then enhanced that diagram in several ways (Figures 3.5 and 3.6).

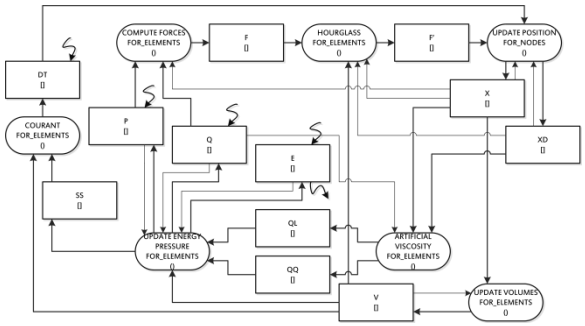


Figure 3.5 - Simplify data and computation

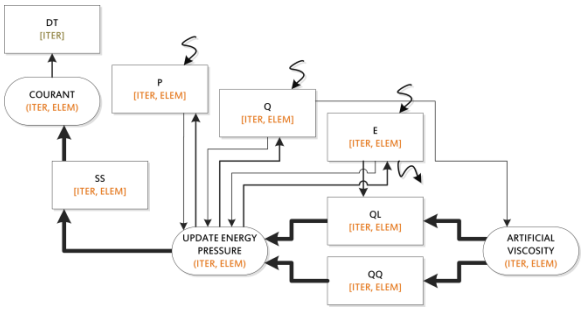


Figure 3.6 - Make reductions explicit

We inserted edges to show inputs and outputs from the environment, added iteration and mode tags (element or node) tags to each computation and data box, and we colored coded the box according to their mode (green for elements; purple for nodes). Finally, we simplified the diagram by showing explicitly which computations are reductions and return a scalar value.

As we worked through the CnC description, we developed a CnC version of the source code. We decided to start with LULESH 2.0 since it is the most complete and up-to-date version of the code. We used Intel's CnC implementation because it is written in C++ allowing us to replace methods in the LULESH code with the CnC equivalent one-at-a-time. The latter allows us to verify each step of the port and preserve correctness.

Conversion steps

The next step after creating the CnC diagram in Figure 3.4 was to convert the code. LULESH uses a global structure called the domain to hold all the data and uses pointers throughout the application to retrieve that data. From the diagram we can see that in the CnC paradigm we need to pass data from one CnC step to another directly. Thus,

the application required a bit of a redesign. For example, “compute forces” produces a force per node. This information previously would be stored in the appropriate node structure in the domain. In the CnC paradigm the produced force is stored in a data item which is then consumed directly by the “hourglass” step.

While altering the code to fit this new paradigm, we also removed loops that looped for all elements or all nodes. In the CnC version, each step works on either one element or one node. To improve efficiency, this could be tiled to work on several elements at a time or several nodes at a time. By breaking up each of the loops we gained more parallelism. Where before we would have to wait for all elements or all nodes to complete execution, now we are free to compute the next step as soon as its data is ready (either neighboring nodes or elements have finished computation).

We were also able to combine steps in order to reduce the number of gather-scatter operations needed whenever we switch iteration spaces (from nodes to element and from elements to node). Again, we can look at the compute forces and hourglass step as an example: Both of these steps iterate over all elements in the domain, and produce a force value for their neighboring nodes. Since the “hourglass” step does not use any of the data produced by “compute forces”, we can either do the two calculations in parallel, or combine them into one step. There were redundant calculations being done in the two steps, so we chose to combine them.

Updated code and diagram

We produced two different CnC implementations, one using Intel CnC and another using Rice University’s CnC-OCR. We chose to start with the Intel version because the libraries are well documented, include debugging tools, and are written in the same language as LULESH. As we converted the code, this allowed us to verify easily that we had not introduced bugs. This version of the code is complete and tested. The CnC-OCR version required more work, since we needed to convert LULESH from C++ to C. That conversion is complete as well. The CnC-OCR version converts the CnC code into OCR. This converted CnC code is also executable in FSim, a simulator of the architecture proposed by Intel for future Exascale machines. After completing the code conversion, we updated the CnC diagram to match any modifications we made (see Figure 3.7).

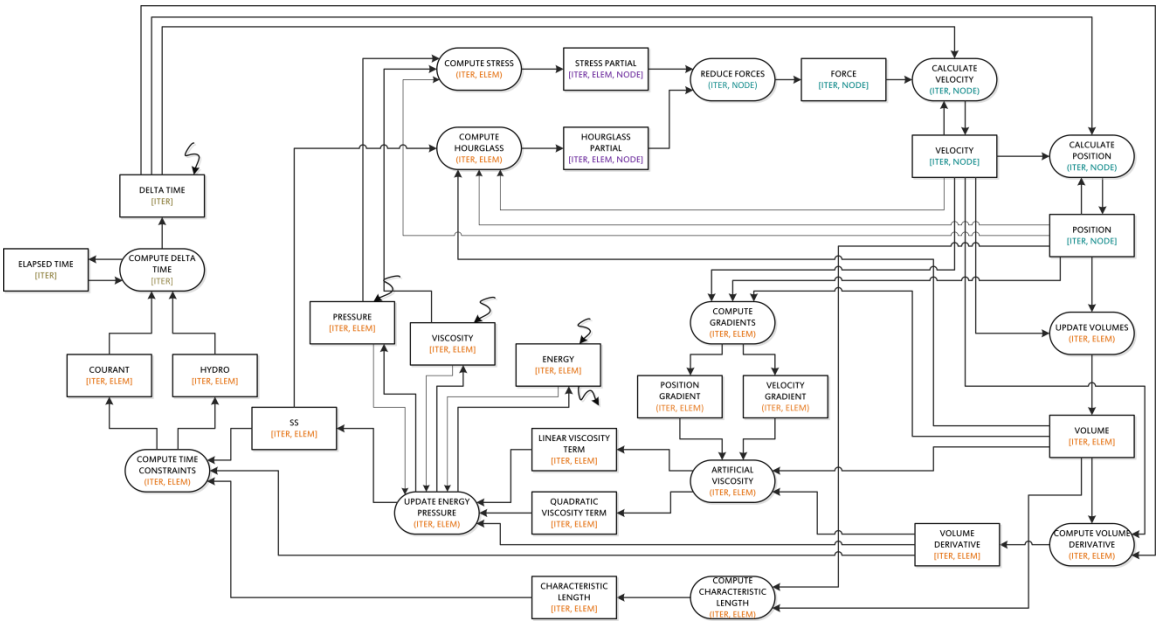


Figure 3.7: Final version of the LULESH CnC diagram. Color indicates iteration space, data items are rectangular, and computation steps are oval, thick lines show the main flow forward, thin lines show backward flow.

Three versions of Lulesh were pushed to the X-Stack repository: C, CnC-Intel, CnC-OCR. Lulesh in CnC-OCR form has successfully run on FSim with 1 and 2 element domain sizes. From these versions, PNNL integrated the per element CnC-OCR version of LULESH into the regression test suite of the OCR public repository. This is the first non-kernel app being hooked in and will provide the OCR team valuable results moving forward.

PNNL's version of OCR has successfully run the per element CnC-OCR version of LULESH, and has helped to point out areas to improve the runtime. The code identified bottlenecks in memory allocation, scaling for large events, and performance of events.

For the CnC OCR LULESH version, the CnC-OCR translator is a moving target as it develops, advancing in features and functionality. In order to keep LULESH running smoothly and to take advantage of the latest and greatest the tool has to offer, we updated our code to be compliant with the latest version of the translator. The updated version has been pushed into the X-Stack repository and is available in the apps directory.

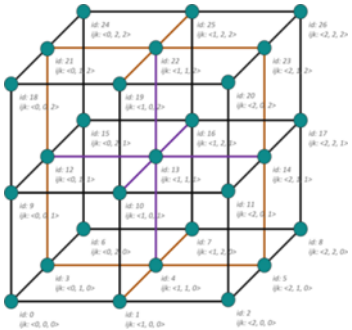
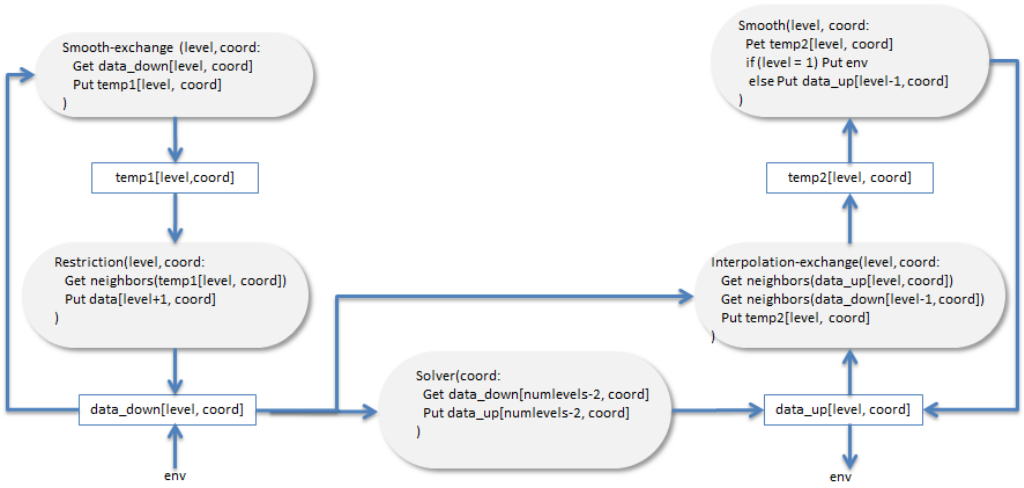


Figure 3.8: LULESH node id and ijk

To help support the auto-tiling research, effort was also made to refactor the indexing in LULESH. LULESH has two main data structures, elements and nodes. Both structures are currently indexed using an integer in the form of a node id and element id. From these ids, neighboring ids can either be calculated or looked up in a global lookup table. Both options come at an additional cost. One way to reduce this cost is to store and reference the nodes and elements using a three dimensional i, j, k tag (see Figure 3.8). Calculating neighbors then becomes an easy task of adding or subtracting one in any of the dimensions. This increases the dimensionality of the tag components and requires a restructure of many of the LULESH data structures. The CnC graph representing LULESH was updated to accommodate the additional information in the tags.

MiniGmG

Although PNNL completed a preliminary MiniGmG CnC graph description (presented below), this application was superseded by HPGMG. We present this initial graph for completeness.



HPGMG:

PNNL's HPGMG implementation is a follows: The first step was to build the HPGMG control in CnC and implement its different cycles (v, f and w). Under the CnC programming model, keeping the cycle code separate from the step code becomes trivial. Thus allowing us to reuse the steps regardless of which cycle is running. The second step was then to implement and test this theory. The final step was to integrate CnC control into the existing HPGMG code.

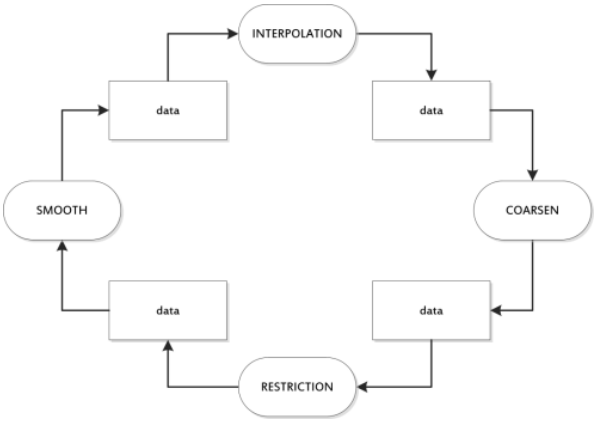


Figure 3.9: The application flow for the HPGMG mini app

Adapting any complex code base such as this one to a different paradigm takes careful design and planning to ensure that we do not introduce avoidable artifacts that can prove costly in performance. To this end, the PNNL team has extensively mapped the code flow of HPGMG and the data structures on which it relies. The high-level processes and data dependencies were then refactored to fit more closely with the OCR model of EDTs (event driven tasks). From this, we developed a CNC graph file (see Figure 3.10) for HPGMG that fits the CNC-OCR translator.

The next step in completing the code conversion was to refactor HPGMG's internal data structures (see Figure 3.11). The approach we took included building tiling as a parameter rather than a thread-dependent value. This will help facilitate robust testing of the code over various scales, but comes at the cost of a more involved design

and implementation refactor. Our use of the EDT model also allows for complete parallelization down to the solve step. Hence, data dependency of the solver at the coarsest level is the only bottleneck in the code.

Once complete, HPGMG's refactored data structures were used to create a serial C baseline version of the code. This serves as both a stepping stone to the CnC version of the code as well as a useful comparison tool for benchmarking. The serial c baseline was then merged into the previously completed CnC outline. The resulting CnC version of HPGMG is currently awaiting support from OCR for math libraries as well as minor bug support from the CnC translator. Both versions have been submitted to the X-Stack repository.

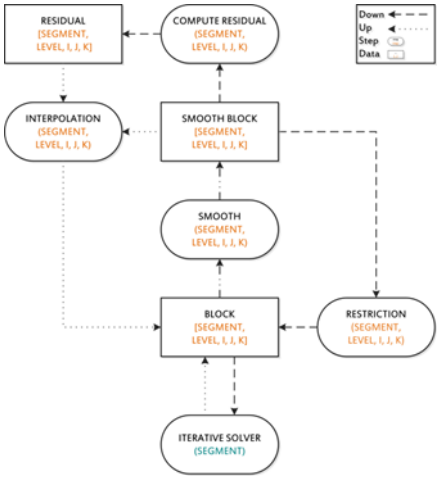


Figure 3.10: HPGMG CnC Graph

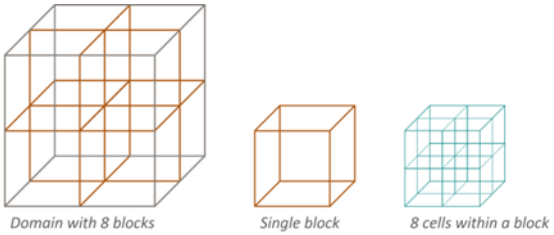


Figure 3.11: HPGMG domain structure

Architected Composite Data Types

As part of our efforts focusing on utilizing Exascale systems, we built the underlying infrastructure that will guide us in the design of the Power Efficient Data Abstraction Layer (PEDAL) for Rescinded Primitive Data Type Access (RPDTA). This infrastructure consists of two major efforts: Group Locality and Architected Composite Data Types which aim to exploit locality and memory access patterns at both the compiler and runtime levels.

Group Locality

In order to observe and control the association between data structures and sets of threads, we are developing a fundamental framework called "Group Locality" (GL). The framework helps us to optimize data placement, movement and data lifetime, taking into consideration the interaction of multiple threads. These three criteria are necessary (but not sufficient) to determine suitability of data manipulation (e.g., compression), as would be required for PEDAL.

Since balancing parallelism and locality is crucial for high performance, we work to improve locality not on a per thread basis, but for a set of threads working together as a group. Our approach is to identify multiple threads that can run concurrently and then compute a memory mapping that improves locality for the group of threads. We have developed a framework for a fine-grained parallelism where threads work together performing a micro dataflow execution. Threads are aware of the implicit low overhead execution and dependency pattern.

Architected Composite Data Types (ACDT)

The Architected Composite Data Types (ACDT) framework explores opportunities to restore operator symmetry between primitives and composites where composites represent application meaningful structures (matrixes, vectors, trees, etc), and primitives are the architectural memory operations (load, stores, increment, etc). The framework is made aware of data composites, assigning them a specific layout, transformations and operators.

The framework can cast an appropriate type for a data composite and determine the techniques that can be applied to it. With this bottom-up approach, the data manipulation overhead is amortized over a larger number of elements and its performance and power efficiency is increased.

In one of our first studies, we developed an ACDT framework on a massively multithreaded runtime system geared towards Exascale systems. We exercised the conceptual framework with two representative processing kernels: Matrix Vector Multiply and the Cholesky Decomposition applied to sparse matrices. The sparse matrices used for the experiment were obtained from the University of Florida database and represent real world use cases. We opted for compress/decompress engines as transformations. We developed two different approaches based on the awareness of the transformation with respect to the application. Both approaches show advantages. However, a certain degree of HW help is needed for the application agnostic approach. In the case that the application is aware of the ACDT, optimization opportunities can be exploited, like algebraic invariant operators by the application. Thanks to these operators, we have observed performance and power improvements that are an order of magnitude better than implementations that are ACDT oblivious. An early exploration of the effects of using compressed data with matrix operations was published in ADAPT'14 and more details can be found in the ICPADS'14 submissions.

In support of ACDT, we have developed (from scratch) a version of OCR, called P-OCR. This version allows us to insert and modify key overheads that might prevent the efficient integration of the ACDT framework inside OCR. We put special interest on the distributed version of P-OCR so it can achieve scalable results in large clusters. The distributed version of P-OCR has the following features:

- Support for Infiniband via the rsocket protocol
- Support for resource distribution hints
- Redesign of the GUID allocator
- Implementation of a memory coherence protocol for distributed OCR that is almost compliant with the OCR spec v.99a
- Improved support for large clusters using the SLURM scheduler
- Inclusion of several code optimizations
- Test of the framework running on 128 nodes (4096 cores) with two selected kernels

As of this report, P-OCR runs correctly all the application code available in the community OCR repository on shared memory including codes translated with/from R-Stream and CNC and fully supports OCR's memory model. However, at the moment, it does not support satisfying slots, data block allocators, or latch events. In addition, the exclusive writes are treated as an exclusive lock instead of a reader/writer lock. Reader/writer locks reduce the performance for most of the applications. For distributed P-OCR, the memory coherence protocol currently does not support writing to the same data block at the same time or reading/writing from/to the same data block at the same time. Merging of data blocks via writing to the same data block will require a less efficient protocol, which will be supported in the future for completeness. In addition, P-OCR introduced some new features not in the specification: parallel loop extensions, pure dataflow tokens (sending a copy of memory without data blocks to events/EDTs limited to 64 bits, and various distribution hints such as round robin, even partitioning, chunks, etc for EDTs, data blocks, and events.

Our P-OCR framework was initially based on TCP sockets between nodes and shared memory within a node. However TCP scalability is limited, so we opted to support a streaming protocol that is written on top of Infiniband's RDMA protocol to help achieve better scalability. The protocol chosen, rsockets, is developed by Intel and is part of the OpenFabrics' librdmacm.

We began working towards providing memory movement hints for kernels/applications that spawn work predominately on a single node. These hints allow P-OCR internally to create memory on the current node, allocate a GUID for a different node that will point to that memory in the future, and redistribute that memory to

the other node while the current process continues to execute. In the future, P-OCR will redistribute memory across nodes without user help. In support of this, we created a new GUID allocator/address space that is decentralized where every node can allocate a GUID that points to any other node.

In our P-OCR framework, a GUID stores destination node information. The destination node has a routing table pinpointing the owner of that data block. Initially, it will always point to itself. During program execution, data block ownership can change during write operations. Data block ownership discovery during execution is hereby more efficient as compared to an overall network discovery.

Moreover, because data blocks can be written at any time by any node, we needed to develop a memory consistency protocol. This protocol supports OCR's acquire/release memory consistency. The difference between P-OCR's memory model and "standard" release consistency is that the release semantics are operating on data blocks and those data blocks can only be acquired once per EDT internally by P-OCR before that EDT begins running. Releases can happen at any point, once an EDT begins executing. So for our protocol, a data block is copied to a node when an EDT requires it as an input. The data block copy exists until it is invalidated. To invalidate a data block, the data block needs to be released or be in a Read Write (RW) or Exclusive Write (EW) mode. If a data block is in those modes, we send an invalidate request to the data block router who in turn sends the requests to anyone who has copies. The router identifies the new owner as the node that sent the invalidate request and asynchronously waits for acknowledgements (ACK) from whoever has currently a copy. Once the router receives all the ACKs, it acknowledges the initial requester. Meanwhile, the requester continues executing its EDT. If it encounters any event satisfies during an ACK absence, the requester will buffer those calls and continue executing asynchronously. Once, the ACK comes, it will execute those satisfies. Similar semantics can happen on a release, but release functionality has not been implemented yet.

Results and Analysis

Group Locality (GL)

Figure 3.12, shows the execution time measured for the stencil in Example 1 using different number of thread groups against code generated by PLUTO. Similarly, Figure 3.13 shows memory served by shared L2 caches for Intel Xeon Phi. Such sharing between caches can reduce strain on limited bandwidth. The advantages we have seen so far are based on increased parallelism and accidental sharing of data between threads. We believe that we can improve on this by having a memory mapping that is not only palatable to caches, but also is less constraining for memory banks and pages. We are currently working on finding a solution for this problem.

```
for (int i = 1; i <= n ; i ++){
  for (int j = 1; j <= n ; j ++) {
     A[ i ] [ j ] = A[ i -1][ j ] + A[ i ] [ j -1] + A[i][j];
  }
}
```

Example 1: Stencil Loop

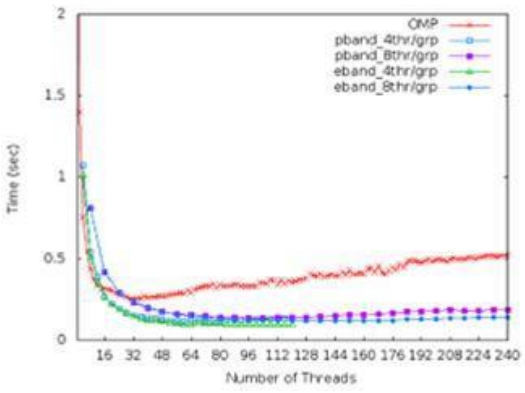


Figure 3.12 - Execution time for different thread group against PLUTO generated OpenMP

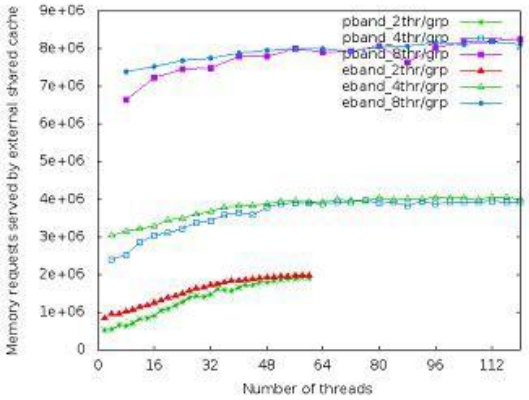


Figure 3.13 - Memory request served by external shared L2 caches

Using Group Locality techniques, we selected and applied them to three example Mini-Apps from the Mantevo Suite [https://mantevo.org]. This effort aimed to showcase the advantages and short comings of the Group Locality framework. The Group Locality framework provides a highly parallelized tiling strategy with intra-tile parallelism and a very fine grain runtime system based on micro data flow and a restructuring data space for the group of threads working together.

The submitted Mini-App report discusses the parallel tiling strategy plus the fine grain runtime system when they are applied to the three selected mini apps. The first mini app is called MiniSMAC-2D, and it is used to solve finite-difference 2D incompressible Navier-Strokes equations. We believed, Group Locality could take advantage of the symmetric Gauss-Seidel relaxation kernel. However upon further analysis, we found that the iteration space cannot be tiled under our framework. This example highlights the cases where our tiling strategy cannot be applied.

The next mini app, called TeaLeaf, is used to solve a heat conduction equation. It uses a 5 point stencil solver that our framework can exploit. Using both, our fine grain runtime and the "Jagged Tiling" framework, we achieved an increase of 30% over state-of-the-art polyhedral generated code for an Intel Phi Architecture.

The final kernel is called Mini AMR, and it does an adaptive mesh refinement. It is composed of an extreme computational heavy kernel. After applying our "Jagged Tiling" strategy, we found out that there was a degradation compared to the state-of-the-art code of around 11%. The memory profile of the application shows that the tiling methodology should have won in these aspects (i.e. lower memory trips, lower cache misses, etc). Thus, this kernel will require more study.

The report is available in the XSTACK wiki page.

ACDT

As part of the XSTACK Traleika Project, PNNL built a prototype of a dynamic self-aware lightweight system[4], the Architected Composite Data Type Framework (ACDTF) – originally called the Rescinded Primitive Data Type (RPDT) -- running on top (at first) of the Open Community Runtime (OCR) interface (ACDTF-OCR). This prototype features real time decision-based compression and algebraic invariant operations that use real-time sampling. The prototype has been implemented across a Linux based cluster and a shared memory node. The prototype supports

[4] A. Marquez, J. Manzano, S. Song, B. Meister, S. Shrestha, T. St. John and G. R. Gao. "ACDT: Architected Composite Data Types Trading-in Unfettered Data Access for Improved Execution." In the 20th IEEE International Conference on Parallel and Distributed Systems, Hsinchu, Taiwan, December 16 – 19, 2014

two compression algorithms (The Floating Point Compression and an in-situ Run Length Encoding)[5], an enhanced compressed format (based on the sparse compressed block format) and two sampling methods (based on block and differential run-length sparseness).

In addition, the prototype offers wrappers to the OCR's interfaces and expands upon them. Governors that make the decisions to compress, sample, and actually do compression are configurable to run on dedicated cores, at OCR interface points, or across all cores. Furthermore, these governors can run asynchronously or synchronously. Thanks to these features, we can do exploration studies on the best prototype configurations across various platforms.

As part of our experiments, we have concentrated on the Cholesky kernel and characterized its performance and its ACDTF overhead on community OCR v.099a.

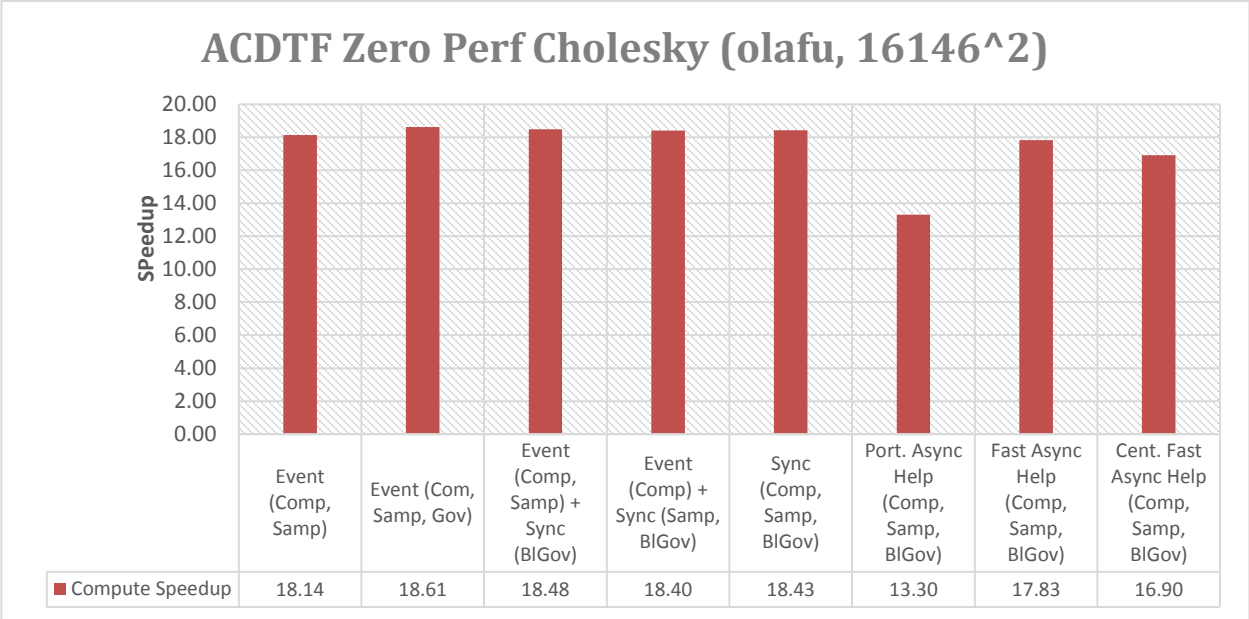


Figure 3.14: ACDFT-OCR running on community OCR v0.99a shows sparse matrix compression improvements for "olafu" at event points across its different configurations -- synchronous vs. (fast-) asynchronous, sampling vs. non-sampling, various governor strategies -. (Comp = Compression, Samp = Sampling, Gov = Governor). Furthermore, the asynchronous version has a portable implementation that works on any version of OCR, and a fast version which requires pointers for the platform to be able to address any memory location. ACDTF supports a centralized core to handle requests (Cent). Block size is ~300.

Preliminary results show that we are able to achieve performance rates between 0.99x to 18x vs. non-ACDT-enhanced OCR across various sparse matrices using a 16 core Xeon system, regardless of the configuration of our system, as long as we detect invariant operations and use our compressed zero-configuration. All others configurations show no improvement or less than 1% performance degradation due to overhead being low for the coarseness of the work (~300^2 block size).

[5] Peter Lindstrom and Martin Isenburg. 2006. Fast and Efficient Compression of Floating-Point Data. *IEEE Transactions on Visualization and Computer Graphics* 12, 5 (September 2006), 1245-1250. DOI=10.1109/TVCG.2006.143 http://dx.doi.org/10.1109/TVCG.2006.143

P-OCR also provides several runtime hooks that make efficient integration of ACDTF possible. The objectives were to allow PNNL to improve OCR's fine-grain performance on shared memory systems, to enhance OCR with ACDTF, and to develop a scalable distributed version of OCR as our ACDT testbed. These objectives are well under way and have produced very promising results.

Our P-OCR system managed to substantially reduce the overhead vs. a version of community OCR (v0.99), see Figure 3.15. Overhead is especially problematic for very fine-grain tasks. To achieve these performance improvements, PNNL used custom designed synchronization algorithms and data structures, amortized the cost of growing structures, aligned data to the cache lines, attempted to reduce false sharing where possible, and developed a custom thread-caching pool-based malloc to reduce hits on the standard allocation table locks for data under 4 kilobytes.

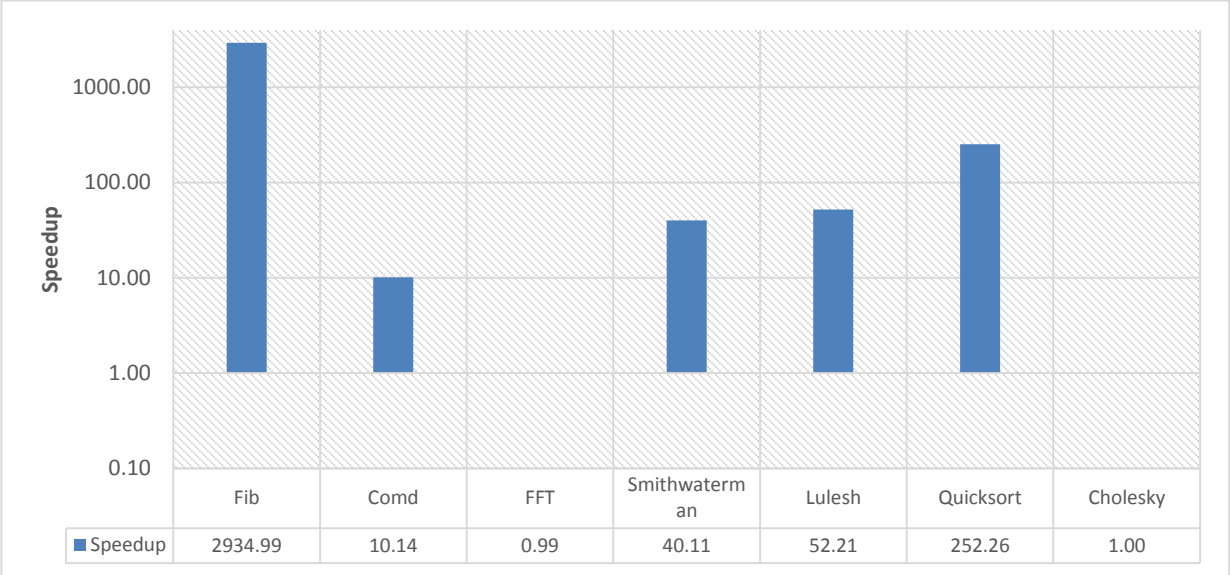


Figure 3.15: PNNL's SHMEM OCR performance vs the Community OCR v0.99a, running on 16 cores (gcc 4.8.3 –O2 –mtune=native, Intel Xeon CPU E5-2690) . Fib: 30, Comd: 12^3, FFT: 2^20, LULESH Domain: 4, Quicksort: 2000 elements, Cholesky: 16K^2, Smith-Waterman: 800

For ACDTF, integrating it into P-OCR allowed many of the overheads to be removed because they were implemented in a manner redundant to the underlying OCR. Additionally, a new design could be done using underlying structures implemented for P-OCR. This included storing the global unique identifiers (GUIDs) of data in a concurrent hash map to represent the ACDT state. This reduced the redundant compression requests and state significantly. Figure 3.16 shows the current performance of ACDTF on 1 node. The performance is 11.77X faster than the ACDT running on top of the Community OCR v0.99a.

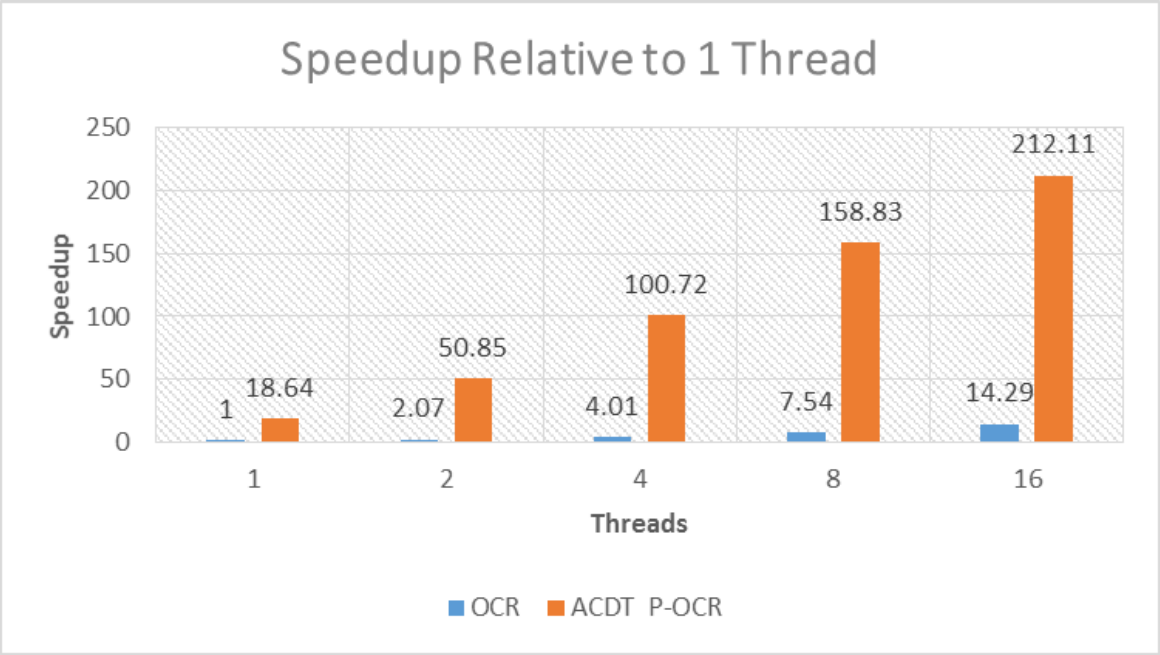


Figure 3.16: ACDTF samples the data of each data block asynchronously and compresses the block of 16^2 matrix. Performance is relative to 1 OCR Thread from the Community OCR (v0.99a)

These enhancements to the P-OCR testbed allowed us to achieve good scalability on Cholesky and Smith-Waterman kernels as shown in Figures 3.17-3.19 for distributed OCR and scalable results for the shared memory kernels as presented in Figure 3.20. For the Cholesky kernel, the main EDT creates a static computation graph and partitions the matrix into data blocks. We use hints to redistribute the memory for these EDTs, events, and data blocks across nodes. These initial distribution movements are accounted for in our timing for the benchmarks below: thus these times are not strictly computation time. For each benchmark, we used a 4 socket 32 core AMD Opteron 6272, 32 threads per node, 64 GBs of memory per node, Infiniband, rsockets, GCC 4.8.2, the SLURM launcher, and a 400^2 tile size. The current data presented here is preliminary. We are still in the development process, adding new features and testing all parts of the testbed. The results presented here are a good indication of the progress of the testbed, but they require deeper analysis and experimentation.

As a strong scaling test, Figure 3.17 shows Cholesky performance with a fixed sized matrix of 80000^2 as node count increases. For each node, we used 32 cores. The max relative speedup we were able to achieve was 74X on 128 nodes for the Cholesky kernel and 94X for the Smith-Waterman kernel. Likely, the performance drop-off from 64 to 128 nodes is due to a combination of processors starving for data and the communication overhead. Figure 3.18 shows a weak scaling test where we fixed the computation per node to be the same as we scaled the nodes. The graph shows very close to ideal scalability up to 64 nodes. The sudden decrease in scalability from 64 to 128 nodes is likely due to the fact that we are timing the initial memory distribution of the 160K^2 matrix movement as well as other memory movement. The graph of the static OCR Cholesky can have a large memory footprint. Case in point, the matrix alone is 204GBs of memory, not including the EDT or event memory. Our initial assumptions lead us to believe that Infiniband saturation and congestion cause the startup of EDTs to be delayed. Finally, Figure 3.19 just computes the gigaflops based on Figure 3.18's timing information. The current numbers reflects a Cholesky kernel that has not used any optimized libraries (e.g. MKL) for its internal operations, which will increase the performance further. Nonetheless, as of now, we have significantly improved OCR's performance for distributed systems.

Figure 3.20 shows the strong scalability of P-OCR in a single node environment. Since the tested node only has 16 FPUs, the scalability drops significantly when going from 16 to 32. However, the kernels shows good scalability up

to 16 (from 8X to 15X) and some even show scalability passed 16 (with better cache locality being the most likely culprit).

Thanks to all these enhancements and features, we have vastly improved distributed OCR compared to our initial implementation.

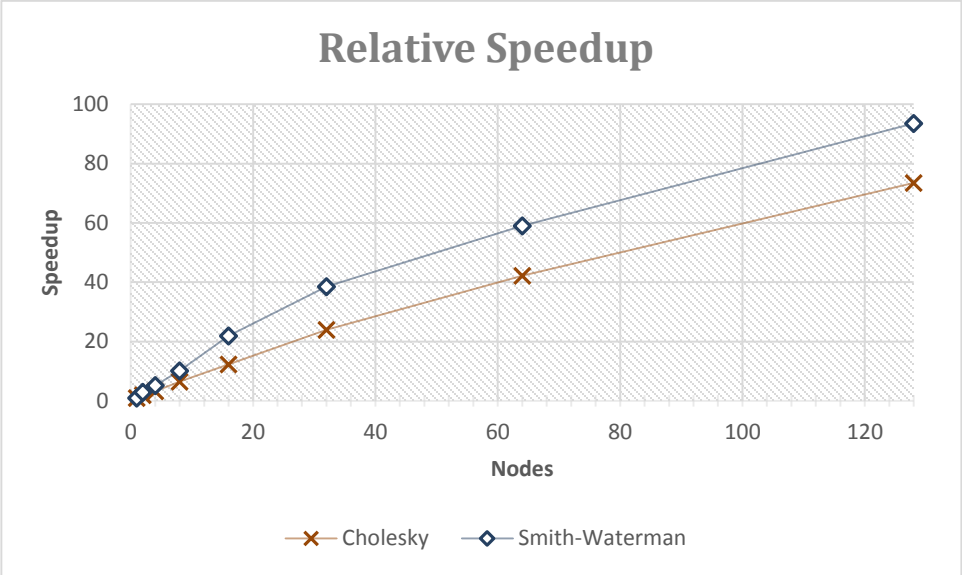


Figure 3.17: The machine has 32 Cores per node for a total of 4096 cores: used 32 threads per node and tile size = 400^2 for the Cholesky kernel. For the Smith-Waterman kernel uses a 8 MiB nucleotide string for its calculation

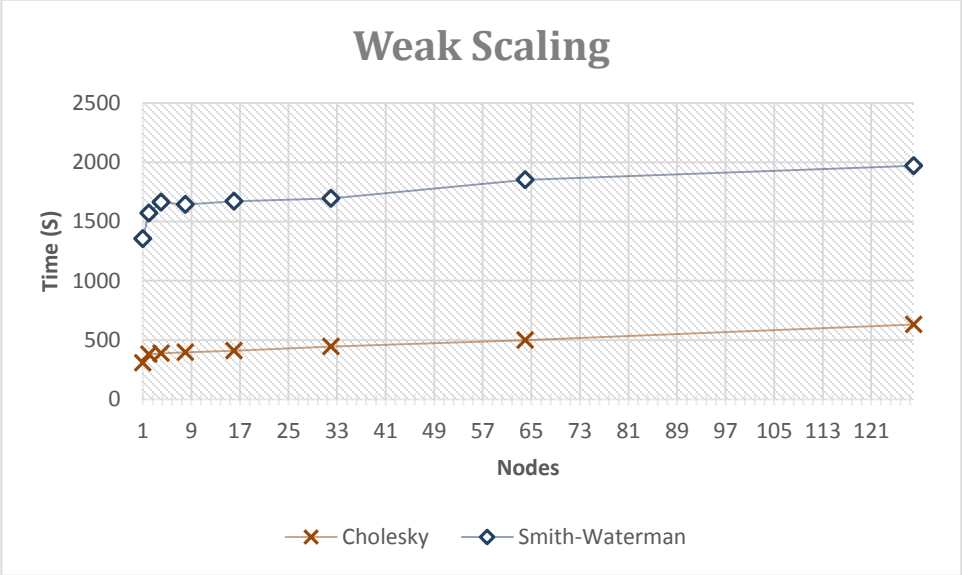


Figure 3.18: Used 32 threads per node and tile size = 400^2; Amount of floating point operations per node = (160000^3/3)/128. The Smith-Waterman kernel use 1 MiB to 12 MiB strings for each nodes

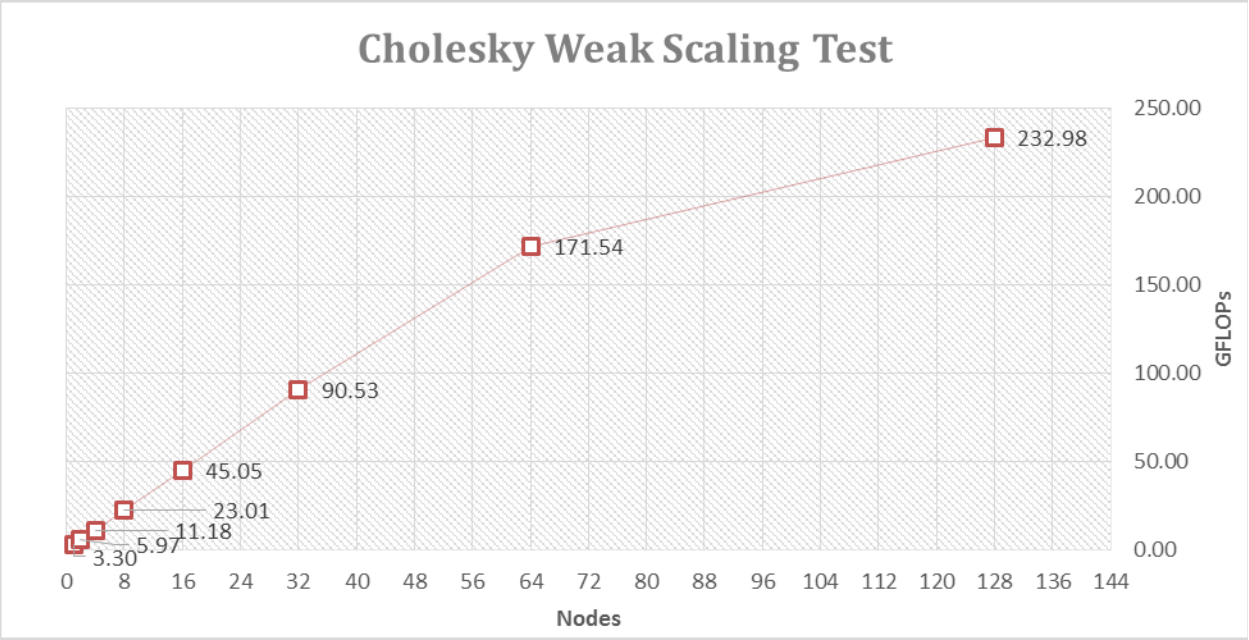


Figure 3.19: Used 32 threads per node and tile size = 400^2; Amount of floating point operations per node = (160000^3/3)/128. See Figure 3.2 for matrix size per node and core count.

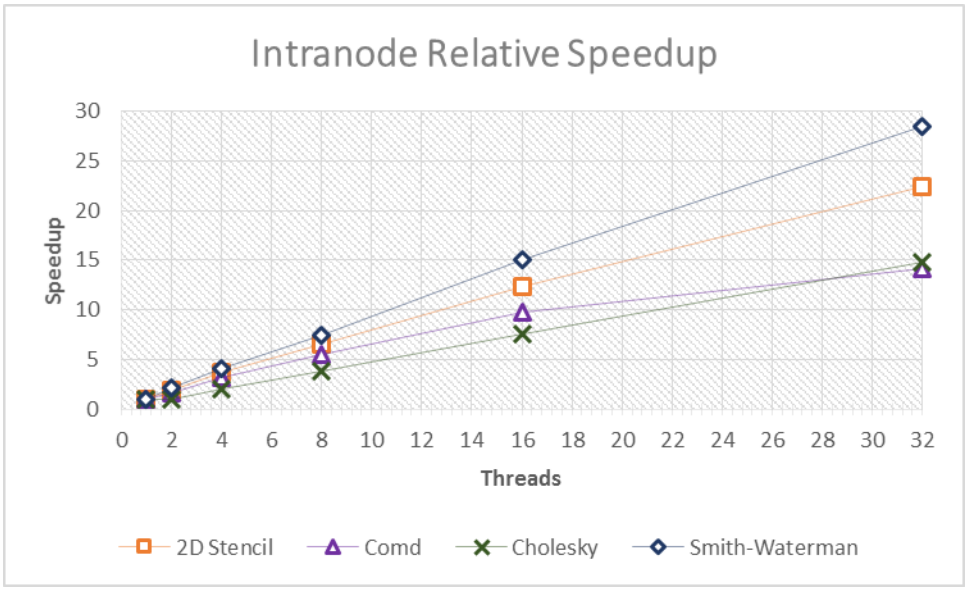


Figure 3.20: Strong Scalability in a single node for selected kernels

Recommendations and Future Work

In the future, we would like to integrate the group locality concept with the ACDT framework to provide a complete solution. However, this is outside the scope of this project. We are planning in the future more ACDT data types which will help the performance of the Exascale application codes. We are integrating the ACDT features on the P-OCR and to characterize different applications running on the ACDTF.

More support for applications and runtimes, this includes performance and debugging tools, software toolchains, among others.

Products and Bibliography

"ASAFESSS: A Scheduler-driven Adaptive Framework for Extreme Scale Software Stacks," T. John, B. Meister, A. Marquez, J. Manzano, G. Gao, and Xiaoming Li
In Proceedings of the 4th International Workshop on Adaptive Self-Tuning Computing Systems (ADAPT'14); 9th International Conference on High-Performance and Embedded Architectures and Compilers (HiPEAC'14), Vienna, Austria. January 20-22, 2014. Best Paper Award

"ACDT: Architected Composite Data Types Trading-in Unfettered Data Access for Improved Execution," Marquez, A. et.al, ICPADS 2014

"Application Characterization at Scale: Lessons Learned from Developing a distributed Runtime System for HPC," J. Landwehr, J. Suetterlein, A. Marquez, J. Manzano, G. Gao, submitted to Twenty First International Conference on Architectural Support for Programming Languages and Operating Systems. April 2-6, 2016, Atlanta, Georgia, USA

4. Intel Applications and Kernels

by David S. Scott, Intel

Challenge Areas

The effort by the Intel Applications team focused on the following challenge areas: Extract parallelism from applications and Energy efficiency and scalability. In all cases, the applications and kernels were coded directly in OCR. This reports on Lulesh, Stencil1D, and Stencil2D.

Lulesh as a study in translating legacy code into the OCR EDT and Datablock model

The original Lulesh 1.1 translation experiment focused on developing strategies to map Lulesh's C++ functions and OpenMP parallel loops into OCR EDTs. Lulesh-OMP enabled focusing on the EDTs by having just one C++ Object. The various data structures, scalars and arrays, in the object were mapped onto a single data-block. The single data-block also acted as a small heap, as all of the malloc()'ed arrays had their space reserved in this single data-block. The C++ functions were mapped to C functions with function signatures that matched OCR EDTs:

```
ocrGuid_t edt_name( int paramc, int64_t *ParamV, int depc, ocrGuid_t DepV)
```

There were some short cuts made in the original translation. The code assumed that data-blocks would not get moved by the run-time, which simplified the emulation of malloc(). If the C++ object's data-block was relocated there were a number of pointers within the data-block which would have had to have their address modified. Also, as a distributed memory version of OCR was developed, a single large data-block did not make it possible for the runtime to efficiently move EDTs to other nodes in a cluster. Finally the translation was done with the knowledge that the whole program was known so there was no effort to develop a consistent calling convention.

The more recent translation strategy effort wanted to address the short comings of the original effort, accommodate changes to the OCR specification, like the removal of ocrDbAquire(), and better data-block support. In particular there was a goal to try and develop the beginnings of an OCR ABI, which would allow a language translator, e.g. the Rose compiler to be built, that would translate applications and standard libraries to the OCR runtime.

The following solutions developed during this effort. (1) Addresses became a GUID:offset so that the data-block could be relocated between uses by an EDT. The native C address was rebuilt by the EDT entry code by combining

the base address of the data-block from the GUID with the offset into the data-block to produce a native C pointer that was valid only within the current EDT. Special consideration was made for global addresses, like function pointers and C external data, all global addresses use the reserved GUID zero and the offset was their actual address. (2) The standard stack structure was modified to be a linked list of stack frame data-blocks, which better enabled distributed memory execution of codelets, as the local stack frame could be a local data-block. Local variables were declared as a member of the stack frame data structure. (3) A standard calling convention was developed with scalar and the offset part of addresses placed in the "ParamV" data structure and the "GUID" part of addresses placed in "DepV".

The linked list of stack frames also enabled the "Stack" to be used as a continuation. This enabled a relatively simple strategy to support complicated data structures, like link the edge vectors in C arrays and linked lists, without needing ocrDbAquire(). When a non-argument address (GUID:offset pair) need to be dereferenced as a native C pointer , all that needs to be done is clone a copy of the EDT with an additional argument. The cloning function then appended to its ParamV, the address' offset, and appended to its DepV, the address' GUID. The cloning function then schedules the cloned function to run using its current stack frame, and returns. The OCR run-time will later schedule the cloned function to run. The cloned EDT will have the cloner's state and the information to be able to construct a native C pointer to the previously inaccessible data-block.

Finally a strategy for OpenMP like parallel loops was developed using a similar cloning strategy, the parallel loop body became an EDT via cloning the function that contained the loop, and invoking the cloned loop body with the current stack frame structure for the shared variables and a separate data-block for the private variables and a RAJA like iterator.

This new translation study was run on smaller example than Lulesh, (Unbalanced Tree Search – UTS 1.1) to more easily show that the techniques were viable and C/OpenMP like constructs could be mechanically translated to OCR's EDTs and data-blocks. It is believed that a translation tool like Rose could be used to constructed to use these techniques to translate legacy C like languages such that the basic blocks in the code became EDTs and that OpenMP (or Cilk) like parallels constructs could be handled. There is some concern that creating the linked list of stack frame data-blocks will require development of some optimization strategies. As an example Cilk spent a lot of effort optimizing the implementation of its cactus stack.

Here is an example of how the standard C main function would get translated to this proposed OCR ABI:

```
/*
 * int main(int argc, char **argv);
 */

#define __WORDS_ocr_main_args_t
((int)((sizeof(__continuation_args_t)+sizeof(int)+sizeof(size_t)+sizeof(uint6
4_t)-1)/sizeof(uint64_t)))
typedef union __ocr_main_args_t {
  struct { __continuation_args_t std; int argc; size_t argv; } vars;
  uint64_t paramv[__WORDS_ocr_main_args_t] ;
} __ocr_main_args_t;

#define __WORDS_ocr_main_guids_t
((int)((3*sizeof(ocrGuid_t)+sizeof(ocrGuid_t)-1)/sizeof(ocrGuid_t)))
typedef union __ocr_main_guids_t {
  struct { ocrGuid_t    new_frame, old_frame, argv; } selector;
  struct { ocrEdtDep_t  new_frame, old_frame, argv; } segment;
  ocrGuid_t  guids[__WORDS_ocr_main_guids_t];
  ocrEdtDep_t depv[__WORDS_ocr_main_guids_t];
} __ocr_main_guids_t;
```

```
ocrGuid_t __ocr_main( uint32_t __paramc, __ocr_main_args_t  *__paramv,
                      uint32_t __depc,   __ocr_main_guids_t *__depv );
ocrPtr_t  __template_ocr_main = { .offset = (size_t)ocr_main, .guid =
NULL_GUID };
```

The CnC Linux version of Lulesh 2.0 was enhanced to include 3-D tiling of the main "node" and "element" data structures. Experiments were run with various tile sizes and for the optimal tile sizes performance of the CnC version was comparable to the original Open/MP version of Lulish 2.0.

Stencil 1D

ocr: sandbox/shared/tgSchedulerAug2015 commit ef244f0f136718f65df769559aab14aecead0431
fsim: master commit: 70886b0fdd73fd29fa00c8f61d9a1ba62f97ff63

Stencil1D is an OCR code that implements a one dimensional stencil calculation over M*N data points. The initial values are all zeros except for the two endpoints which are one. At each time step, the updated values are computed as:

$$a_{new}[i] = {}^{1}/_{2}\, a[i] + {}^{1}/_{4}\,(a[i-1] + a[i+1])$$

Each grid point converges monotonically from 0 to 1 with the outer values converging faster.

When the computation is spread across multiple EDTs, it is necessary to exchange local boundary values with the two neighboring EDTs at each iteration. Since EDTs cannot receive new data once they have started, each EDT sends the boundary values by satisfying events with small datablocks and then launches a clone with two dependences (which are satisfied by the neighbors). In particular, the code has three input values, N the number of worker EDTs, M the number of datapoints on each worker, and T the number of iterations.

Three scaling studies were performed. In all cases energy and time are reported relative to the left most entry. Three runs were made and the average values were reported. All runs were 10 time steps. In the first study 8 worker EDTs were run on the 8 XEs of a single block while M was varied from 1 to 16384. As shown in Figure 4.1, for most values of M the compute time was less than the overhead time so most of the runs took about the same amount of time and energy. It was only the last three values that show time and energy growing approximately linearly with M. The second study shows strong scaling starting with a problem size of 262144 and spreading it over more and more worker EDTs. There was no over prescription so the number of XEs use equaled N the number of workers, so that N=16 was run on two blocks, N=32 was run on four blocks, and N=64 was run on eight blocks. As shown in Figure 4.2, time dropped and then remained fairly constant while energy was fairly constant until the communication between multiple blocks drove it up. In the final study a fixed problem size with a fixed number of workers was run on more and more blocks. As shown in Figure 4.3, time dropped as more compute resources became available until there was only one worker EDT per XE. With no over prescription, an XE became idle until the data for its next iteration has arrived. Energy use grew larger as communication became required between blocks.

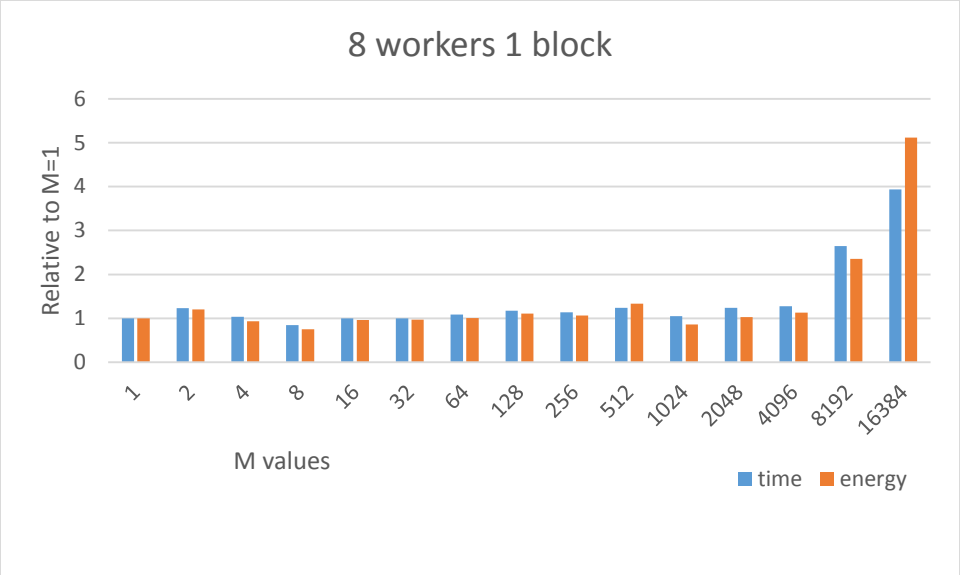


Figure 4.1

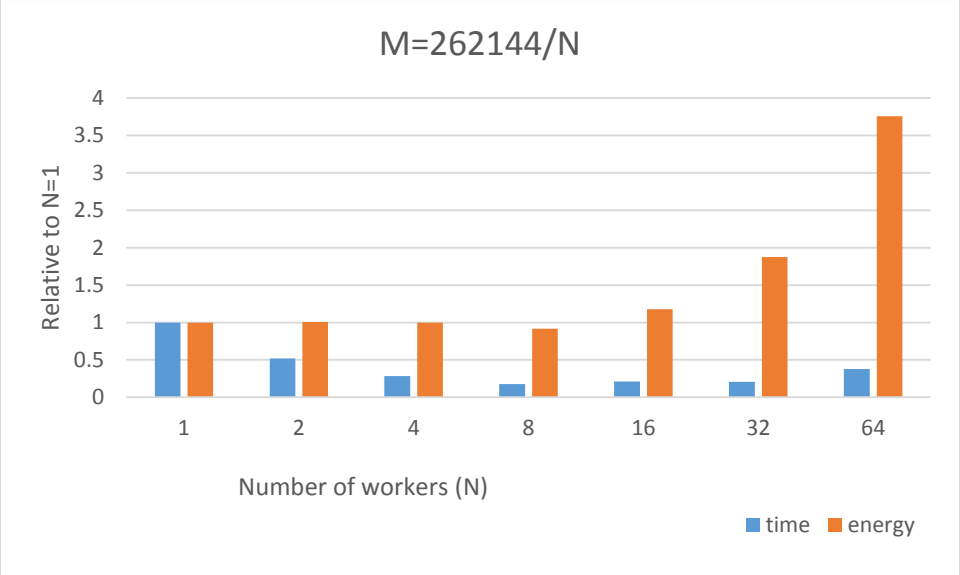


Figure 4.2

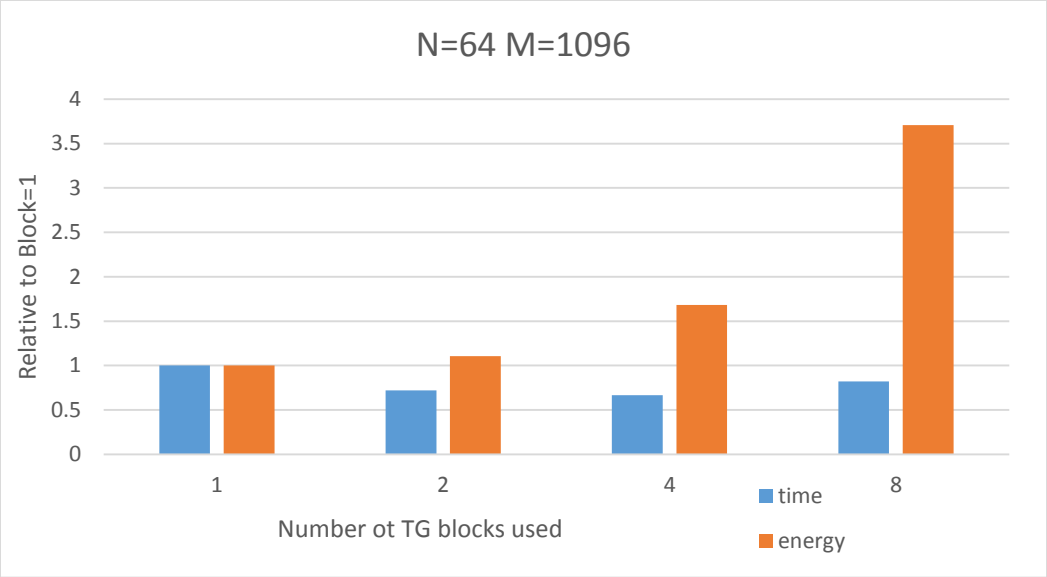


Figure 4.3

Stencil 2D

ocr: sandbox/shared/tgSchedulerAug2015 commit: ebb65abb8889d4edadecaca9fc6d9e0789fc85b3
fsim: master commit: 011a4e4902e4c1a14065f4ca7fc97fadd9bfa7c4

This code involves applying a gradient magnitude operator on a 2-D scalar field over an NxN uniformly discretized grid. The domain is split into a 2-D grid of smaller domains and the stencil operations are applied within each subdomain via halo exchanges with its surrounding 4 neighboring subdomains. The computational tasks in each subdomain are further split into smaller EDTs –send/recv EDTs for halo exchanges with left, right, top and bottom neighbors and the main stencil computation EDT that applies the stencil operator. A dependency graph is setup between these EDTs to advance to a single iteration. The stencil computation is repeated for multiple iterations in the entire computational domain where the output from the previous iteration becomes the input for the next iteration. The solution is verified at the end of the simulation. It is worth noting that the dependency graph sets up dependency links between the sender and receiver EDTs between the nearest neighbors and that the implementation has no global barriers in between the timesteps exposing as much asynchrony as possible.

Strong scaling on TG/Fsim is presented next for two problem sizes: small problem size of 512x512 grid with a memory footprint of 4 MB and a large size of 1024x1024 with 16 MB footprint. For reference, the combined L2 Scratchpad memory available on 8 blocks (only 896 KB out of actual 2 MB sL2 per block is available for the application) is about 7 MB aggregate. Therefore, the “small” problem fits in the on-die memory whereas the large doesn’t on 8 blocks. In all cases there was one “worker EDT” per XE.

Figure 4.4 shows the TG’s performance normalized to serial OCR on x86 (1 Xeon(R) CPU E5-2690 core @ 2.9 GHz) for 1, 2, 4 and 8 blocks (each block with 8 XEs) for the small and large problem sizes. TG’s performance on 1 block is about 9% for small and 12% for large compared to OCR on single x86 core. For the small problem, TG performance stayed relatively flat up to 4 blocks and worsened on 8 blocks indicating lack of scaling for the fine-grained tasks in these initial experiments. And the poor scaling trend is also observed for the large problem where the TG performance increased slightly from 12% (single block) to up to 19% (on 8 blocks) compared to OCR on single x86 core.

Energy per grid point scaling with block count is shown Figure 4.5. Ideally, the energy is expected to stay constant. Parallelization overheads due to increased surface to volume ratio result in energy increase as the block count increases. The small problem energy increased by 1.6x on 8 blocks compared to 1 block whereas the large problem energy trend is relatively better and shows 1.9x increase in energy on 8 blocks compared to a single block.

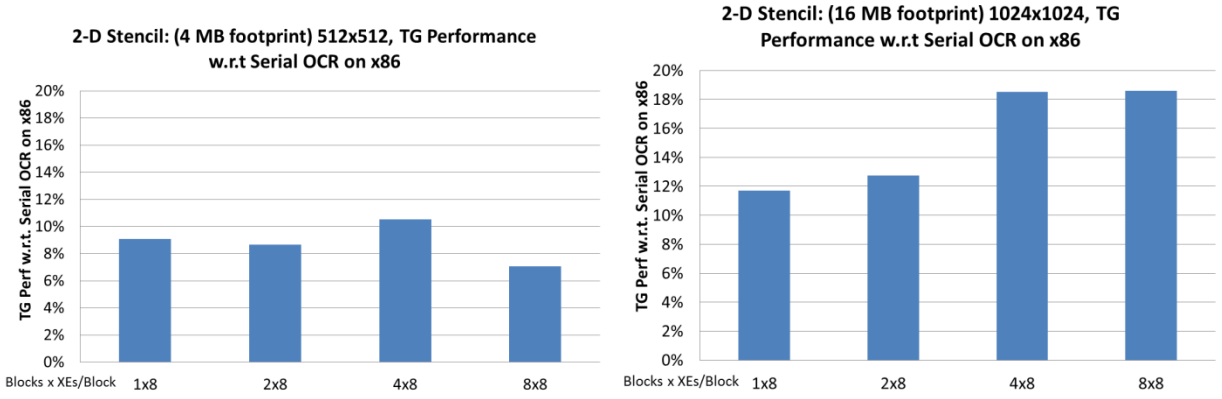


Figure 4.4: TG Performance on up to 8 blocks compared to a single x86 core.

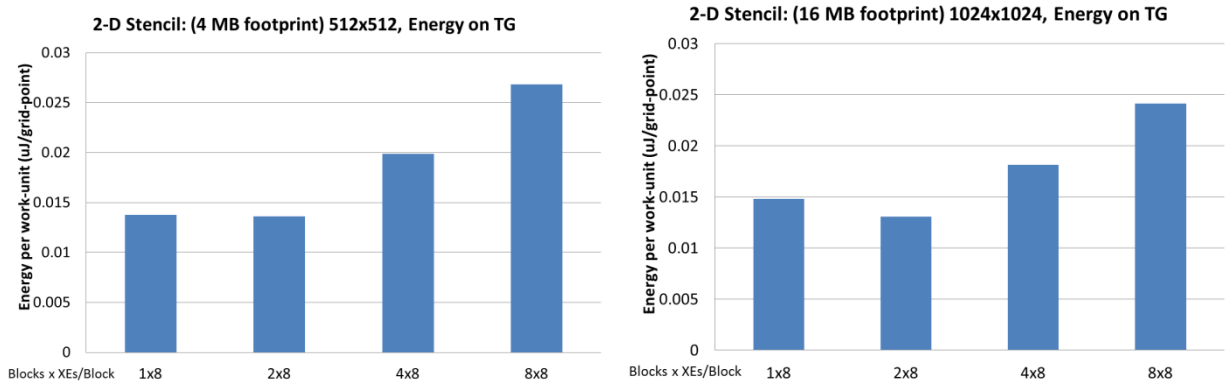


Figure 4.5: TG Energy of 2-D stencil strong scaling study on up to 8 blocks.

Recommendations and Future Directions

Most parallel algorithms can be translated directly to the OCR environment. Datastructures become datablocks. Worker EDTs clone themselves every time new data is needed. The resulting code is verbose and debugging race conditions can be difficult. A key performance question is the overhead in OCR which must be reduced going forward. Using FSIM to measure execution time and power is currently unreliable. To improve performance on TG it will be necessary to take better advantage of the scratchpad memory. This will require coordination between the OCR implementation and the application coder. There is hope that tools, libraries, and/or a higher level environment will allow programmers to write performant code with less effort than writing directly in OCR.

Products and Bibliography

The source code for OCR versions of Lulesh, Stencil1D, and Stencil2D are available in the xstack public repository

5. University of Illinois, Urbana Campus - Applications and Productivity

By Adam R. Smith, Chih-Chieh Yang, David A. Padua (PI)

Challenge Focus

Our group was focused on the integration of the HTA library with the OCR runtime. Our challenge focus was twofold. First to provide productivity to programmers, and second, to separate the user from the runtime system.

Hierarchically Tiled Arrays

Hierarchically Tiled Arrays (HTAs) is a class of objects and operations that encapsulate tiled arrays and data parallel operations on them. HTAs and their operations provide a natural expression for most parallel applications using tiling ubiquitously to control locality and parallelism. We believe the HTA programming paradigm is a great solution to the programmability difficulties of asynchronous tasks. HTAs have been successfully implemented in Matlab and C++, and have been studied for both distributed and shared memory environments.

Productive Parallel Programming

Essentially an HTA program can be seen as a sequential program containing operations on tiled arrays. It has been demonstrated that HTA code is expressive, concise, and easy to reason about. It is also simple to start from a baseline sequential program and parallelize it with the HTA notations. The model provides a global view of data which lets tiles and scalar elements be easily accessed through chains of index tuples without explicitly specifying communication functions to fetch the data required. These features improve programmability and minimizes application development time.

Separation of Application Code and Runtime System

Application codes written in the HTA notation are portable across different classes of machines since low level details are not exposed to the users; the programmer only needs to be concerned with expressing their application. For example, a map operation can be implemented using a parallel for loop, or it can be implemented as SPMD computations. Users write the same code using the high level constructs provided by the HTA paradigm and they need not know the details of the underlying machines. Compared with completely rewriting existing applications using OCR, it can be preferable for application developers to use the more familiar programming paradigm provided by HTA while still enjoying the benefits of executing applications on the OCR runtime system.

Technical Approach

The Parallel Intermediate Language

We have implemented HTAs on top of the Parallel Intermediate Language (PIL). A PIL programmer describes the parallelism available in a program using a collection of PIL nodes. An example can be seen in Figure 5.1 below in which each PIL node represents a parallel loop that can have one or more instances executing in parallel. The user or the library fills in the body of PIL nodes as a sequential function. Our current implementation uses C functions as the body of PIL nodes. Since HTAs are implemented in PIL, the user can write code directly in PIL while leveraging the provided HTA library.

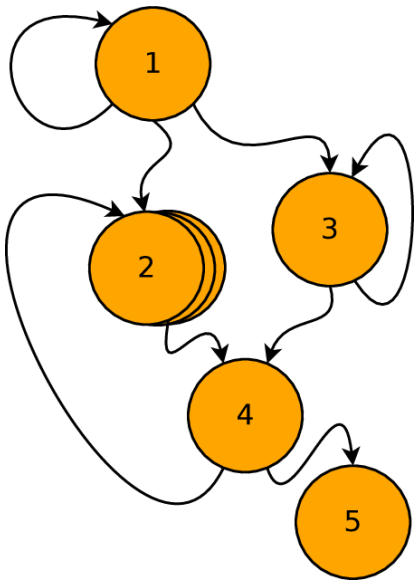


Figure 5.1: An example PIL program graph.

PIL is capable of generating code for several backends, including the Open Community Runtime (OCR). For OCR, each PIL node is implemented as three Event Driven Tasks (EDTs), as seen in Figure 5.2 below. There is one each of an Entry, Body, and Exit EDT. Since PIL nodes are parallel, there can be multiple instances of the Body EDT, as described in the user's program. The entry node forks all of the instances of the body EDTs, and they join at the exit EDT. This provides fork/join parallelism within a PIL node. Any data that a user needs for the node is stored in Data Blocks (DBs). PIL provides automatic encapsulation of data, so they need not worry about the creation of DBs themselves.

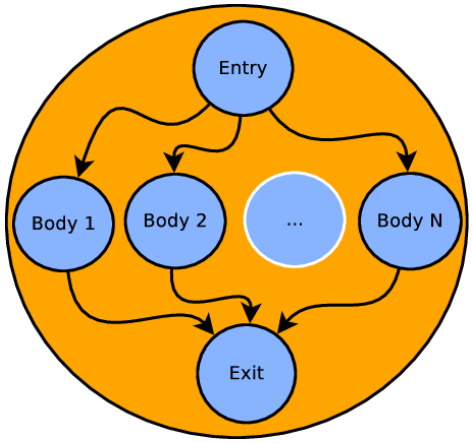


Figure 5.2: The EDTs that make up a single PIL node.

We want a version of HTA+PIL that runs with as much asynchrony as possible, so we have developed a new SPMD programming model for PIL and the HTA library, as depicted in Figure 5.3 below. Previously, all instructions outside of the HTA library were executed by a single main EDT. Any computations by this main EDT would need to be communicated to the parallel operations of the HTA library. In an effort to reduce communication, the SPMD approach will have multiple EDTs each executing exactly the same instructions, thus performing extra computation in exchange for reduced communication. Each of the new EDTs will only need to communicate with its successor to provide data.

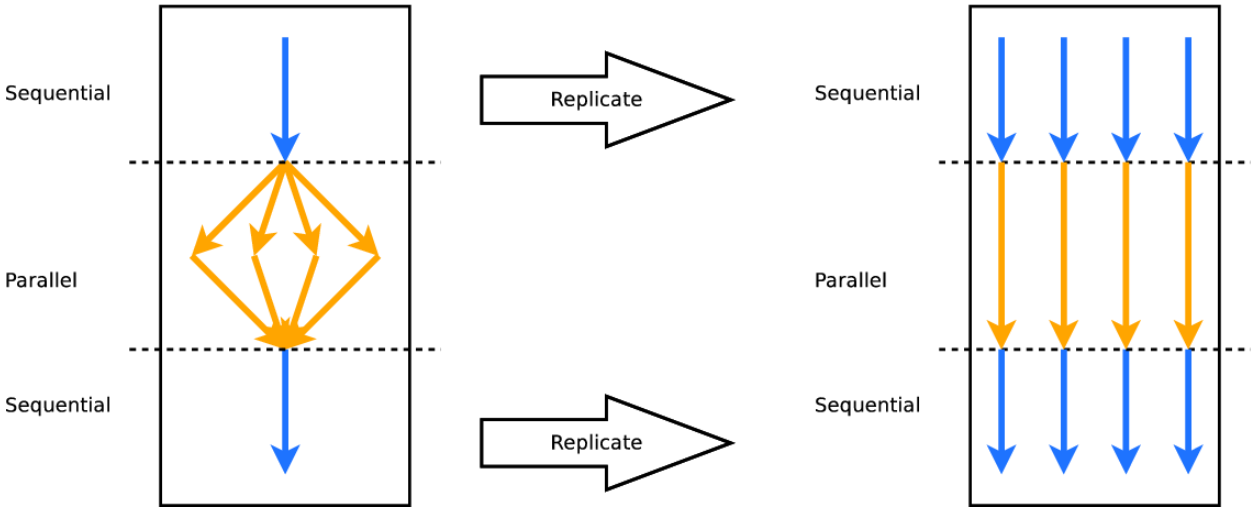


Figure 5.3: The PIL fork/join model (left) and SPMD model (right)

Hierarchically Tiled Arrays

We have provided an implementation of HTAs in PIL to provide a high level way of representing data and its relation to computation and communication. The HTA data structure is a C struct complemented by other data types such as Tuple, Region, and Distribution to be used in the creation and selection of HTA tiles.

An HTA program can be divided into two parts - the sequential part and the parallel operations. The sequential part includes everything that is not a parallel operation. For example, the computation that updates the stack variables when calling a subroutine, forms the sequential part. The parallel operations, when executed, require multiple EDTs working on different tiles of the HTA, possibly in parallel. HTA programs can be executed in both fork/join PIL and SPMD PIL.

With the fork/join PIL, one long-lived EDT executes the sequential part, and spawns slave EDTs whenever a parallel operation is encountered. The slave EDTs join back to the master when they are done with their assigned work so that the master EDT can proceed the execution of the sequential part until the next parallel operation happens. The approach is intuitive and easy to implement. However, it results in implicit barriers for all parallel operation invocations and hinders the execution performance.

The alternative is to run HTA code in SPMD PIL. The program execution starts with multiple main EDTs (conceptually the same as “processes” in conventional SPMD model), each with its own rank, redundantly executing the sequential part. HTA data tiles are statically distributed to these EDTs. Since the sequential part is redundantly computed, all the EDTs have identical program state and thus a global view of data (i.e. they know how to acquire data tiles owned by the others). When a parallel operation is executed, an EDT examines whether the computation is owned by it following owner-computes rule. If so, it gathers the input operands from other EDTs using point-to-point communication and executes the operation. Otherwise, it might still need to supply the data tiles to the owner of computation. When it is part of the parallel operation, either to supply data or to perform computation, is completed, it can then proceed to the subsequent sequential part of the program without waiting globally for all other EDTs.

With SPMD execution, implicit barriers are no longer required, since the owner of computation synchronizes with the suppliers of the input operands right before performing a parallel operation. This point-to-point synchronization is the necessary minimal amount of communication due to real data dependences. The execution can proceed once the data dependences are satisfied without having to wait for all other EDTs at a centralized synchronization point in contrast to the fork/join model. Thus, it grants more asynchrony in the execution which potentially results in better performance.

One issue with SPMD execution is that there is an upper bound of exploitable parallelism. At any given moment, a fixed number, P, of EDTs representing processes can execute. Intuitively, P should be the number of processors. However, when the workload distribution is not balanced, the running EDTs could be less than the fixed number P due to the wait caused by data dependences. To overcome this problem, it is necessary to have an extra level of nested parallelism which allows each process EDT to spawn more EDTs for finer-grain parallel computations and so the runtime system can dynamically schedule them onto idle cores. We present the results to compare against the single-level SPMD execution in the later section.

Integration with the R-Stream Compiler

We have integrated HTAs with Reservoir Lab's R-Stream compiler through integration with the PIL compilation framework. A visual depiction of the PIL compiler flow can be seen below in Figure 5.4. The HTA library is implemented on top of PIL. The functions that comprise the PIL node bodies are fed into the R-Stream compiler which emits backend code. PIL emits backend code for the structure of the PIL nodes. Later, the backend compiler is called to generate a single executable for the entire program. The R-Stream compiler automatically transforms the user's code to provide tiling for locality and auto parallelization. This allows the user's code to be written as simply as possible, while extracting maximum performance. The PIL nodes express one level of parallelism, and any parallelism discovered by the R-Stream compiler in the body function gives us a second level of parallelism, thus providing nested parallelism.

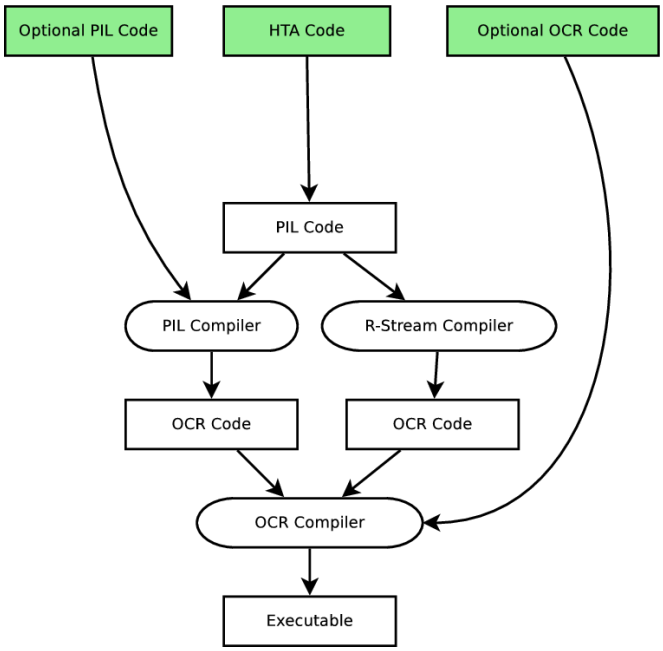


Figure 5.4: The PIL compiler flow.

Results and Analysis

All results shown here were gathered on the XSTG FooBar cluster with dual Intel Sandybridge E5-2690 processors with 16 cores and 32 threads and 128 GB of memory, and OCR commit number 6f11691d0d86f036bb378f0ea4f08b9e4749ca36 on the runtime/master branch.

Cholesky Decomposition

Cholesky decomposition is a linear algebraic decomposition of a Hermitian, positive-definite matrix A into the product of a lower triangular matrix L and its conjugate transpose L^*, such that $A = LL^*$. Cholesky decomposition is commonly used to solve systems of linear equations, since it is more efficient than LU factorization. We have

chosen to implement a tiled Cholesky decomposition, as it is easily expressible in terms of operations on tiles, and the tile accesses can provide a locality optimization versus a naive implementation.

Cholesky decomposition allows us a unique opportunity to scale the input sizes of the data as we see fit. We have standardized our results using an 80x80 tiled matrix, while changing the sizes of the tiles. We have four tile sizes: 1x1 element per tile, 100x100 elements per tile, 200x200 elements per tile, and random elements per tile. For each experiment where we change the tile size we have three versions of the code, and each version of the code generates two different backends. The backends used are OpenMP and OCR. The versions of the code are HTA fork/join (orange lines), HTA SPMD (green lines), and hand coded PIL fork/join (blue lines). We include a hand coded version of the benchmark in PIL to show how little overhead the HTA library has.

The 1x1 elements per tile experiment allows us to measure the overhead in creating all of the necessary tasks for the benchmark. Execution times for this experiment can be seen in Figure 5.5. It can be seen that the OpenMP generated code for all versions of the benchmark have significantly lower executions times than the OCR generated code. This is due to the task creation and scheduling overhead in the OCR runtime.

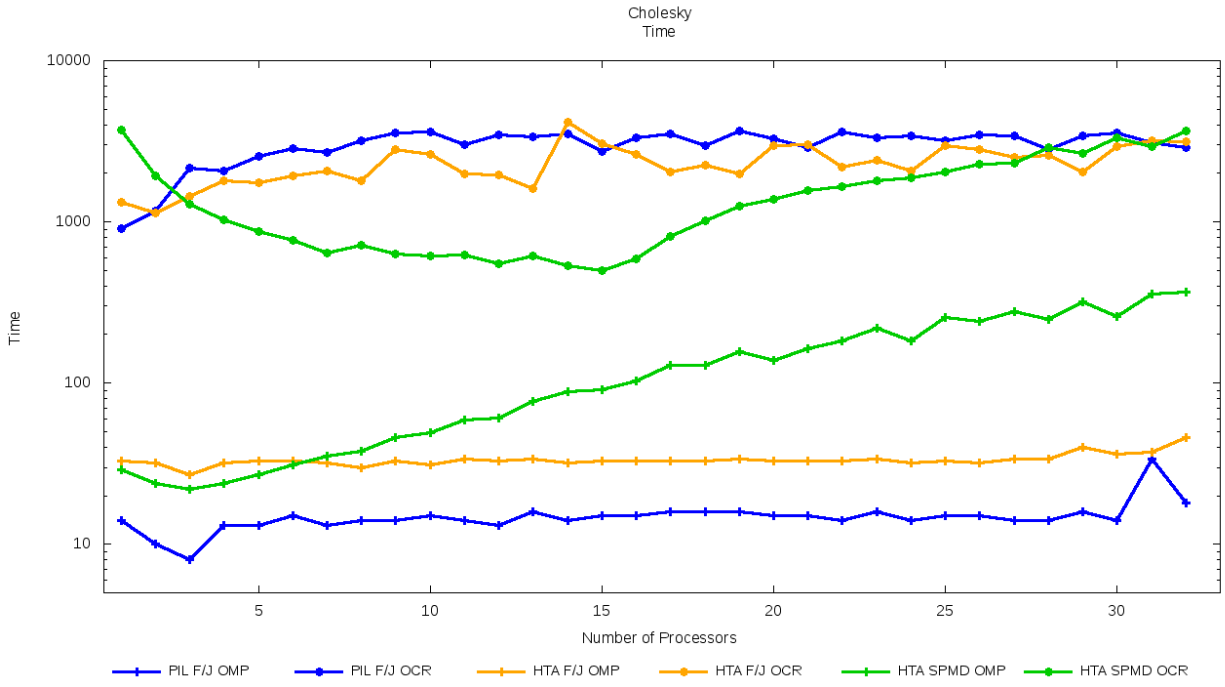


Figure 5.5: Cholesky decomposition execution time with 80x80 tiles and 1x1 elements per tile

The 100x100 elements per tile experimental speedups can be seen in Figure 5.6. We see that once again, the OCR generated codes are clustered at the bottom because of the overheads of task creation and scheduling. There is not yet enough work within a single tile to amortize this overhead.

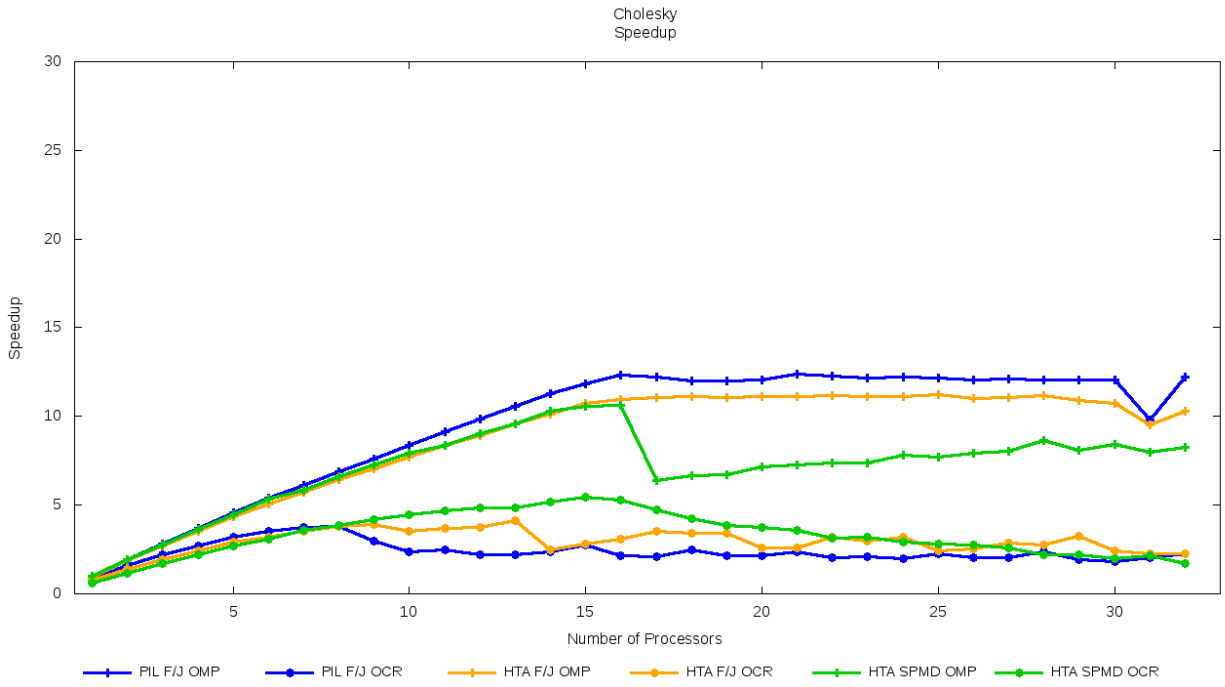


Figure 5.6: Cholesky decomposition speedup with 80x80 tiles and 100x100 elements per tile.

The 200x200 elements per tile experimental speedups can be seen in Figure 5.7. Here the amount of work within a single tile allows the overhead of task creation within OCR to be amortized, and the OCR numbers are much closer to the OpenMP numbers.

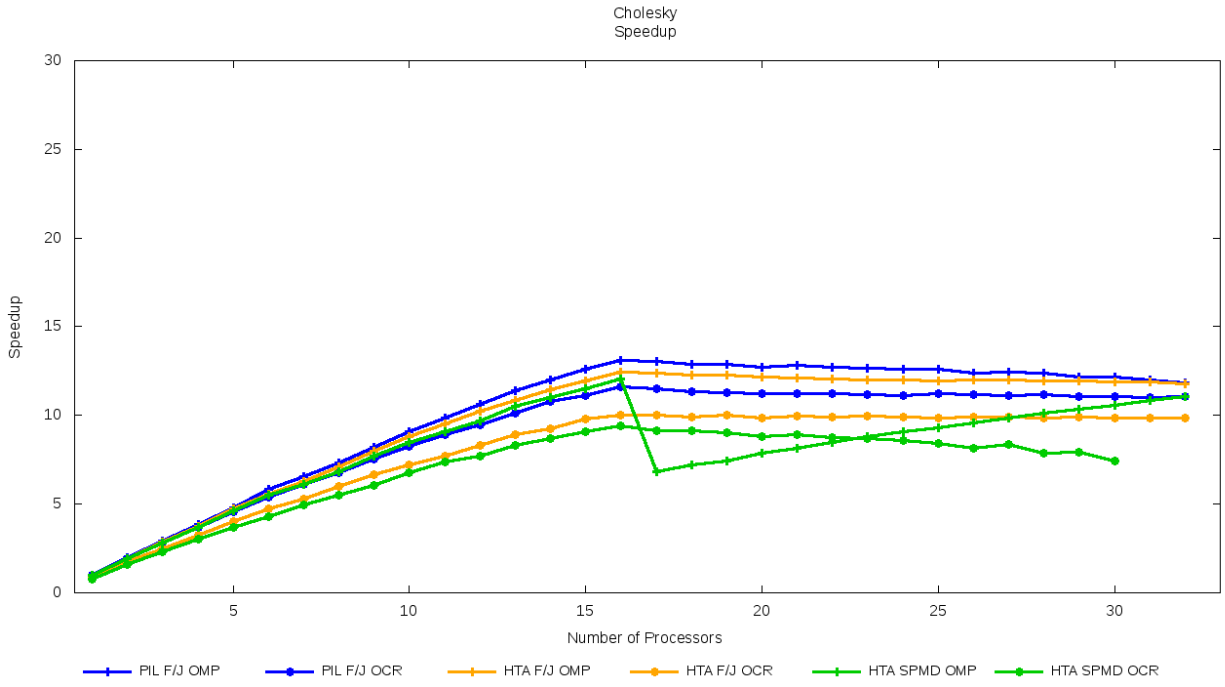


Figure 5.7: Cholesky decomposition speedup with 80x80 tiles and 200x200 elements per tile.

The random elements per tile experimental speedups can be seen in Figure 5.8. This experiment was run as a simulation of sparse data and load imbalance. Since each tile has a random amount of work to perform, load balancing must be performed by the runtime system scheduling EDTs to achieve good results. In this experiment, the OCR runtime does a very good job of scheduling the EDTs in the fork/join algorithm to achieve load balance, and we are able to achieve the best speedup numbers. The gap between the OCR and OpenMP generated code is much smaller than the previous tile sizes, and is accounted for by the overheads discussed above. For the SPMD code, we are unable to keep all of the cores as busy as in the fork/join version, and are thus under-utilizing the machine.

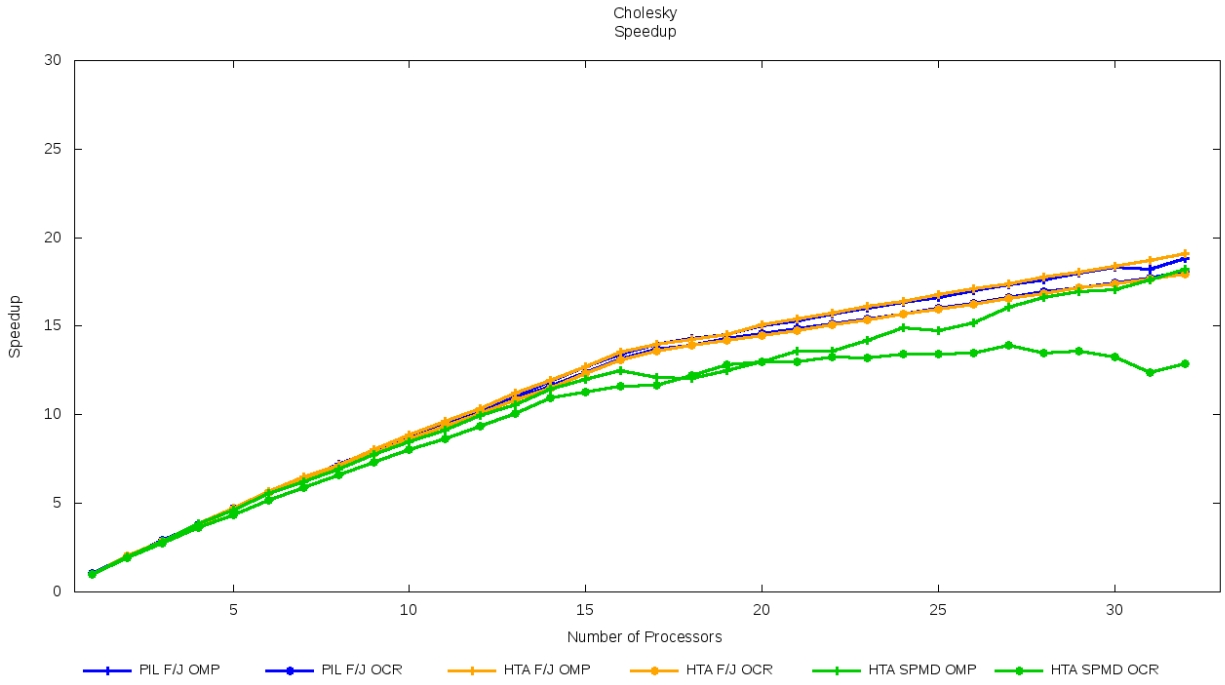


Figure 5.8: Cholesky decomposition speedup with 80x80 tiles and random work per tile.

In all of the graphs, it is apparent that the SPMD code is not able to achieve enough asynchrony to reliably outperform the fork/join version of the code. The reason for this is twofold. First, the SPMD version of the code is required to communicate data during the synchronization, and this is not required in the fork/join version of the code. Second, and most importantly, we have discovered that our initial implementation of the HTA library has the presumed necessity of statically distributed data. This static distribution of data can lead to large load imbalances as tiles are distributed initially, and the Cholesky task graph changes with each iteration.

The NAS Benchmarks

The NAS Parallel Benchmarks are a set of programs designed to evaluate parallel supercomputers. Each of the benchmarks within the NAS suite are designed to stress a particular portion of the machine. This gives us a good foundation to show that HTAs can support all of the operations contained within the complex set of benchmarks. We have implemented six of the NAS benchmarks: EP, IS, FT, CG, MG, and LU.

We have less flexibility in the NAS benchmarks than we have in the Cholesky decomposition benchmark described above. In the Cholesky decomposition, we can adjust the number of tiles as well as the size of the tiles as we see fit. However, the NAS benchmarks describe the input data for each benchmark. We are required to use the data provided with the benchmark. The size reported in these numbers are the class C size.

There are two implementations of the HTA library as discussed in detail in previous reports. The first is a fork/join version and the other is an SPMD version. However, application code requires only to be written once, and can use the fork/join or SPMD library with a simple recompile. Since the HTA library is implemented using PIL, that can generate code for multiple parallel runtimes, we can generate multiple backend versions of the application at compile time. This provides the programmer with four versions of the code generated from a single application program. Here we use the same HTA implementation to generate OpenMP code in addition to OCR code, and compare the HTA code to a highly tuned hand-coded version of the benchmarks in OpenMP.

The NAS Benchmarks with Fork/Join Parallelism

The fork/join model uses global barriers for synchronization. This assures that any data computed before the barrier is synchronized with all threads that may need the data. Global barriers can be expensive when there are a large number of tasks or there is a load imbalance between tasks.

In Figure 5.9 we can see speedup numbers of our fork/join HTA implementation. The magenta line shows execution performance of an efficient pure OpenMP implementation of the algorithm. The red line shows OpenMP code generated from the HTA implementation. The blue line shows OCR code generated from the HTA implementation.

It can be seen that the HTA generated OpenMP code scales nearly identically to the handwritten OpenMP version, and in fact outperforms it for the MG benchmark. It can be seen that when exceeding the 16 physical cores and using 32 hardware threads most of the benchmarks see no benefit since the algorithm is sufficiently compute bound. The OCR generated code shows some overhead, but still scales up to 16 PEs.

The NAS Benchmarks with SPMD Parallelism

The SPMD model removes barriers and uses point-to-point synchronization. This means that tasks only synchronize with each other when the need data that the other has. The point-to-point synchronization allows asynchrony in this manner, but also has the added overhead of communication costs that were not necessary in the fork/join version.

In Figure 5.10 we can see the speedup numbers of our SPMD HTA implementation. Once again, the performance of the OpenMP generated HTA code is very similar to the highly tuned handwritten OpenMP code. For the CG and MG benchmarks the OCR generated code scales until 8 PEs, but significant overheads are incurred at 16 and 32 PEs.

In the NAS benchmarks, the problem size is clearly defined. In the strong scaling experiments discussed above, that means that as the number of PEs in increased, the tile size decreases. This means that the work done within a single EDT is reduced as the number of PEs is increased, thus increasing the effect of the overhead of creating and firing the EDT. As the amount of work within a single EDT increases, the portion of execution time lost to overhead decreases. This is not surprising. What is surprising is that the amount of work that needs to be done within a single EDT to amortize the overhead is quite large.

Nested Parallelism in SPMD Execution

To mitigate the performance issue in SPMD execution, we have experimented with introducing nested parallelism within processes. By using an extra level of parallelism in SPMD execution, the amount of exploitable parallelism is no longer limited to a fixed number P. At the same time, the abundant finer grain level tasks could be scheduled by the runtime system on a shared memory address system, thus achieving better dynamic load balance. We conducted an experiment with HTA Cholesky decomposition application running on PIL OpenMP backend on a single node machine with 4 Intel Xeon E7 4860 processors (40 cores and 80 threads) to demonstrate the benefits of this scenario.

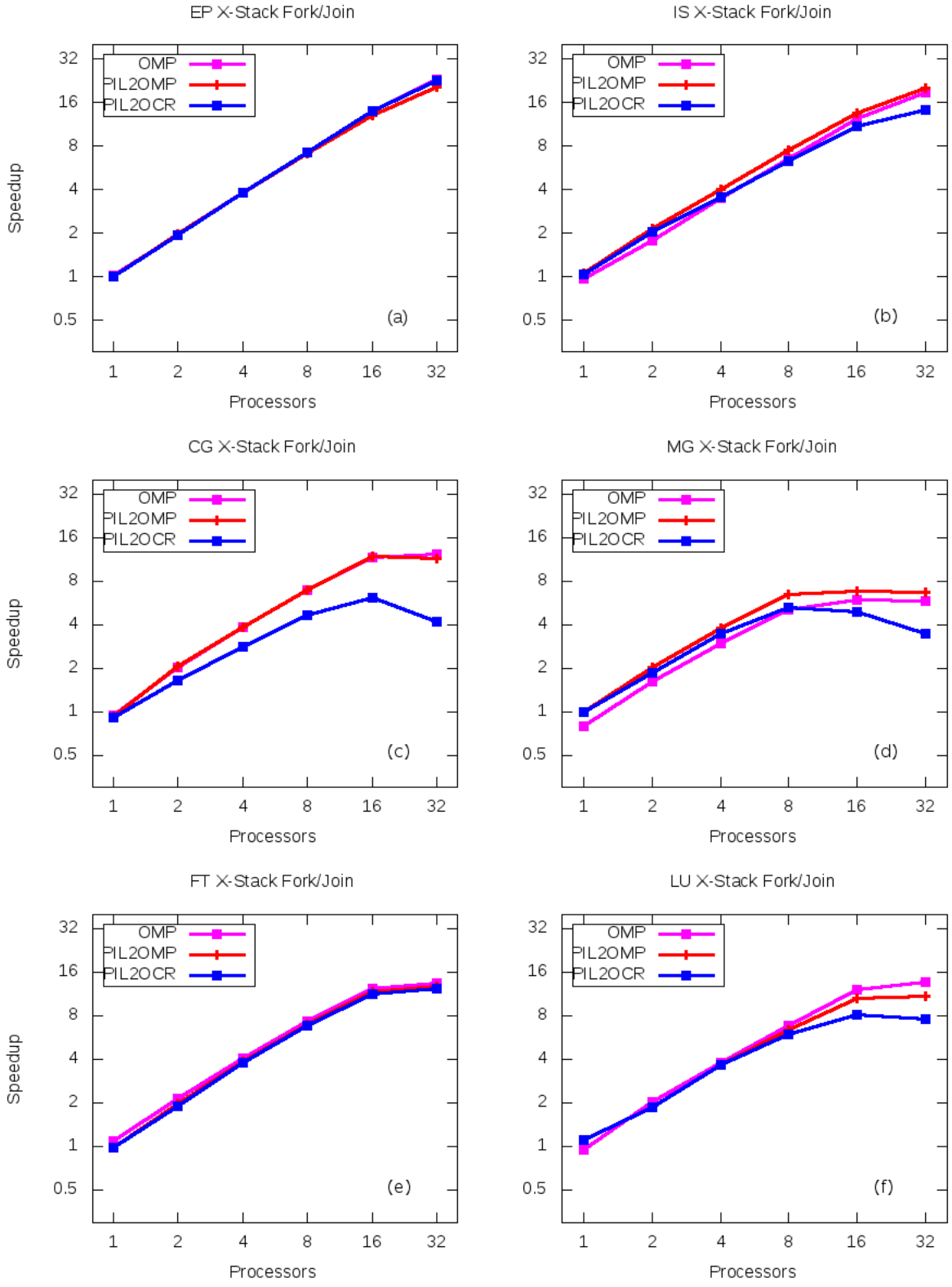


Figure 5.9: NAS fork/join speedup.

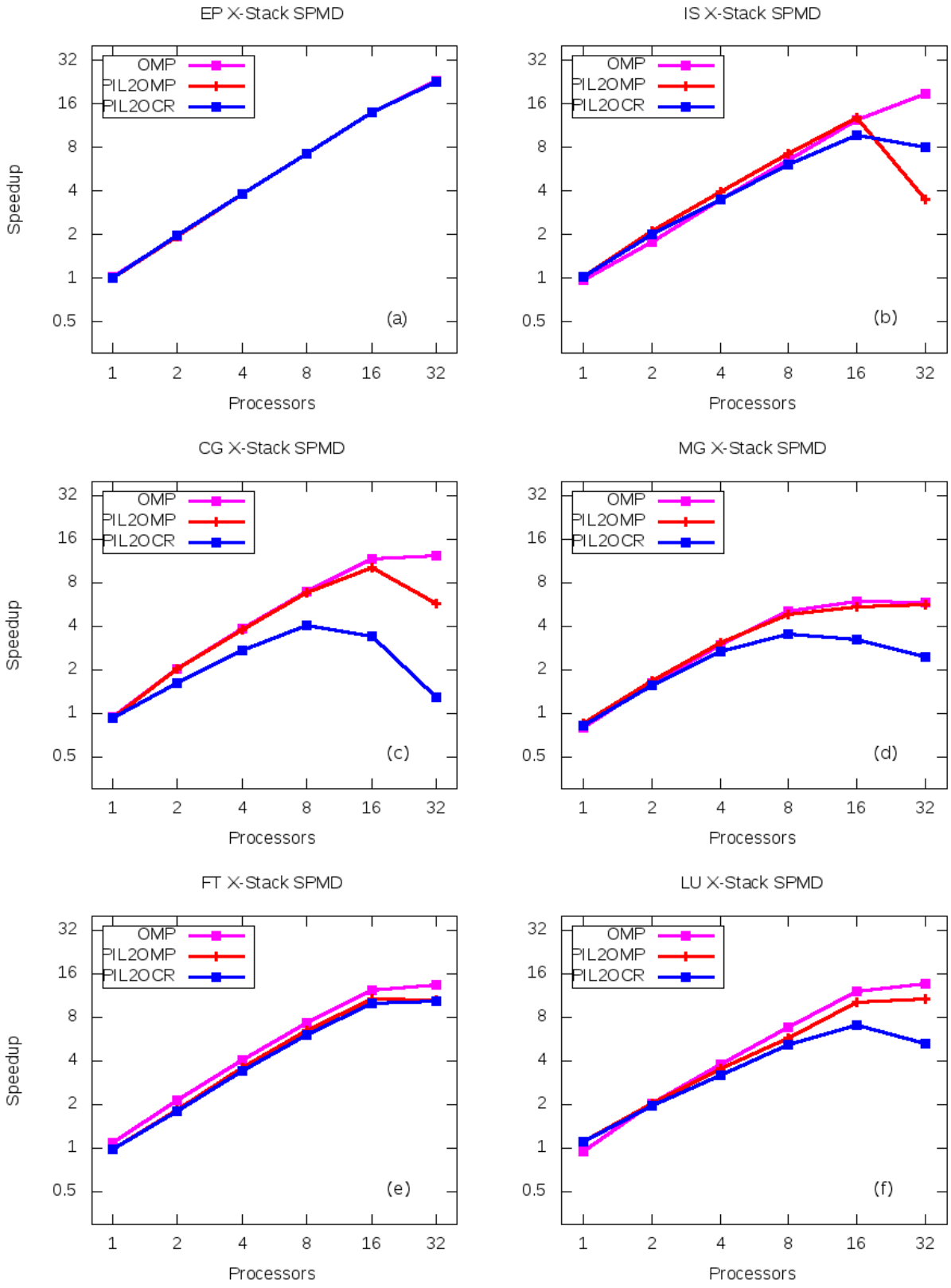


Figure 5.10: NAS SPMD speedup.

As shown in Figure 5.11, using the same HTA application code, the performance scales better in fork/join execution mode (the orange line) than in SPMD execution mode (the green jagged line). The reason that SPMD execution does not perform well is because the HTA data tiles are statically distributed to EDTs representing processes. Whereas the workload associated with the data tiles changes with the progression of the execution. When the workload associated with the tiles change but the distribution does not change with it, load imbalance occurs. When a process EDT owns more tiles to compute in an operation, other process EDTs run out of work sooner and some processors are left idling.

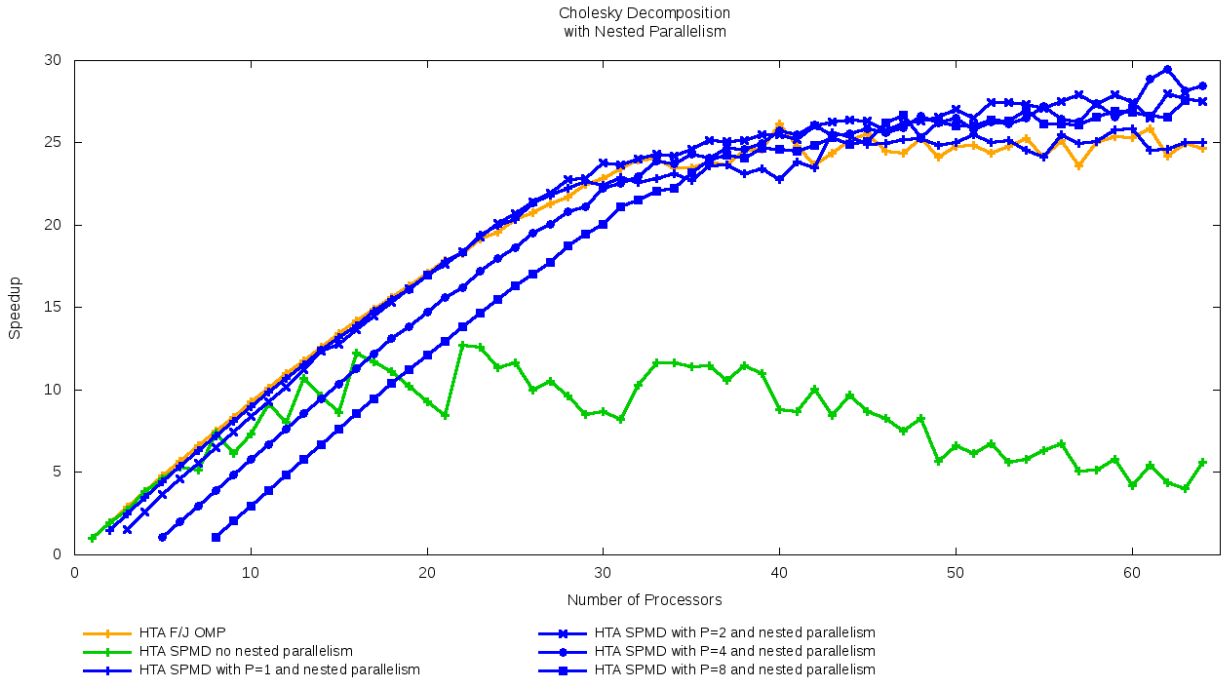


Figure 5.11: Nested parallelism in SPMD execution with Cholesky decomposition speedup with 64x64 tiles and 100x100 elements per tile.

When nested parallelism is available, there are more fine grain EDTs in the ready queue for the runtime system to dynamically schedule them on a shared memory address machine. In this scenario, we get similar performance scaling result as fork/join execution when P=1, and better results at higher worker thread counts when P=2, 4 and 8. There are some interesting future research topics left to explore. For example, given the number of processors on a shared memory machine, the optimal number P for the first level parallelism is application dependent. Also, by assigning priority to the EDTs according to the position of the tasks in the dependence graph, it is also possible to minimize the wait due to data dependences and results in better performance.

Integration with the R-Stream Compiler

We have evaluated the integration of PIL and R-Stream with a matrix-matrix multiplication benchmark written in PIL. The benchmark program is first compiled with PIL compiler, and the computation kernel is compiled by R-Stream. Both compilers generate code for the same runtime and finally everything is built by gcc into a single executable.

The algorithm parallelizes matrix-matrix multiplication by rows. A fixed number P is statically chosen to represent the amount of parallelism at coarse-grain level in PIL (i.e. P groups of rows), and a sequential compute kernel performs dot-product computations in a group. The sequentially written compute kernel is optimized and parallelized by R-Stream, which creates fine-grain parallel tasks (or EDTs) to perform the dot products within a group, thus providing an extra level of nested parallelism.

Figure 5.12 shows the speedup of P=4 with varying number of worker threads in OpenMP. We can observe that if only PIL is used, the amount of parallelism exposed to the runtime system is limited to 16. As a result, the speedup does not change when the amount of worker threads increase. But when the compute kernel is optimized and parallelized by R-Stream, the overall speedup is much better. Also, when more worker threads are used, the fine-grain tasks generated by R-Stream can utilize the extra worker threads and the speedup scales well. This shows great potential in using PIL for coarse-grain parallelization combined with R-Stream's ability to automatically optimize and parallelize in the fine-grain level. We have also conducted the same experiment with the OCR runtime, which also shows similar speedup results. We are still dealing with some correctness issues, so the results are not presented here.

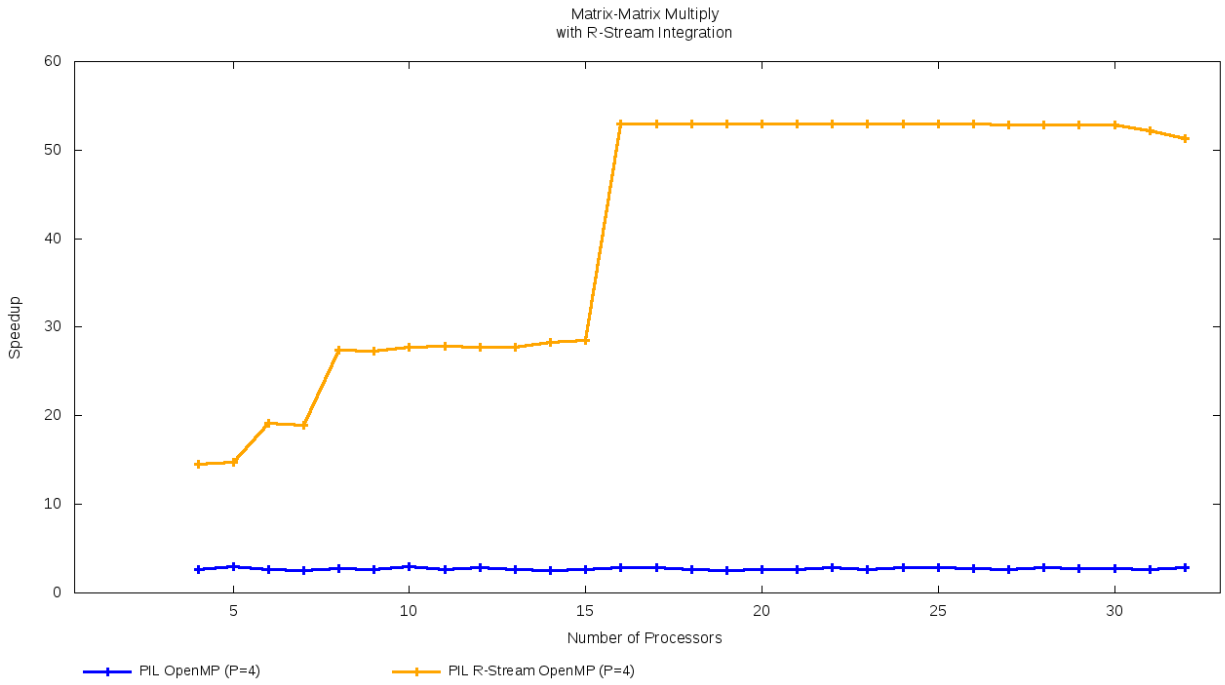


Figure 5.12: PIL Matrix-matrix multiplication benchmark speedup comparison of non-optimized vs R-Stream optimized versions with 2048x2048 matrices.

Recommendations and Future Directions

We have discovered that the overhead of firing an EDT in OCR is greater than anticipated. The original goal of OCR was to provide the user with lightweight asynchronous tasks. We have discovered that they are not yet lightweight. Additional work will need to be put into the runtime and scheduler to make them be lightweight.

Products and Bibliography

All of the source code for the HTA library and PIL language can be found on the community release of OCR at https://github.com/01org/ocr. The source and documentation can be found in the hll/hta directory. Any questions pertaining to the work described in this document can be directed to David Padua at padua@illinois.edu.

Chih-Chieh Yang, Juan C. Pichel, Adam R. Smith, David A. Padua. *Hierarchically Tiled Array as a High-Level Abstraction for Codelets*. In the Fourth Workshop on Data-Flow Execution Models for Extreme Scale Computing, 2014.

6. Rice University

By Vivek Sarkar, Zoran Budimlić, Vincent Cavé, Kath Knobe, Nick Vrvilo

Challenge Focus

OCR provides a low-level C API, accessible to concurrency experts ("hero programmers"), or suitable as a target for compilers and libraries. Our goal is to present higher-level programming models for OCR that are accessible to a larger audience of programmers, who do not necessarily have the low-level programming skills or the extensive knowledge of the complex OCR APIs to use OCR directly. These higher-level programming models still need to deliver competitive parallel performance. We also aim to make programmers more productive by providing simple APIs, concurrency safety guarantees, and debugging tools. Finally, we strive to maintain a separation of concerns between the application code and platform optimization by providing a decoupled tuning mechanism. We chose HCLIB and CnC as the programming models to realize these goals. We also worked with our DoE partners at PNNL on implementing two DoE proxy applications in CnC on OCR, as well as developing several smaller kernels in both CnC and HCLIB.

Technical Approach

HCLIB

HCLIB is a library-based programming API implementation of the Habanero-C language that can execute on top of an OCR execution platform. The HCLIB programming API provides a bridge between traditional parallel programming patterns and the fully event-driven macro-dataflow programming style the OCR execution model promotes. The HCLIB programming API proposes a collection of high-level constructs to write explicitly parallel programs that covers popular parallel programming patterns such as fork-join, loop chunking and macro-dataflow. Additionally, accumulators and phasers constructs are respectively used to perform safe reductions and for lightweight point-to-point and collective task synchronization.

The HCLIB distribution is composed of a C-based header file that exposes the HCLIB programming API along with a supporting runtime implementation that can use OCR as an execution platform.

HCLIB programming model

HCLIB relies on three main constructs to express parallelism: async, finish and data-driven future.

The async API call causes the currently executing task to fork a new child task to execute a function that takes the specified arguments. The async call returns immediately, i.e., the parent task can proceed to its next statement without waiting for the child task to complete. The finish construct materializes as two API calls to respectively start and end a finish scope. A finish scope performs a join operation that causes the currently executing task to execute all statements in between start and end, then wait until all the tasks created within have terminated (including transitively spawned tasks). Data-Driven Futures (DDFs) enable asyncs to be coordinated in a macro-dataflow fashion. A DDF instance holds a value that can be set through a put operation and read through a get operation. DDFs are single-assignment, meaning a unique put can occur on a DDF. When a put occurs, the DDF is satisfied and its value can be read through a get operation. An async instance can depend on a list of DDFs. In that case, the async is scheduled for execution only when each of the DDF it depends on has been satisfied. We name an async that has DDFs dependences a Data-Driven Task (DDT). A DDT typically reads its input from a set of DDFs

and is executed only when all its inputs are available; i.e. a put has occurred on each DDF. A DDT may enable other DDTs to run by putting on DDFs they depend on.

HCLIB is a convenient stepping-stone to convert an application to OCR. For instance when an application is already parallelized with OpenMP, classical parallel programming patterns such as fork-join and embarrassingly parallel loops can be translated to HCLIB finish-async and forasync constructs. The programmer can then progressively refactor parts of the application to the HCLIB macro-dataflow model using asyncs and DDF constructs that map well to the EDTs and events of OCR. Since HCLIB is library-based, the kind of code refactoring effort necessary to use the HCLIB programming API is similar to that of OCR. Namely, identify pieces of code that should be outlined as tasks and define their input and output. However, the level of detail that needs to be specified at the HCLIB level is less than what is needed at the OCR level.

HCLIB implementation

The HCLIB programming API is implemented as a thin layer that relies on an underlying runtime for execution. Precisely, there are three layers of API. First, the user facing API defines data types and functions to manipulate HCLIB constructs. Second, a runtime agnostic layer implements the constructs. Finally, a runtime 'bridge' interface allows for the agnostic layer to call into a supporting runtime. Adding support for a new runtime in HCLIB only entails implementing the bridge interface. Two runtime bridge implementations have been developed. The first one targets the Open Community Runtime (OCR) and the second the Custom RunTime (CRT). The Custom RunTime (CRT) is a stripped down version of the Habanero-C runtime developed at Rice University. The main motivation for developing CRT is to establish a very basic and low-overhead runtime performance baseline on today's systems, though it lacks many of the features of OCR.

The OCR runtime bridge implementation relies on the OCR user API and a number of OCR API extensions. The OCR user API is mainly used to create EDTs that are used to carry out the execution of HCLIB's asynchronous tasks. HCLIB uses two of the OCR API extensions. The OCR legacy interface allows HCLIB to control the life cycle of the OCR runtime (starting and stopping). The OCR runtime interface allows the user to query information such as the number of available OCR workers, provides access to EDT local storage and advertise operations that have a blocking semantic. These APIs are critical to allow a runtime system to operate on top of OCR. While HCLIB uses EDTs, the HCLIB programming model and implementation do not fully comply with the OCR philosophy that all data accessed by an EDT should be done through a datablock (DB) that it depends on. Instead, HCLIB may pass pointers as parameters to EDTs instead of datablocks GUIDs. A direct consequence of this limitation is that the current HCLIB implementation only works on top of an OCR instance built for a shared-memory x86 platform.

CnC-OCR

CnC is a programming model that separates specifying the chunks of computation, the dependences among these chunks, and application tuning. We describe CnC as *dependence programming*. In CnC, we focus on exposing constraints rather than extracting parallelism. A key component of a CnC application is the *graph specification*, a dependence graph describing the relationships among *item instances* (data) and *step instances* (computation chunks) in the application. This graph comprises the exact set of constraints needed for correctness, which exposes the maximal parallel execution schedule for the application. We implemented CnC on top of OCR to increase programmer productivity, and to explore the separation of concerns between applications and tunings. The CnC programming model provides many desirable safety guarantees, such as determinism and data-race freedom.

The CnC-OCR implementation was created from scratch, and shares nothing with Intel's C++ library implementation of the CnC model that was built on the TBB and MPI runtimes, which we refer to as "Intel CnC" in the following text. CnC-OCR has multiple features not available in Intel CnC, including support for an input graph language and translator, and support for mapping on to OCR.

CnC-OCR Implementation

CnC-OCR uses a set of productivity tools to generate a layer of "glue code" between the application programmer's CnC code and the underlying OCR. A CnC-OCR kernel can be embedded seamlessly within a larger OCR application;

however, applications can also be written for CnC-OCR without directly touching any of the OCR API. Writing applications to the runtime-agnostic CnC API allows us to run the same CnC application code with multiple runtime backends. Our currently supported backends include OCR and MPI+TBB (via Intel CnC). Platform-specific optimizations can then be specified in an external tuning file, keeping true to the *separation of concerns* theme of CnC.

Concept	OCR construct	CnC construct
Task classes (code)	EDT template	Step collection
Task instance	EDT	Step instance
Data classes	All DBs have type void* (keeping track of individual DBs' types is the app programmer's responsibility)	Item collection
Data instance	Datablock	Item instance
Unique instance identifier	GUID	Tag (step tag / item key)
Dependence registration	Event add dependence	Item get
Dependence satisfaction	Event satisfy	Item put

Table 6.4: Mapping of programming concepts and constructs in CnC and OCR.

As outlined in Table 6.4, most of the CnC constructs have a direct correspondence to OCR constructs. This allowed us to implement the CnC-OCR runtime "glue" code almost entirely as light-weight wrappers around OCR API functions. The largest obstacle was the mapping between the CnC tagspace (which is currently restricted to integer tuples), and the OCR GUID space. We currently use a hashtable to do this mapping. We have a pure OCR implementation of the hashtable data structure that is compatible with any compliant OCR implementation, as well as versions optimized for specific OCR implementations (e.g., x86 and x86-mpi). Experimental support for "labeled GUIDs"—which allow mapping GUIDs to/from integers—was recently added to the shared-memory x86 OCR. Once this feature is more broadly supported, it could serve as a more efficient way to implement the tag-GUID mapping, especially for dense/regular tag spaces.

Application Programming Toolchain

Graph Translator

The main component of the CnC-OCR toolchain is the graph translator tool. The translator takes as input a textual specification of the CnC application graph, then generates as output the runtime scaffolding code, as well as suggested application code based on the dependence relationships in the graph specification. The translator also accepts zero or more tuning specifications, which are also used during code generation. The translator can be invoked multiple times for the same application, in which case the runtime scaffolding code is overwritten, but the application code remains untouched. This is useful for testing different tunings on the same application. The ability to combine multiple tunings at code generation time allows the tuning expert to keep the definitions of orthogonal tunings separate, and then mix-and-match the tunings to find an ideal combination. For example, separating the tuning spec for task priorities from that for locality allows the tuning expert to reuse the priority tuning with several locality tunings, or among locality tunings for different cluster hardware.

Debugging and Analysis Tools

Debugging task-based, parallel codes is difficult when using standard tools (designed for debugging serial applications) for one big reason: the stack does not hold the control-flow history. We have developed a set of tools

for analyzing CnC application traces that creates an interactive, graphical trace of a CnC program execution. The nodes in the graph represent item and step instances, and the edges represent dependences. All nodes are labeled with their collection name and unique tag (integer tuple), which makes it simple to map a node in the visualization back to the corresponding item or step instance in the execution. This visualization also works on partial application traces, such as those produced when the application deadlocks or crashes.

Since we have all of the dependency information in the application's graph specification, we are also able to do additional analysis on applications to determine the root cause of a deadlock. Often, one missing dependence has a cascading effect, blocking many tasks from starting. By taking the dependences into account, we can analyze the relationships among stalled tasks, and trace those back to the root cause of the problem. We have found that automatically filtering out the noise from a deadlocked application trace using this technique allows the programmer to identify the source of a deadlock much more quickly than when using our current tools for debugging general OCR programs.

DoE Proxy Applications

We worked closely with our partners at Pacific Northwest National Laboratory to design and implement two of the DoE proxy applications using CnC-OCR: LULESH and HPGMG. We also worked with our partners at the San Diego Supercomputer Center to port CoMD to CnC-OCR.

Additionally, we have been in active collaboration with Milind Kulkarni's group at Purdue, exploring the potential for using graph rewriting techniques to optimize CnC applications. Their group is part of the X-Stack SLEEC project, and our collaboration started as a result of a discussion at an X-Stack project PI meeting. Specifically, we have been testing rewrite techniques to automatically tile the PNNL implementation of LULESH (which is currently written at a per-element granularity).

CnC Checkpoint/Restart

The unique properties of the CnC programming model—e.g., single-assignment data items and side-effect-free computation steps—make CnC an ideal programming model for a scalable, asynchronous resilience system. As a proof of concept, we implemented a prototype checkpoint/restart system for CnC in the Habanero C language. The system leverages the unique properties of the CnC programming model to build checkpoints continuously and asynchronously, which we believe to be critical for scalable resilience on future exascale machines.

Results and Analysis

HCLIB Evaluation

Because HCLIB can run on top of multiple runtime implementations, HCLIB programs can be used to compare performances of task-based parallel runtimes. We experiment using a classical fork-join implementation of Fibonacci in HCLIB to compare performances and scaling of the OCR and the CRT runtimes. The comparison is relevant to measure implementation overheads because OCR is configured to use the same scheduling strategy as CRT does. Hence, performances discrepancies are the result of going through different user-facing runtime APIs and runtime implementations

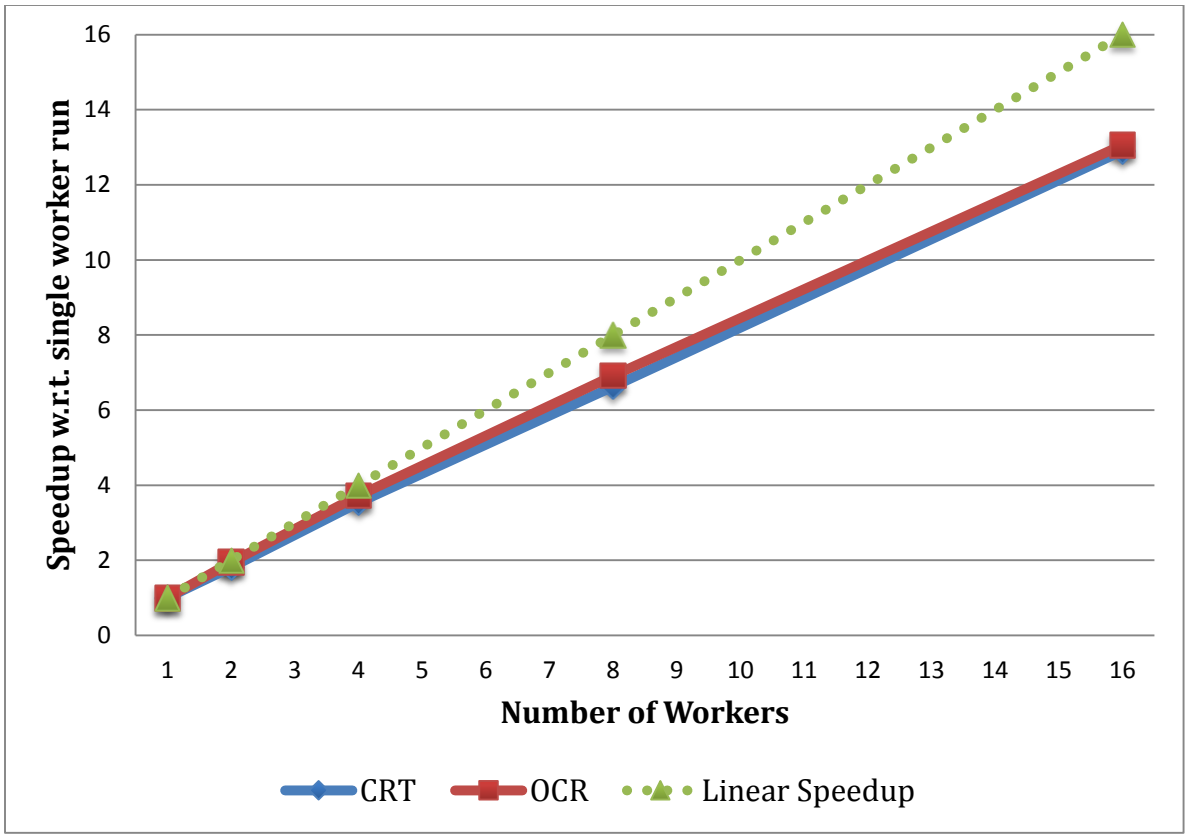


Figure 6.24: Scaling comparison of HCLIB Fibonacci N=36 with cutoff running on top of OCR or CRT

Figure 6.24 shows a scaling comparison of the HCLIB Fibonacci implementation for N=36, running on top of HCLIB targeting either OCR or CRT. In the case of a fork-join style Fibonacci implementation, as soon as the dynamically generated task-based recursion tree is large enough, each worker thread can operate uncontended on a part of the task sub-tree. We can see from Figure 6.24**Error! Reference source not found.** that the two runtimes implementations achieve very similar scaling in this context. Using CRT-based runs as a baseline, we measure the runtime overhead of OCR as being between 5 and 6 times the one of CRT. This is essentially pointing out there is an overhead associated with scheduling an OCR EDT (task) in the OCR v1.0 implementation. This is a parameter application programmers can take into account by controlling the amount of computation carried out by a task. In the Fibonacci example, a cut-off threshold can be implemented so that the leaves tasks of the recursion tree are coarse-grained enough to counteract task's scheduling overheads. For instance, when Fibonacci N=36 is ran with a cut-off value of either C=5 or C=6, an OCR-based run overhead can range between plus or minus twenty percent of the CRT baseline. These experiments were run on the runtime/master head revision from 2015-07-28 (ba75c45) with an additional patch for HCLIB (gerrit 1661) to be part of v1.0.1 of OCR.

CnC-OCR Evaluation

We evaluate CnC-OCR's performance and scaling on shared memory using the Cholesky kernel, and on distributed memory using the Smith-Waterman kernel. We compare the CnC-OCR versions of the kernels against the base OCR implementations, as well as Intel CnC implementations (iCnC). The iCnC implementations use the same codebase as the CnC-OCR kernels, but use a different layer of glue code to run on the MPI+TBB-based iCnC runtime. These experiments were run on the apps/master head revision from 2015-08-06 (0ac78b0).

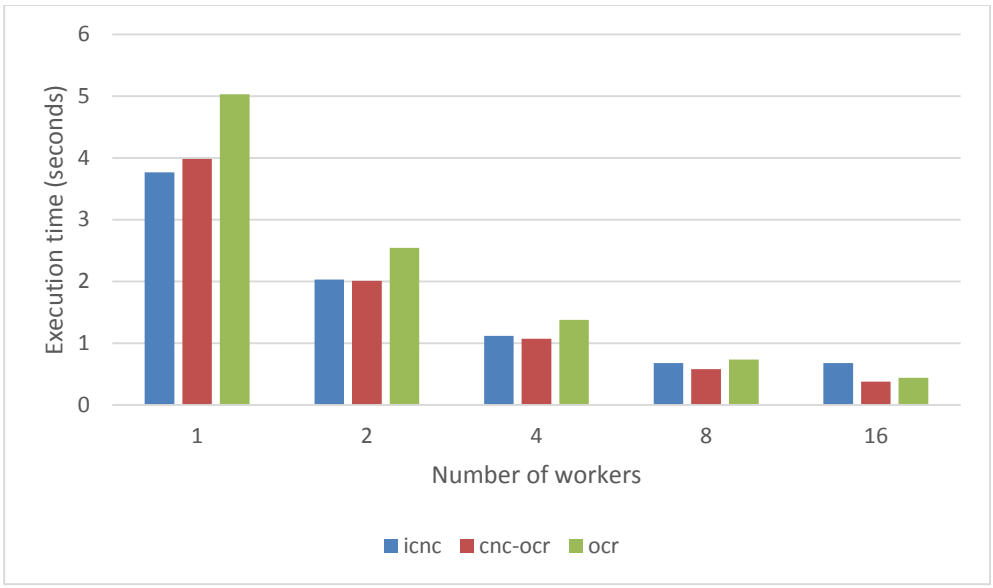


Figure 6.25: Single-node scaling with Cholesky, a 3000x3000 matrix with 50x50 tiles.

Figure 6.25 summarizes the single-node, shared-memory performance results with Cholesky. These were run on a single node on the X-Stack cluster at Intel. Since the base OCR version of Cholesky is structured somewhat differently than the CnC versions, it is difficult to make a direct comparison. However, we see that the CnC-OCR and base OCR kernels exhibit similar performance and scaling for 1 to 16 workers on the single 16-core node. Interestingly, the iCnC runtime appears to have a slightly higher overhead than the OCR runtime, and stops scaling at 8 cores. The sample C++ Cholesky implementation that is shipped with the iCnC distribution shows a similar performance and scaling trend to the version using our unified C codebase. The kernel was run on a 3000x3000 triangular input matrix, with tiles of size 50x50. These tiles were large enough to amortize the runtime overheads of OCR and the CnC layer on top; however, if too small a size of tiles is used (e.g., 20x20), then the runtime overheads become more obvious and the scaling degrades.

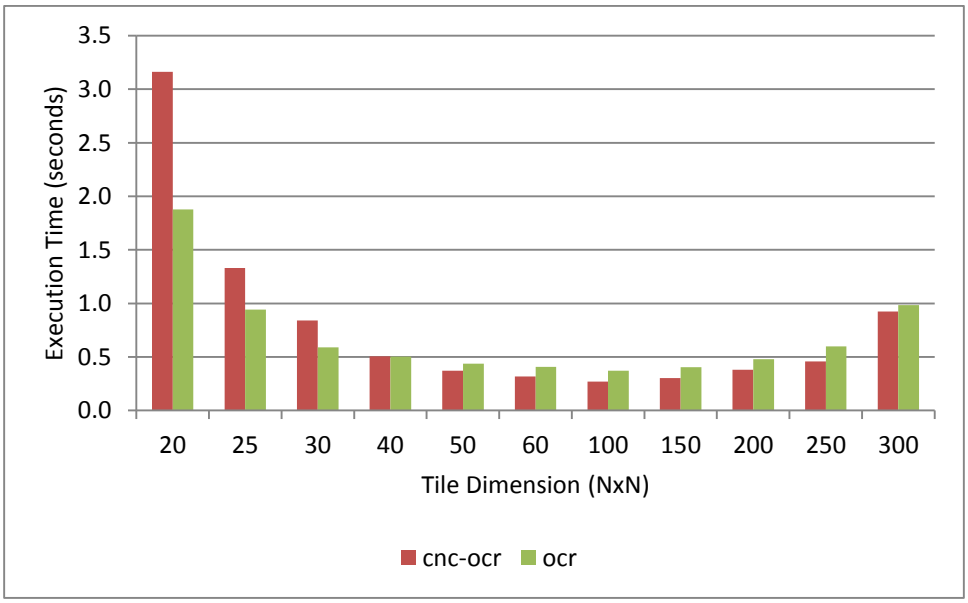


Figure 6.26: Cholesky calculation time for different tile sizes on a 3000x3000 matrix.

Figure 6.26 shows the trend for execution time relative to the specified tile size with 16 workers. The dominant computation step in the calculation (the *update* step) does a double-precision floating point multiply and add inside a triply-nested loop. For an N×N tile, the step does N^3 multiply operations. Therefore, for the timings shown in Figure 6.25—or the 50x50 tile size—the average EDT performed 125k multiply operations, and the average datablock was 2,500 elements, or 20kB per datablock. In contrast, if the tile size is set to 20x20, then there are only 8k multiply operations per step, and 3.2kB per datablock. This granularity is too small to mask the runtime overhead of EDT and datablock creation. The overhead for these small tasks is significantly larger for CnC-OCR version than for the OCR version due to the additional level of indirection (through the CnC item collections) for all input and output items.

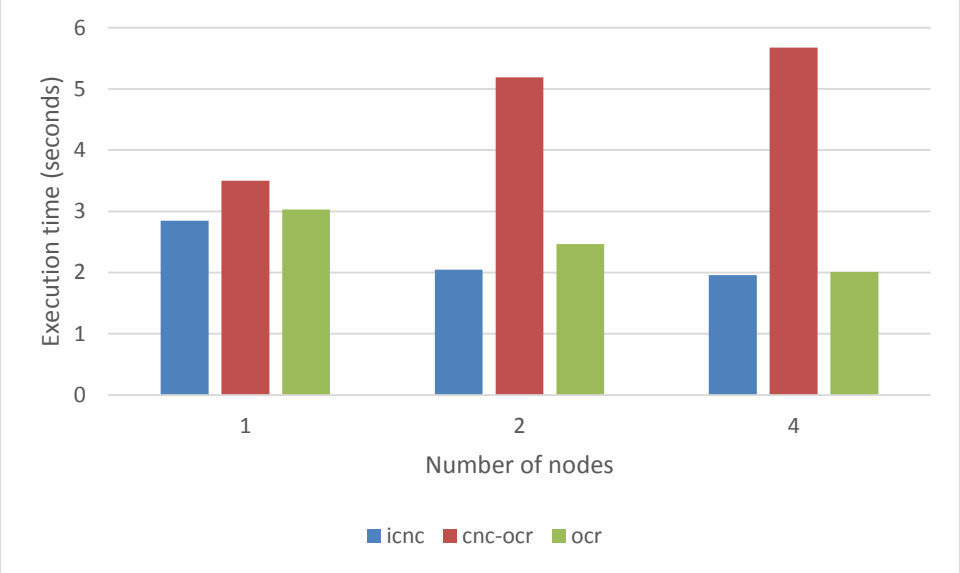


Figure 6.27: Distributed scaling for the Smith-Waterman kernel. The input sequences were each about 99k in length, and there were about 3000 total tiles in the sequence alignment matrix.

Figure 6.27 summarizes the distributed performance results with Smith-Waterman. Again, the OCR and CnC implementations are structured differently, so it is difficult to draw a direct comparison. However, while none of the implementations scaled well distributed, the CnC-OCR version exhibited significant negative scaling. We believe this is due to the poor performance of our distributed item collection data structure, which ends up being a scalability bottleneck. As mentioned, the introduction of labeled GUIDs in OCR can help avoid this bottleneck.

CnC-OCR applications run on FSim on multiple blocks, but we are unable to collect any meaningful statistics at this time.

	Cholesky	Smith-Waterman
Base OCR	492	314
CnC-OCR	171	164
% Reduction	65%	47%

Table 6.5: Comparison of Logical Lines of Code counts between the base OCR and CnC-OCR implementations of two kernels. The code counts were measured with the Unified Code Count tool.

Error! Reference source not found. shows a comparison in the logical lines of code required to implement the Cholesky and Smith-Waterman kernels in both CnC-OCR and base OCR. Although fewer lines of code is not necessarily a direct indicator of increased productivity, it is our experience that programming directly with the OCR API results in a significant amount of boilerplate code written by the application programmer. Additionally, the CnC-OCR graph translator tool generates a significant amount of the project scaffolding based off the application's graph specification, further reducing the need for the application programmer to deal with boilerplate code.

Recommendations and Future Directions

We believe that the task-based programming models such as HClib, CnC, and OCR are the most promising direction for delivering programmability, performance, and resilience on future exascale systems. Our current implementations attain competitive performance and scalability on shared-memory workloads with coarse-grained tasks. There is room for improvement in our implementations of the distributed memory workloads and of the handling of fine-grained tasks. This is a limitation of our current implementations rather than an inherent deficiency of the task-based programming model, and will be addressed as we continue work on this ecosystem.

Going forward, we believe that an emphasis on resiliency and dynamic adaptivity of the runtime is of critical importance. Some of the task creation and data block manipulation overheads in both CnC and OCR are the result of design decisions made to enable scalability of our programming models at the extreme scale. We will explore the effect of these design decisions on the high resilience and adaptability demands imposed by a true exascale machine.

We also believe that we need to continue to address programmer productivity when writing programs directly in OCR. As we have collaborated with our partners at the government labs, we have found that the transition from the more traditional parallel programming models (such as MPI and OpenMP) to OCR tends to be very difficult and error-prone. We need to provide a simpler, higher-level programming interfaces to ease the transition, as well as improved tools for debugging and performance analysis. These facilities will provide a smoother learning curve for domain experts transferring their existing software to OCR.

Products and Publications

Products:

- HClib. http://habanero-rice.github.io/hclib/
- CnC-OCR toolchain and examples. https://habanero.rice.edu/cnc-ocr/
- Available in the X-Stack Git repository under the hll/cnc directory.

CnC-related publications since 2012:

- The Tuning Language for Concurrent Collections. Kathleen Knobe, Michael G. Burke. CPC workshop, January 2012.
- Mapping a Data-Flow Programming Model onto Heterogeneous Platforms. Alina Sbirlea, Yi Zou, Zoran Budimlic, Jason Cong, Vivek Sarkar. LCTES conference, June 2012.
- Folding of Tagged Single Assignment Values for Memory-Efficient Parallelism. Dragos Sbirlea, Kathleen Knobe, Vivek Sarkar. Euro-Par conference, August 2012.
- The Flexible Preconditions Model for Macro-Dataflow Execution. Dragos Sbirlea, Alina Sbirlea, Kyle B. Wheeler, Vivek Sarkar. DFM workshop, September 2013.
- Bounded Memory Scheduling of Dynamic Task Graphs. Dragos Sbirlea, Zoran Budimlic, Vivek Sarkar. PACT conference, August 2014.
- DFGR: an Intermediate Graph Representation for Macro-Dataflow Programs. Alina Sbirlea, Louis-Noel Pouchet, Vivek Sarkar. DFM workshop, August 2014.
- Asynchronous Checkpoint/Restart for the Concurrent Collections Model. Nick Vrvilo. M.S. Thesis, August 2014.

- Memory and Communication Optimizations for Macro-dataflow Programs. Dragos Sbirlea. Ph.D. Thesis, May 2015.
- High-level Execution Models for Multicore Architectures. Alina Sbirlea. Ph.D. Thesis, September 2015 (expected).
- Polyhedral Optimizations for a Data-Flow Graph Language. Alina Sbirlea, Jun Shirako, Vivek Sarkar, LCPC workshop, August 2015.

Habanero-C and OCR-related publications since 2012:

- Integrating Asynchronous Task Parallelism with MPI. Sanjay Chatterjee, Sagnak Tasirlar, Zoran Budimlic, Vincent Cave, Milind Chabbi, Max Grossman, Vivek Sarkar. IPDPS conference, April 2013.
- Runtime Systems for Extreme Scale Platforms. Sanjay Chatterjee. Ph.D Thesis, December 2013.
- HabaneroUPC++: a Compiler-free PGAS Library. Vivek Kumar, Yili Zheng, Vincent Cave, Zoran Budimlic, Vivek Sarkar. PGAS conference, October 2014.
- Heterogeneous Habanero-C (H2C): A Portable Programming Model for Heterogeneous Processors. Deepak Majeti, Vivek Sarkar. PLC workshop. May 2015.
- Heterogeneous Work-stealing across CPU and DSP cores. Vivek Kumar, Alina Sbirlea, Ajay Jayaraj, Zoran Budimlic, Deepak Majeti, Vivek Sarkar. HPEC'15 conference. September 2015.
- Optimized Event-Driven Runtime Systems for Programmability and Performance. Sagnak Tasirlar. Ph.D. Thesis, October 2015 (expected).

Copies of all publications can be obtained from https://wiki.rice.edu/confluence/display/HABANERO/Publications

7. Reservoir Labs

By Muthu Baskaran, Benoit Meister, Tom Henretty, Sanket Tavarageri, Benoit Pradelle, Athanasios Konstantinidis, Preston Briggs, Richard Lethin

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. Reservoir Labs develops and delivers R-Stream, a source-to-source automatic parallelization and optimization tool targeted at a wide range of architectures including the Exascale architectures. Automatic optimizations in R-Stream are key to providing DOE with the best of both worlds: extreme scale hardware performance with high productivity in producing and sustaining software. 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 takes in loop codes written in stylized sequential C and produces a parallel version of the input program. Research conducted prior to and outside X-Stack has established R-Stream as a leading tool in providing advanced polyhedral optimization methods, with facilities for generating optimized code for multicore, GPGPU, and other hierarchical, heterogeneous architectures. R-Stream is notable for features that can transform programs to find more concurrency and locality, and for features that manage communications and memory hardware explicitly as a way of saving energy. These features were exploited in our X-Stack research efforts.

Challenge Focus

The accomplishments of research efforts in the X-Stack project, utilizing R-Stream, include demonstrating the ability to automatically map to a deep hierarchical architecture such as Traleika Glacier (TG) that has multiple levels and types of processing units and memories (local memory/scratchpad, DRAM, etc), the ability to parallelize to the asynchronous event-driven task (EDT) runtime model, and the ability to assist the runtime to spawn tasks/computations and create/manage data in a more scalable manner suited for Exascale execution. These accomplishments relate directly to multiple challenge areas of the X-Stack program: (1) automatic code generation and data management enables *high productivity*, (2) *energy efficiency* is improved through the ability to generate

different forms of efficient (minimal) communication (e.g. explicit data copies, DMA operations) and through the ability to *find more locality* in the context of *finding more concurrency*, 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.

In our research, we targeted R-Stream for x86 systems and the TG functional simulator – FSIM, and generated mapped programs in the form of Open Community Runtime (OCR) code.

Technical Approach

Automatic generation of OCR Code

We implemented 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 datablocks, respectively. These tiles are created after applying optimal polyhedral transformations. R-Stream also captures the dependence between tiles in a concise R-Stream internal representation. This tile dependence information (or task dependence information) is embedded in the generated code that dynamically sets the dependence between different EDTs (and datablocks) during execution. Figure 7.1 shows a code snippet that illustrates the non-trivial optimized OCR code generated by R-Stream.

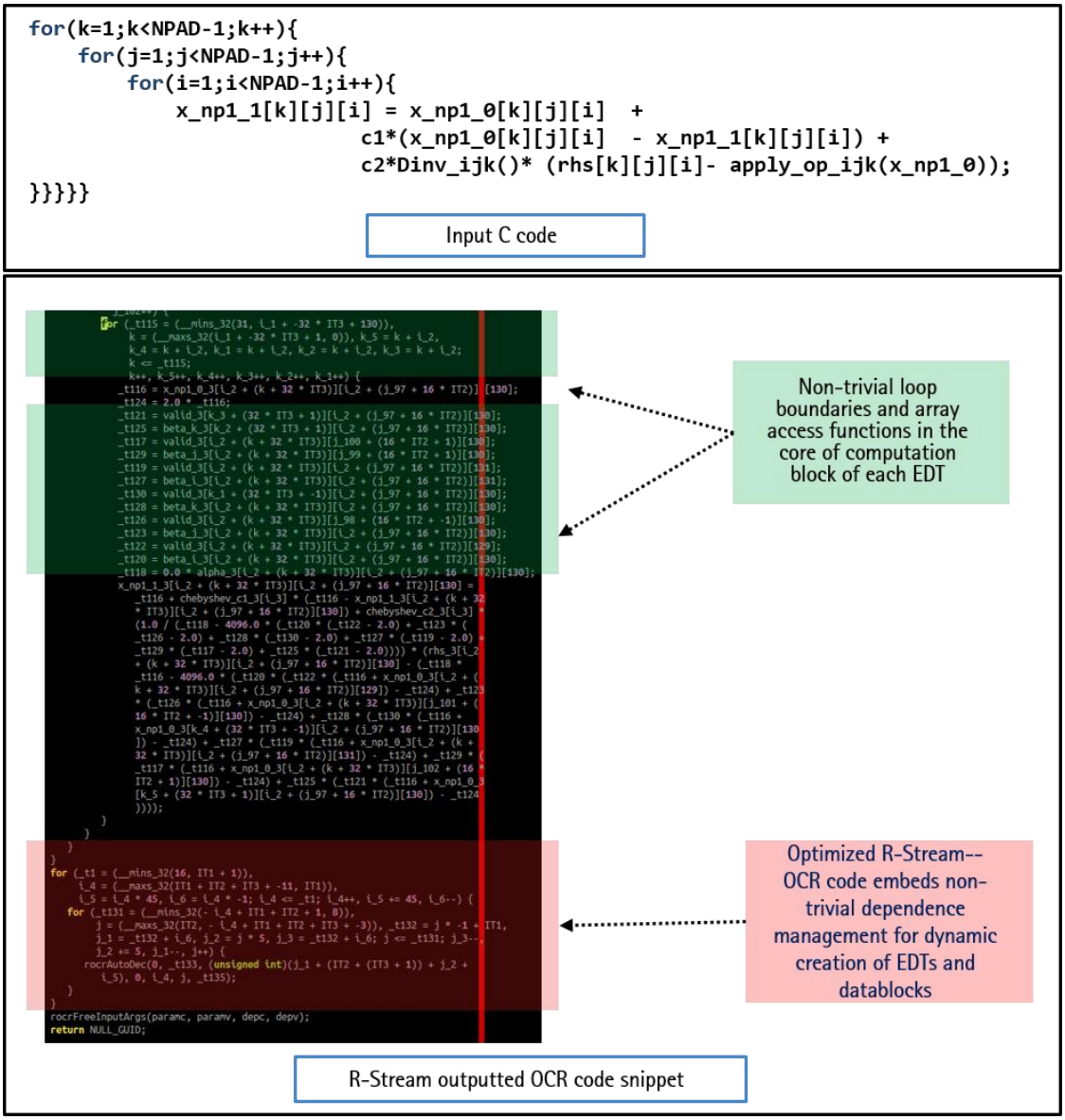


Figure 7.1: Code snippet (out of 2400+ lines of code) from R-Stream generated OCR code showing non-trivial loop boundaries and array access functions in the core of computation block of each EDT after applying polyhedral optimizations. Also, shown is the code embedded in each EDT for non-trivial management of dependence between EDTs and datablocks, and dynamic creation of EDTs and datablocks.

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 "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.

Scalable creation and management of datablocks

With research conducted prior to and outside of X-Stack, R-Stream had the ability 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 were exploited to generate datablocks for OCR code generation. The initial implementation involved creating one large datablock for each array, then create smaller datablocks within EDTs that fit in the local scratchpad (for e.g. XE scratchpad in TG architecture) based on the data accessed within an EDT, and generate communications between the datablocks.

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

Furthermore, to compliment the on-the-fly scalable creation of EDTs, we implemented the support for on-the-fly creation of datablocks. Creating all the datablocks that are needed during execution, at the beginning of the execution, incurs a huge "startup" overhead and it may not be always necessary. In R-Stream generated OCR code, a datablock is not created until it is needed for the first time by an EDT that is ready to execute and uses the datablock for its computation. R-Stream automatically identifies the dependence between different EDTs and the dependence between EDTs and datablocks, and automatically generates code for optimal on-the-fly EDT and datablock creation. R-Stream has a light-weight runtime layer that operates on top of OCR and handles these capabilities, namely, on- the-fly creation of EDTs and datablocks.

Communication Generation

R-Stream generates communications in different forms – 1) explicit copy loops transferring data between different arrays (or datablocks) and 2) bulk DMA communications between arrays (or datablocks). The specification of the target architecture is given to R-Stream through a "machine model" description. R-Stream generates a code version based on the specification. For example, for a cache memory level, it can generate code with or without the creation of local arrays (virtual scratchpad), and accordingly, with or without explicit communications (in the form of copy loops or bulk DMAs). For a scratchpad memory level, it generates explicit communications in the form of copy loops or bulk DMAs. R-Stream generates code to perform bulk DMA communications in TG architecture through calls to an R-Stream internal DMA wrapper over TG MemISA DMA instructions.

Supporting TG MemISA in R-Stream

We created a DMA runtime wrapper in R-Stream that provides a straightforward C interface to TG MemISA DMA instructions (previously available only through inline assembly language). We created the DMA wrapper for both the 32-bit (V4.0.8) and 64-bit (V4.1.0) TG ISA. We implemented the wrapper as strided DMA functions where both source and destination locations may have independent strides. We compiled and tested the code with the LLVM and binutils toolchain for the applicable ISA. We also implemented a test suite for the wrapper and ran it successfully on the 32-bit FSIM code. The test suite consists of six subtests that exercise features from simple one-dimensional data transfers to DMA where both input and output arrays have different strides.

Handling sparse computations

We extended R-Stream optimizations for handling irregular or sparse computations. R-Stream handles irregular computations with few user-inputted annotations (to describe an approximation of the irregularity). We extended the capability of R-Stream to exploit the sparse data structures, reduce the overhead in supporting the irregularity in computations, and improve data locality.

Results and Analysis

The results that we produced with R-Stream's automatic optimized OCR code generation capability are presented below. These results clearly highlight the performance and productivity benefits that R-Stream offers to the Exascale software stack. We used R-Stream v3.15.0.1 for our experiments.

Results on x86 target

We conducted an experimental study of R-Stream OCR code generation and optimization on the x86 target. We ran our experiments on a 48 core (96 thread) quad socket Intel Xeon (Ivy Bridge) server. We used the OCR version that was available in the public OCR code repository (https://xstack.exascale-tech.com/git/public/xstack.git (runtime/master branch commit # 845a754a7068b1aa748139e5a9c8468336eee310).

It is to be noted that we conducted our experiments on a shared memory system and not on a distributed memory system. The advanced features of R-Stream such as dynamic creation of EDTs and datablocks, and dynamic spawning of EDTs are supported through a thin runtime layer of R-Stream on top of OCR. This thin runtime layer has non-OCR-based race-avoidance mechanisms to enable the advanced features. 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 to enable the advanced features. 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 run optimized distributed OCR versions. Otherwise, we would have to resort to running non-optimized OCR codes on distributed memory nodes.

HPGMG

The HPGMG benchmark [HPGMGa, HPGMGb] consists of two different code bases that can be compiled and run independently. One is a finite element code; the other is a finite volume. Further, users must set a number of options at build time, including multigrid cycle, bottom solver, and smoother. Finally, actual problem size is specified at run time. We investigated configurations with one of the HPGMG authors, and found that modern GMG solvers can best be proxied with the finite volume code configured with V-cycles, a biconjugate gradient stabilized (BiCgStab) bottom solver and red-black Gauss-Seidel (GSRB) smoothers operating at each level of the multigrid code. Future GMG solvers are best proxied with finite volume code configured with full multigrid F-cycles, a BiCgStab bottom solver, and Chebyshev smoothers at each level.

We examined the timing of HPGMG benchmark and identified that the performance critical sections include smoothing, restriction, interpolation, and ghost zone exchange (as evident from Figure 7.2).

After identifying the smoother as the primary computation bottleneck area, we focused on examining the GSRB and Chebyshev smoothers in depth. We identified and benchmarked the existing hierarchical parallelism in the code (presented in Figure 7.3). Coarse grain parallelism was identified at the MPI layer where the problem domain was equally divided amongst ranks. Fine grain parallelism was identified where subsets of the per-rank set were assigned to individual OpenMP threads. The finest grain parallelism was found inside each box where subsets of the box were assigned to be computed by individual OpenMP threads.

```
                                 0          1          2          3          4          5          6
box dimension                 64^3       32^3       16^3        8^3        4^3        2^3        7^3      total
------------------       ---------- ---------- ---------- ---------- ---------- ---------- ---------- ----------
smooth                     1.245632   0.150392   0.019383   0.004413   0.002750   0.006761   0.000000   1.429331
residual                   0.322012   0.020849   0.002486   0.000716   0.000405   0.000928   0.000108   0.347506
applyOp                    0.000000   0.000000   0.000000   0.000000   0.000000   0.000000   0.000681   0.000681
BLAS1                      0.103295   0.008873   0.001553   0.000259   0.000494   0.001599   0.002615   0.118689
BLAS3                      0.000000   0.000000   0.000000   0.000000   0.000000   0.000000   0.000000   0.000000
Boundary Conditions        0.000000   0.000000   0.000000   0.000000   0.000000   0.000000   0.000000   0.000000
Restriction                0.025140   0.002997   0.000467   0.001272   0.010481   0.001163   0.000000   0.041521
  local restriction        0.024981   0.002866   0.000225   0.000182   0.000184   0.000071   0.000000   0.028509
Interpolation              0.045437   0.004343   0.002054   0.001658   0.002594   0.000562   0.000000   0.056648
  local interpolation      0.045284   0.004148   0.001052   0.000336   0.000188   0.000091   0.000000   0.051099
Ghost Zone Exchange        3.232224   1.409549   0.147795   0.041618   0.081498   0.008794   0.001440   4.922919
  local exchange           0.103664   0.007946   0.001020   0.000845   0.001074   0.000840   0.000242   0.115630
MPI_collectives            0.708028   0.000000   0.000000   0.000000   0.000000   0.000000   0.000254   0.708281
------------------       ---------- ---------- ---------- ---------- ---------- ---------- ---------- ----------
Total by level             5.674306   1.589918   0.172874   0.051655   0.101318   0.018587   0.005997   7.614655

   Total time in MGBuild      0.121208 seconds
   Total time in MGSolve      7.627352 seconds
      number of v-cycles            11
Bottom solver iterations            56

            Performance      1.179e+07 DOF/s


calculating error...
 h =  2.232142857142857e-03  ||error|| =  4.827992800680606e-09
```

Figure 7.2: Timing of different sections of HPGMG benchmark. It is evident that smooth and ghost zone exchange (communication) are dominant kernels.

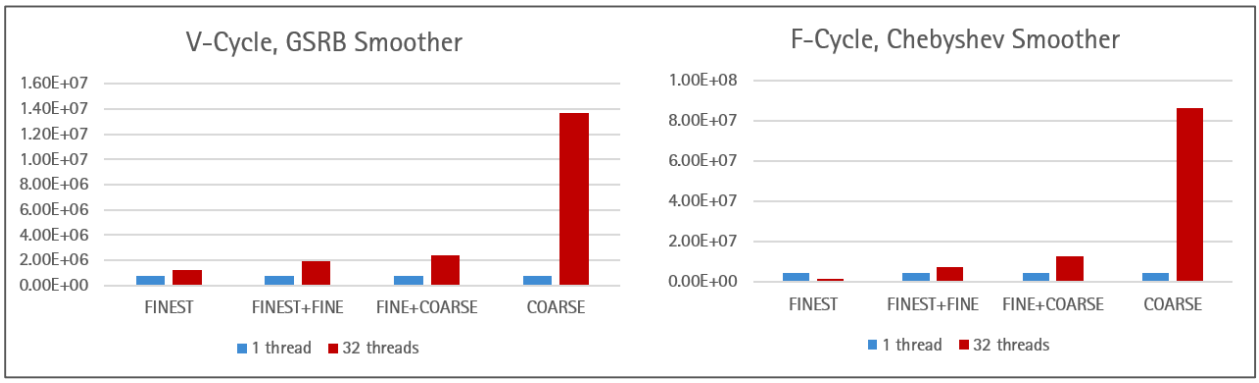


Figure 7.3: Performance of different granularity of parallel computations expressed in DOF/s (Degrees Of Freedom per second) - a HPGMG performance metric of choice. The problem size used was 8^3 grid of 64^3 boxes. Y-axis represents DOF/s; higher DOF/s implies better performance.

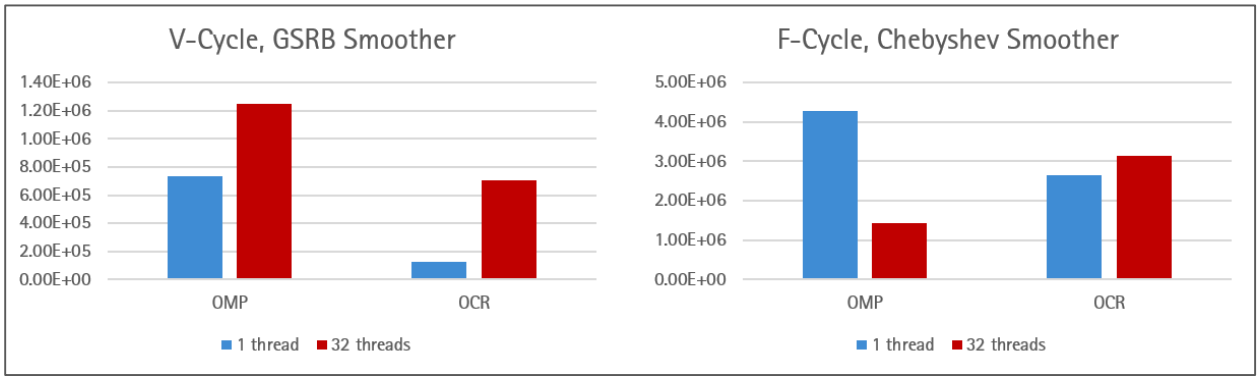


Figure 7.4: Performance of finest grain parallel region DOF/s. The problem size used was 8^3 grid of 64^3 boxes. Y-axis represents DOF/s; higher DOF/s implies better performance.

For our initial experiments, we chose to map the finest grain region of the GSRB and Chebyshev smoothers to enable automatic parallelization and optimization, and OCR code generation by the R-Stream compiler. We

compared the R-Stream generated OCR code against the OpenMP code for the finest grain parallel region. Initial results showed that OCR performance is better than the OpenMP performance for F-cycle and lower than the OpenMP performance for V-cycle, as illustrated in Figure 7.4. The possible reason, for performance of OCR code being only 60% that of OpenMP code for V-cycle, is that V-cycle has lots of fine-grid data-parallel tasks and OpenMP is fine-tuned for that granularity of data-parallelism.

We then 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 7.5.

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 datablocks, 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.

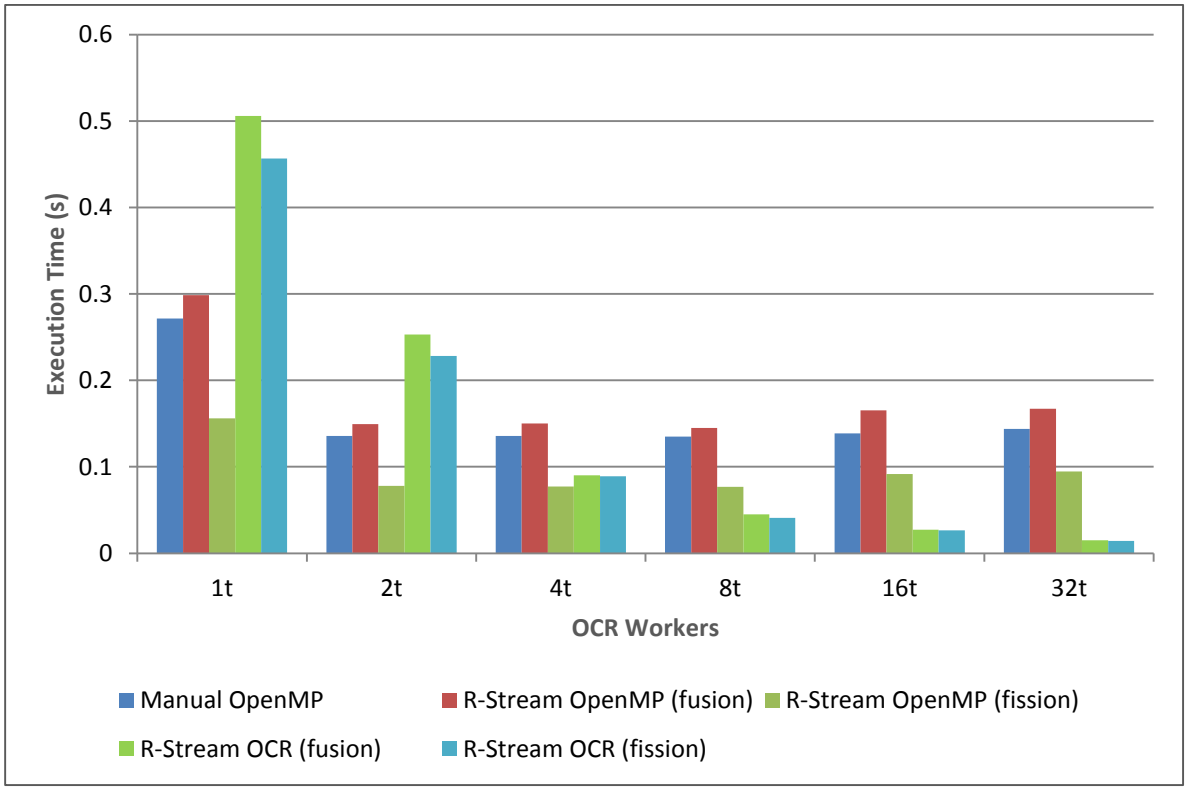


Figure 7.5: Chebyshev smoother performance on 64^3 array using multiple parallelization techniques for 1 to 32 threads. R-Stream OCR shows up to 5x performance increase vs. hand parallelized OpenMP.

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.

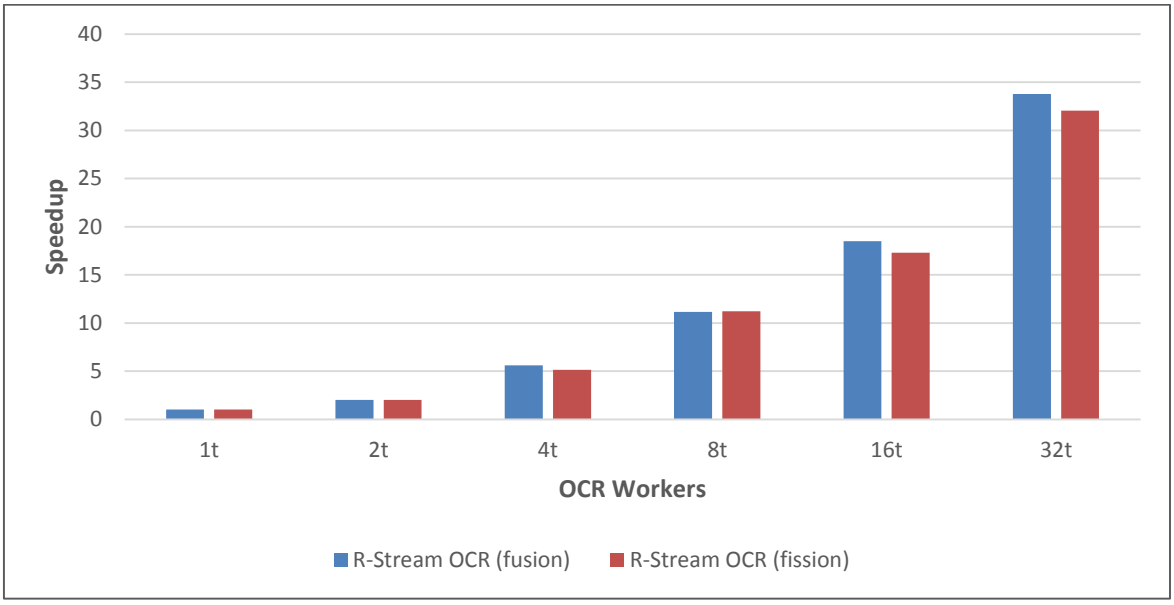


Figure 7.6: R-Stream OCR scalability. R-Stream OCR shows over 33x performance increase at 32 worker threads. The super-linear speedup is due to the fact the code is tuned for each worker count with a tile size that leads to better cache utilization and performance for that worker count.

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 7.6. We generated 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.

CoSP2: sparse matrix matrix multiply

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

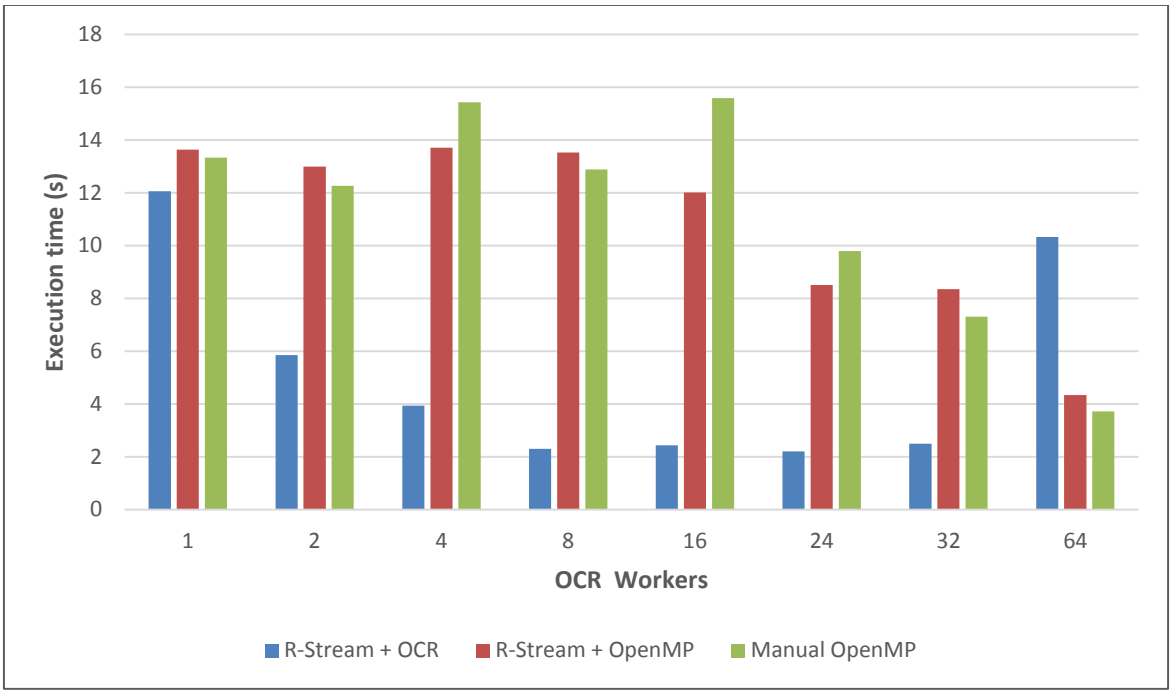


Figure 7.7: Performance of manual OpenMP version and R-Stream generated OpenMP and OCR versions of spmm code. R-Stream OCR version shows better performance (compared to other versions) up to 32 threads and then significantly drops performance for number of threads > 32. The R-Stream and manual OpenMP versions show similar performance.

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 7.7. The R-Stream OCR version exhibits the best performance, among all versions, up to 32 threads. The performance of the R-Stream OCR version significantly drops as the number of threads is increased beyond 32. The scalability limitation of OCR version has to be further investigated. The R-Stream and manual OpenMP versions perform almost similarly. One possible analysis is to investigate the effects of hyperthreading on OCR and OpenMP versions.

Kernels

We produced R-Stream optimized OCR codes for various benchmark kernels spanning multiple domains – linear algebra, stencils, etc. The characteristics of the benchmark kernels (size of data, number of computations, size of EDTs, number of floating point operations per EDT, etc.) are presented in Table 7.1. The OCR and OpenMP codes were generated with tile sizes selected by a static heuristic within R-Stream.

Kernel	Data size	Number of iterations	Number of EDTs	FP ops per EDT
FDTD-2D	1000^2	500 x 1000^2	148 K	48 K
GS-2D-5P	1024^2	256 x 1024^2	16 K	80 K
GS-3D-27P	256^3	256^4	256 K	6.75 M
JAC-2D-5P	1024^2	256 x 1024^2	16 K	80 K
JAC-2D-27P	256^3	256^4	256 K	6.75 M
STRSM	1500^2	1500^3	200 K	16 K
TRISOLV	1000^2	1000^3	60 K	16 K

Table 7.1: Benchmark kernel characteristics.

The results of the experiments on the benchmark kernels (comparing the GFLOPS performance achieved by OpenMP and OCR versions) are shown in Table 7.2. For kernels that present the opportunity to exploit asynchronous execution (such as the stencil kernels), the OCR version performed significantly better than the OpenMP version. For some of the linear algebra kernels (such as STRSM and TRISOLV) where there is a mix of parallel and permutable loops, the OpenMP version performed better. This emphasized the importance of tuning different parameters (such as EDT granularity) that affect the runtime performance to generate good OCR code.

Kernel	Version	Number of threads					
		1	2	4	8	16	32
FDTD-2D	OCR	1.20	2.31	4.34	8.13	13.96	17.14
	OpenMP	0.83	0.29	0.56	0.95	1.41	2.46
GS-2D-5P	OCR	0.89	1.72	3.23	5.90	10.38	15.04
	OpenMP	1.13	1.14	1.16	1.19	1.22	1.28
GS-3D-27P	OCR	1.82	3.56	6.90	12.95	24.71	37.53
	OpenMP	2.06	3.16	5.51	10.16	18.86	29.26
JAC-2D-5P	OCR	1.71	3.22	6.11	11.08	18.98	21.72
	OpenMP	0.92	0.92	0.91	1.13	1.40	2.19
JAC-3D-27P	OCR	2.41	4.70	8.94	16.72	31.66	34.48
	OpenMP	2.43	3.43	5.66	10.36	18.87	25.95
STRSM	OCR	4.49	7.58	11.76	17.62	15.95	11.72
	OpenMP	3.66	5.60	10.52	19.84	37.97	39.15
TRISOLV	OCR	1.64	2.95	4.89	7.63	7.55	5.29
	OpenMP	2.09	4.29	7.77	15.15	28.67	23.28

Table 7.2: Performance of R-Stream generated OpenMP and OCR versions of various kernels (in GFLOPS).

Results on FSIM

We conducted a brief experimental study of R-Stream OCR code generation and optimization on TG functional simulator FSIM. We used the OCR version that was available in the public OCR code repository (https://xstack.exascale-tech.com/git/public/xstack.git sandbox/shared/tgSchedulerAug2015 branch commit # ebb65abb8889d4edadecaca9fc6d9e0789fc85b3).

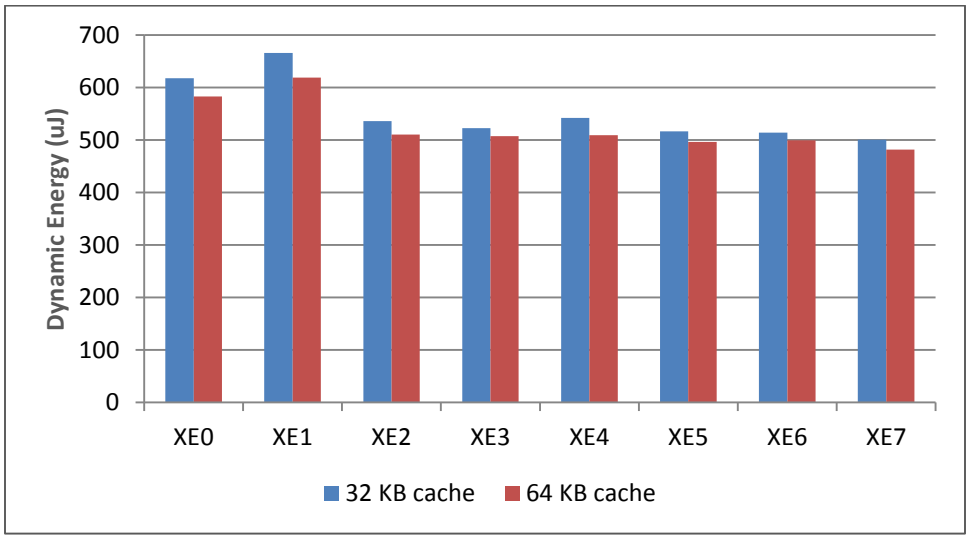


Figure 7.8: Dynamic energy consumption of R-Stream generated OCR code on FSIM for two different cache sizes – 32 KB and 64 KB.

We successfully ran and measured the energy consumption of R-Stream generated OCR codes of different kernels such as stencil kernels (GS-2D-5P, JAC-2D-5P) and HPGMG Chebyshev smoother kernel. We did not have reference OCR code for these kernels and hence did not perform any detailed comparative study. We changed the cache configuration (changed the cache size from 32KB to 64KB) and measured the energy consumption on the kernels. We saw a slight decrease in energy consumption for the 64KB cache simulations on the kernels we experimented. Figure 7.8 shows the dynamic energy consumption for the HPGMG Chebyshev smoother kernel for 32 KB and 64 KB cache simulations.

Low Level Compiler and Binary Utilities for TG Architecture

Reservoir Labs was responsible for developing a low level compiler and binary utilities including assembler, linker for the TG architecture through its various revisions (versions 4.0.6 through 4.1.0). We made numerous updates to the LLVM code generator (low level compiler) to support various versions of TG ISA and introduced optimizations to produce high quality code. We maintained the GNU Binutils which are a set of programming tools for creating and managing binary programs, object files, libraries, profile data, and assembly source code for the said revisions.

Below is the list of major tasks accomplished to maintain and enhance the LLVM code generator and GNU binutils.

- We ported the LLVM TG compiler from version 2.9 to 3.2
- We extended the 3.2 LLVM compiler to exploit the *.any* addressing mode, as a natural part of code generation (that is, without requiring programmer intervention). We modified the front end, adding keywords to support address-space extensions, replacing hard-to-maintain macro definitions and hard-coded integer values throughout the type-qualifier handling functions.
- To complement our work on supporting the .any flavor of instructions, we also added support for all new instructions into binutils. We also corrected all reported errors in binutils and worked to improve usability, particularly error reporting.
- We developed new optimizations in the LLVM code generator to make loading of immediate values more efficient and shift operations nimble. We enhanced the code generator and binutils to handle additional floating point division instructions added to the ISA.
- We introduced ISA modifications to more compactly encode integer signed and unsigned arithmetic.
- We updated binutils to incorporate queue-management (QMA) instructions that were added to the ISA in revision 4.0.8.
- We worked with Intel architects in defining the 64-bit ISA 4.1.0 --- our input was in the form of reviews for consistency, completeness, and compiler feasibility.
- We ported LLVM to generate code for 64 bit ISA 4.1.0. We made use of the enhanced ISA capability to emit codes that have fewer comparison operations that use the large register file effectively.
- We ported binutils to the 64-bit ISA version 4.1.0 and while doing so, we changed the file format used for object code, executable files, etc. from Elf32 to Elf64.

Recommendations and Future Directions

R-Stream positively supporting an "extended" runtime

One of the primary objectives of R-Stream as a high-level optimizing compiler is to have a positive interference with the underlying runtime to improve performance, energy efficiency, and reliability. There are multiple runtimes that are developed in the DOE-sponsored programs (OCR, Realm – Legion, etc.). In future, we expect that there will be an "extended" runtime that encompasses all the salient Exascale features of the different runtimes to provide an effective runtime system. The core R-Stream compiler features are developed in a generic (runtime-agnostic) manner and are not specifically limited to any runtime. The R-Stream backend code generation is (has to be) tied to a runtime and it supports multiple runtime backends (e.g. generating OCR APIs, SWARM APIs, etc.) with

the same core compiler optimizations. Hence R-Stream seamlessly fits in to support any Exascale (EDT-based) runtime.

R-Stream seamlessly supporting scalable deep hierarchy optimizations

Existing polyhedral compiler algorithms have scalability issues when they are used for generating code for deep hierarchy (e.g. more than three levels). The recent innovations in R-Stream compiler (developed partly outside of X-Stack program) have drastically improved R-Stream's scalability when optimizing deep loop nests. R-Stream exploits a novel approximate representation in which some of the polyhedral dimensions are hidden to the compiler during a part of the compilation. The approximate representation enables to perform most of the costly polyhedral compiler computations on simpler polyhedral data (while still maintaining the integrity of the polyhedral transformations), which significantly reduces the compilation time. In future, we will extend the existing hierarchical representation of R-Stream compiler to enable compilation for arbitrarily deep hierarchies. The extended hierarchical representation will also play a pivotal role in generating explicit communications and performing automatic memory management in deeply hierarchical platforms. As a result, memory and communications management will be transparent to the application developers.

Runtime-feedback-driven compiler optimizations

A key aspect of future optimizing compilers is the ability to do more dynamic optimizations, especially those that are driven by feedback from the underlying runtime. In future, we will extend R-Stream to perform dynamic runtime-feedback-driven optimizations.

R-Stream generating (EDT model-based versions of) communication-avoiding programs

Communication-avoiding programs have become increasingly important and relevant for modern architectures. There are a handful of programs for which the communication-minimal version has been implemented by hand (mostly from Prof. Demmel's group at UC Berkeley). An ideal solution for implementing communication-avoiding versions of programs is to auto-generate them using a high-level optimizing compiler. Reservoir is developing communication avoiding technologies in the DARPA PERFECT program. In future, we will work to add annotations to facilitate the programmer to provide hints about communication avoidance, and also explore ways to extend the compiler to generate a broader set of communication-avoiding schedules. The resulting technology will be a tool for DOE scientists to rapidly generate (EDT model-based) code for Traleika Glacier that is communication avoiding and that exploits other hardware features, e.g., energy proportionality and nimble dynamic voltage controls, that are present in TG hardware.

R-Stream for in-situ UQ

In the X-Stack program, Reservoir invented proprietary approaches to implementing in-situ UQ with PDE solvers through advanced scheduling and fusion transformations in the polyhedral manner in a way that minimize communications and memory footprint. In future, we will extend existing general purpose language or domain specific languages for programming, in a way that reduce arbitrary semantic side effects that constrain the polyhedral compiler from performing this fusion and in fact guide the compiler to this fusion. We will also explore utilizing scheduling hints (hierarchical affinity) that can guide the runtime to achieve this fusion and lock into it dynamically.

R-Stream optimizations for novel key algorithms

Reservoir is developing novel low-power algorithms in the DARPA PERFECT program. These algorithms are critical to various DOD and DOE applications, especially in the context of improving energy efficiency when such applications are run on Exascale computing systems or low-power embedded systems. These algorithms include low-power compressive sensing algorithms, spectral support preconditioning and nearly-linear time solvers, low-power and low-communication FFTs using Fast Multipole Methods (FMM). In future, we will apply/extend R-Stream optimizations to automatically generate energy-efficient codes for these algorithms.

Products

Tools

R-Stream source-to-source optimizing compiler

R-Stream is not an open- source software. To get R-Stream, please contact Reservoir Labs using the URL: https://www.reservoir.com/company/contact/. Reservoir Labs will set up user accounts so that R-Stream binary can be downloaded directly from the Reservoir web site. People downloading R-Stream will have to use the click commercial license for R-Stream before download. Additional special licensing options are available for government and academic organizations.

Papers

Nicolas Vasilache, Muthu Baskaran, Tom Henretty, Benoit Meister, M. Harper Langston, Sanket Tavarageri, and Richard Lethin, "A Tale of Three Runtimes." arXiv:1409.1914v1

Code

The following benchmarks/apps codes are available in the public OCR code repository (https://xstack.exascale-tech.com/git/public/xstack.git)

1. HPGMG (available in the folder : apps/hpgmg/refactored/rocr/reservoir)
2. OCR codes automatically generated by R-Stream for sample benchmarks (available in the folder: hll/rstream/examples)

Bibliography

[HPGMGa] LBNL tech report LBNL-6630E: https://crd.lbl.gov/assets/pubs_presos/hpgmg.pdf

[HPGMGb] Adams, Mark, Jed Brown, John Shalf, Brian Van Straalen, Erich Strohmaier, and Sam Williams. 2014. "HPGMG 1.0: A Benchmark for Ranking High Performance Computing Systems". United States. doi:10.2172/1131029.

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

8. Intel's Distributed Runtime OCR

by Romain Cledat (Intel), Bala Seshasayee (Intel), Sanjay Chatterjee (Intel), Min Lee (Intel), Vincent Cave (Rice Univ.)

Over the course of the XStack project, we developed both a specification for OCR, which can be downloaded here: https://xstack.exascale-tech.com/git/public?p=xstack.git;a=blob;f=ocr/spec/ocr-1.0.1.pdf;h=1450ba3cde23b0b2c3b2a34203c003217e0ab9b4;hb=8e2250a761e7c42d32a5618583fe379ec429b66, and a reference implementation. We had several goals when developing the reference implementation for OCR:

- Modular: One of the main goals of the XStack project is to develop a community around a common standard for a future exascale programing model. The OCR reference implementation is therefore very modular by design and, although this introduces additional overhead, allows various groups to work on different components (schedulers, allocators, etc.) independently while still maintaining compatibility with other components.
- Multi-platform: Although the initial primary goal of OCR was to provide a programming environment for the Traleika Glacier (TG) architecture, we quickly realized that OCR should also be able to stand on its own and function on existing architectures. The modularity previously mentioned also allows us to easily

support multiple platforms and maximize code reuse. We currently support x86, distributed clusters (over MPI or gasnet), TG and an emulation of TG on x86 (still a work-in-progress).

- Compliant: Our reference implementation is compliant with the OCR specification.

As such, our reference implementation can be viewed as a framework to aid in the development of runtimes that implement the OCR specification.

Our OCR reference implementation is currently in active development. The interested reader can track progress on our GIT: https://xstack.exascale-tech.com/git/public/xstack.git and get more detailed information on our Wiki: https://xstack.exascale-tech.com/wiki.

In this section, we describe some key API features; highlight some of the technology developed during the project and present results on the various platforms for our reference implementation.

Key user-facing API features

Hints

The OCR hint framework is intended for runtime developers to explore hints that can be useful for creating adaptive runtime algorithms, either through the addition of new hints or by using existing ones. The framework unifies the handling of two kinds of hints: user specified hints and system generated ones. While the user hints are specified as part of the application program, system generated hints are part of the feedback generated by the runtime introspecting its own behavior. Currently, there are four types of hints: event-driven task (EDT) hints, datablock hints, event hints and group hints. OCR groups are logical entities to which multiple OCR objects can be associated. Providing a hint for a group will help the runtime guide the scheduling of these OCR objects. Group hints can be applied either to each individual object in the group or to the group as a whole. An example of a group hint is “distribute the EDTs within this group” or, conversely, “keep these EDTs close together.

Hints in OCR use a special type called *ocrHint_t*. A variable of this type cannot be shared across EDTs: it must be declared, initialized and used all within the same EDT. To use an *ocrHint*_t variable, the user can set multiple hint properties and then apply them to an OCR object whose GUID is known. It is also possible for the user to read the hints that are set on a specific OCR object and modify them. The hints API is described in Section B1 of the OCR specification. As an example, we show how hints can be added to an OCR program solving a tiled Cholesky factorization problem:

```
//Here we set hints on the EDT template for the various tasks:
//In this example, we show how to set the default affinity of all EDTs
//being generated out of these EDT templates to the DB that is passed
//in to slot 0. That is why we set the value of the hint property
//OCR_HINT_EDT_SLOT_MAX_ACCESS to be 0;

ocrHint_t hintVar;
ocrHintInit(&hintVar, OCR_HINT_EDT_T);
if (ocrSetHintValue(&hintVar, OCR_HINT_EDT_SLOT_MAX_ACCESS, 0 /*slot number hint*/) ==
0) {
    ocrSetHint(templateSeq, &hintVar);
    ocrSetHint(templateTrisolve, &hintVar);
    ocrSetHint(templateUpdateNonDiag, &hintVar);
    ocrSetHint(templateUpdate, &hintVar);
}
```

To quantify the effect of this hint, we conducted an experiment to measure spatial locality of tasks scheduled by the runtime with and without the hint. In this experiment, we define a dominant DB for every task as the DB that is accessed most by that task. Now, if a task that accesses a dominant DB executes at the same location where the

DB was last accessed, then we count that as a DB Spatial Hit (DSH). Else, that task would count a DB Spatial Miss (DSM). The locality metric, known as SDR (Spatial DB Reuse), is therefore defined as: $\mathrm{SDR} = \frac{\mathrm{DSH}}{DSH+DSM}$. For tiled Cholesky of size 4000x4000, using hints in a scheduling heuristic, called MAP, improves spatial locality over the base heuristic by 4x, as shown in Figure 8.28.

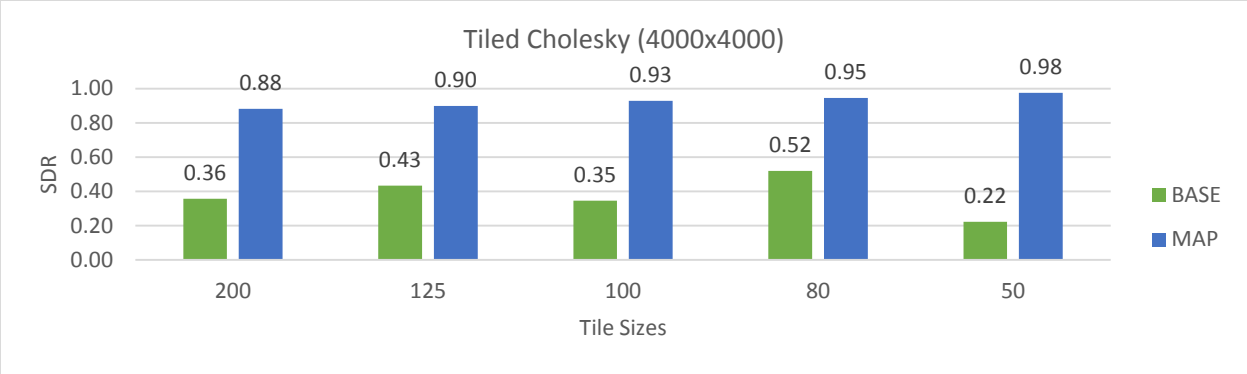


Figure 8.28 Improvements in data locality (measured as the percentage of hits to the dominant datablock) when using the hint framework

Labeled GUIDs

Labeled GUIDs allow the programmer to deterministically associate a user-defined index value with an opaque GUID thereby allowing two distinct EDTs to **agree** on the GUID of a given object without necessarily sharing this information. Concretely, the API (described in Section B2 of the OCR specification) provides a mechanism for reserving a certain number of GUIDs for a particular purpose (sticky events being the most common) and getting a single GUID back to represent this ensemble of GUIDs. This GUID can then be passed down to multiple EDTs. As a specific use case, if two EDTs what to communicate over a sticky event, **both** EDTs extract the indexed GUID and make it into a STICKY event (only one of them will succeed in creating the event, the other can just use the GUID). The receiver creates a new EDT with the event as a dependence while the sender satisfies the event with the appropriate datablock. For good hygiene, the receiver should destroy the event.

In addition to simplifying the initialization of an OCR application, labeled GUIDs offer another significant advantage. Without labeled GUIDs, each communication requires a new event. These events must be created and sent along with the data. This pollutes the application's data-structures. Labeled GUIDs make it possible to **reuse** the same GUID the next time you need to communicate thereby eliminating the need to pass GUIDs around (you always know which one to use). It is, however, necessary to ensure that the receiver has destroyed the event before the sender tries to recreate it. In many applications (including the 1D stencil), it is sufficient to double buffer the GUIDs, using a particular GUID only every other iteration.

For a 3D 27-point stencil calculation using Labeled GUIDs only the GUID of the list needs to be created and passed before going parallel. Without labeled GUIDs, 27 events must be created for each parallel task and carefully placed in the correct datablocks before going parallel. Figure 8.29 captures this for different sizes of 3D grids.

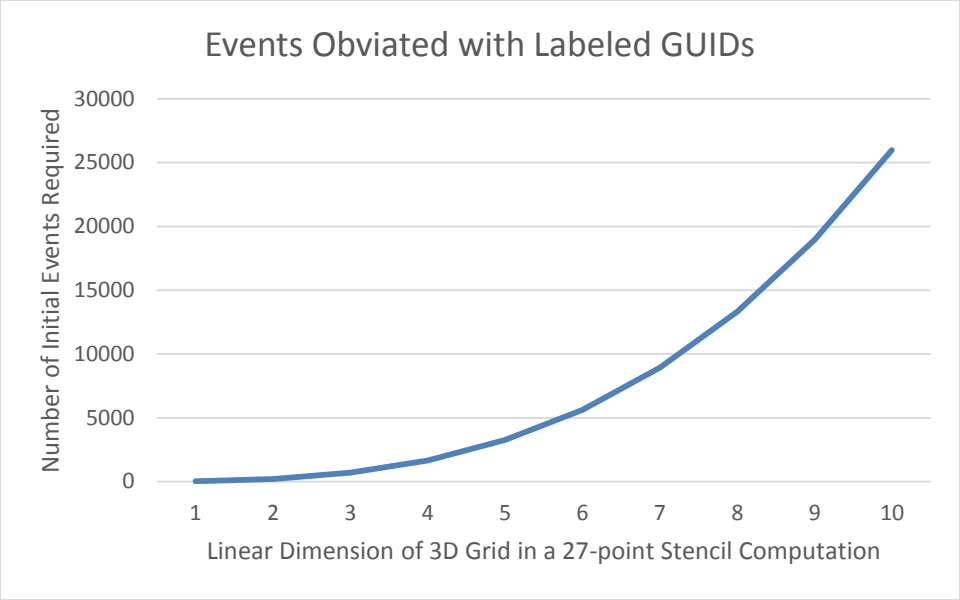


Figure 8.29 Usefulness of labeled GUIDs

Key technologies

This section describes a few of the key technologies in OCR.

Basic runtime design

As mentioned in the introduction, OCR is built around modules. The main modules are:

- Allocation: Our implementation strives to be entirely self-contained and rely on as few external platform-specific libraries as possible and therefore provides its own memory allocation routines
- Scheduling: The main function of OCR is to schedule tasks (EDTs) and place datablocks in the various memories (in the TG architecture, in the scratchpads). Our implementation provides a flexible and pluggable scheduling framework as well as several basic scheduling heuristics.
- Naming: All objects in OCR are identified by a globally unique ID (GUID). OCR provides modules to deal with the naming and resolution of those names.
- Low-level platform: Several modules are present to support the various backends OCR runs on. Specifically, three low level modules abstract away memory resources, computation resources and a communication framework between compute and memory resources.

The interaction between the various instances of these modules is managed by what we refer to as a **policy domain**. Conceptually, a policy domain defines the heuristics and policies for a part of the machine. This allows OCR to support heterogeneous architectures (which may require differing policies for different hardware) and scale better. Our implementation currently limits a policy domain to a single address space.

For our distributed implementation, a policy domain encapsulates a node (equivalent to a rank). In our TG implementation, each CE and each XE is an individual policy domain. The division could be done differently but this enabled us to make the XEs “dumb” and to locate all the smarts in the CEs.

Allocator

We developed a new allocator, called “quick allocator”, designed to meet the requirements of fast allocation and effective use of the TG memory space, namely scratchpads and non-coherent caches. It is based on the TLSF

algorithm[6] with many improvements and optimizations such as the use of concurrent heaps and private heaps. Private heaps are used for L1 scratchpads and are fast due to an absence of locking. Concurrent heaps, on the other hand, allow concurrent accesses to the central, larger, heap. Our concurrent heaps employ fine-grained locking which reduces contention to cases where the same free list is requested by multiple consumers. Finally, it uses a per-thread caching scheme to handle common cases such as small objects.

Designed for hierarchical memories

To support the TG memory model, initially the memory at each level had its own heap (bookkeeping metadata) to manage it and allocation calls were repeatedly called for each level in order L1, L2, L3, etc. until one succeeded. This technique had a lot of overhead (especially if the allocation only succeeded in L5). With the quick allocator, the free-lists for the various scratchpads are dealt with together. In other words, blocks from all L1 scratchpads are placed at the head of the list, and those from L2 scratchpads follow, and so on. Naturally, the search order is implied by this order. This way, this combined free-list can deal with blocks from multiple memory levels and the search for the suitable-sized blocks is simplified. Figure 8.3 shows a graphical representation of this scheme.

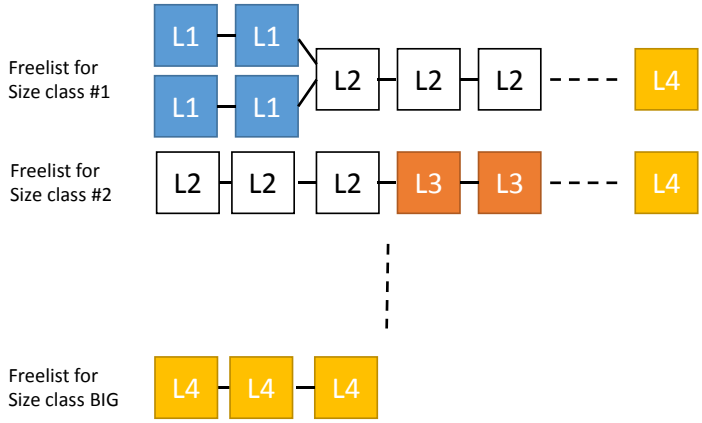
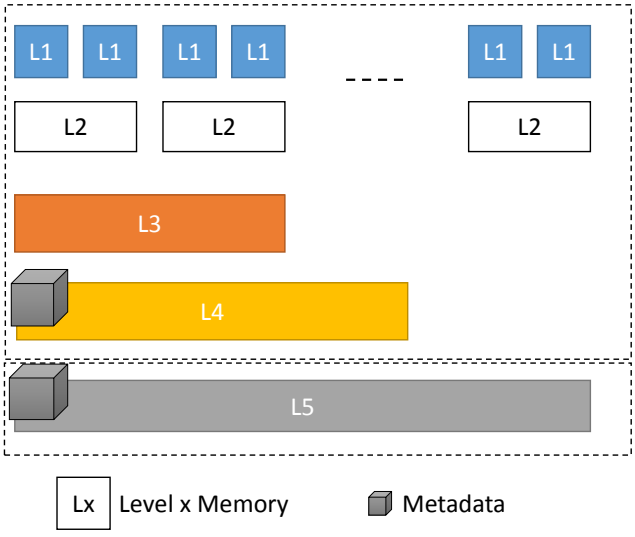


Figure 8.3 Combined free-lists to support memory hierarchy



[6] A constant-time dynamic storage allocator for real-time systems. Miguel Masmano, Ismael Ripoll, et al. Real-Time Systems. Volume 40, Number2 / Nov 2008. Pp 149-179

Figure 8.4 Supporting scratchpads and DRAM

Meanwhile, DRAM needs to be handled with care because DRAM has several different characteristics to consider. This includes its incoherent cache model and false sharing. For this, the DRAM heap has a free-list management that is different from that of the scratchpads. In effect, we now have two global heaps: one for scratchpads, and one for DRAM. To fully support the distinct coherence domains that the TG architecture has, quick allocator uses separate free-lists for each coherency domains. This approach avoids wasting space (as opposed to an approach based on memory alignment) and fragmenting memory (using separate heaps). Each domain has its own free-list and there is one 'safe-for-all-domains' list. The allocation tries the domain-specific list first, then fallbacks to the safe list. Blocks may go to safe list if covered or spanned cache lines are evicted or flushed from caches. Figure 30 illustrates this design.

Evaluation

In simple experiments, when compared to DLmalloc, which is the default allocator in glibc, quick allocator shows up to 20% speedup for common cases (small size objects). Figure 8.31 and Figure 8.32 show these results.

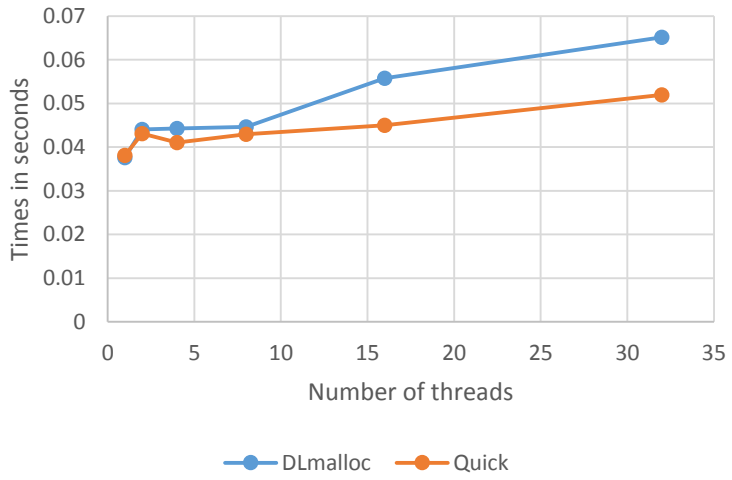


Figure 8.31 Allocation of fixed small sized objects (36 bytes)

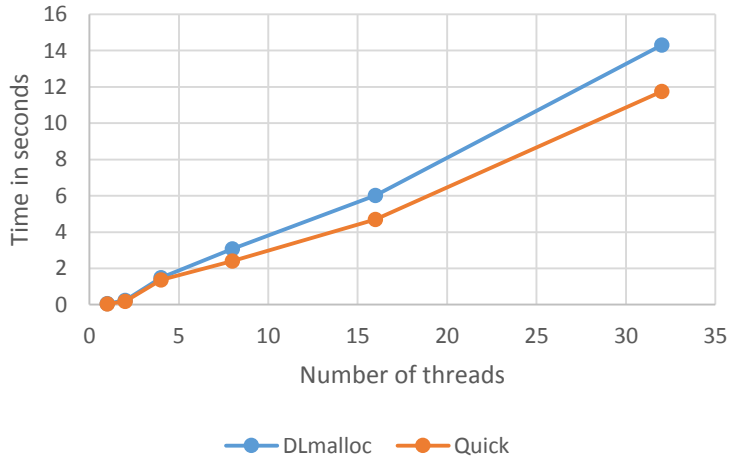


Figure 8.32 Allocation of random sized objects (up to 1 kB)

Scheduler

The scheduler is one of the key runtime modules in OCR that decides *when* and *where* a task is executed and data is placed. In the future, it will also have a say in *where* data is moved. Each dynamic instance of a policy domain contains a scheduler instance inside it. A scheduler is structured using multiple distinct sub-modules described below:

Scheduler (driver)

This module interfaces with the policy domain and thereby with the rest of the runtime. It also acts as the top-level module that drives every other module in the scheduler.

Scheduler objects

Scheduler objects are collectively all the OCR object instances and data structures managed by the scheduler. Scheduler objects are defined as a hierarchical composition of other scheduler objects. The simplest (primitive) kind of scheduler objects are OCR objects created by the user (datablocks, EDTs…). Each scheduler maintains a root scheduler object describing the entire state known to that scheduler.

Scheduler heuristics

Scheduler heuristics manage the insertion and removal of scheduler objects into and from the root scheduler object. A scheduler can be configured with one or more heuristics. Each heuristic is used to optimize costs for a specific objective function. If there are multiple heuristics in a scheduler module, the scheduler driver chooses which heuristic to invoke at runtime. Different heuristics can be invoked at different times in a program depending on the system state. All scheduler heuristics have to work on the same root object that the scheduler was configured with.

The scheduler driver's interface in OCR is defined by the type of interaction done with the other modules in the runtime. There are 4 kinds of interface operations defined, as shown in Figure 8.33.

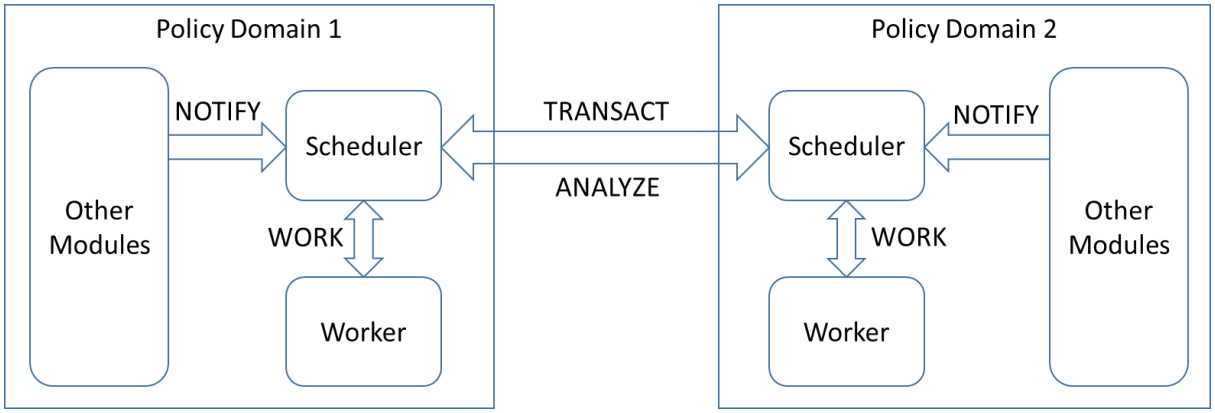


Figure 8.33 Operations defined for the OCR scheduler: Notify, Work, Analyze and Transact.

Runlevels: platform independent bring up and tear down

OCR uses the concept of runlevels to maintain a uniform startup and shutdown sequence across all platforms, with clear boundaries for runlevel transitions. This makes it easier for a runtime developer to port OCR to a new platform because it is very clear when modules are expected to be up and running. Table 8.1 illustrates the various runlevels and the services supported at each runlevel.

Runlevel	Startup	Shutdown

CONFIG_PARSE	Configuration parsed, runtime structures created	Platform-specific cleanup
NETWORK_OK	Communication capabilities between policy domains (PDs) operational	Inter-PD communication is torn-down
PD_OK	Each PD initializes its structures	PD shuts down its structures
MEMORY_OK	Memory allocators are operational	Destroy memory allocators, identify leaks if any
GUID_OK	Global naming is functional	Release GUID providers' resources
COMPUTE_OK	Compute resources initialized	Join all compute resources so that only one is actively running
USER_OK	Startup complete, transfer execution to *mainEDT()*	User code has been exited (via *ocrShutdown()*)

Table 8.1

Visualization tools

Efforts have been made to provide OCR application developers with a lightweight way to visually represent the behavior of their applications. This allows developers to identify inefficiencies and bugs in their apps, analyze, re-factor, and immediately see how their changes compare with previous iterations. These tools rely on post-processing debug logs, which can be collected via runtime configuration options prior to running an application. As its own entity, debug logs provide extensive insight into how the runtime interacts with the application. Having a tool-set that post-processes these logs allows developers to pinpoint and fix undesirable issues, both on the application side, and the runtime side.

Timeline: 2D representation of EDT execution on a per-worker (thread) basis.

Figure 8.8 represents the Fibonacci kernel, targeting the x86-mpi distributed OCR platform on two nodes. Each row represents a worker, and each colored block, an EDT. Each color is unique to the specific EDT type.

Figure 8.34 Execution of the fibonacci benchmark on two nodes with 8 workers each

Flowgraph: 2D representation of EDT execution as a complete DAG.

Figure 8.9 represents the Fibonacci kernel, targeting the x86 platform. EDTs, events, and datablocks are represented as nodes, and satisfactions, as directed edges. The critical path is highlighted in green.

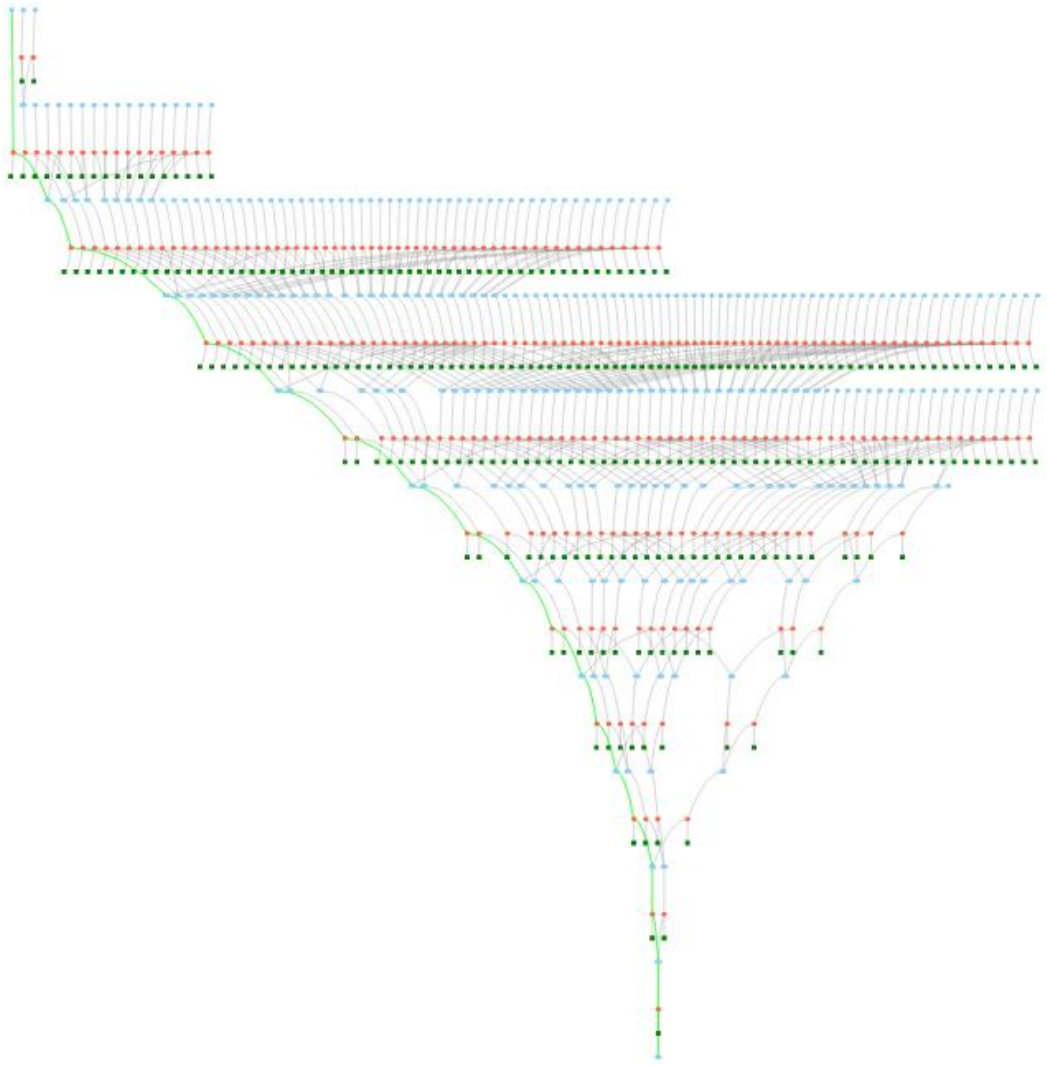


Figure 8.35 Critical path (in green) of the fibonacci benchmark

Implementation details

OCR implementation on TG

On TG, OCR largely uses the common platform-neutral codebase, except for the policy domain and the communications layers. There is a clear separation of concerns, with the application entirely running on XEs and the runtime running on CEs. To better aid manageability, OCR creates a single policy domain for each core in the TG architecture. Further, the policy domains on XEs manage only their local scratchpads, with the higher levels of memory, up to the DRAM, all managed by CEs. Each XE policy domain directly communicates only with the CE in its

block and no other cores. The CE policy domains are arranged in an implicit hierarchy mirroring the TG architecture, with each CE directly communicating only with its parent, children or siblings.

The TG implementation also differs in how OCR is bootstrapped, since the lack of an operating system means that even basic I/O services are absent. This means that the usual method of creating modules during startup based on the indicated configuration file would not work (see prior paragraph on runlevels). Instead, the startup occurs in two phases – in the first phase that runs on an x86 platform, the configuration file is used to create the various modules and their relationships to the policy domain, and these are saved in an external file (for each of CE & XE). In the second phase, during startup of the TG machine, this binary image of the modules is copied to the core's scratchpads directly by the bootstrap code, and execution is then transferred to OCR. The common data representation among the ABIs of CE, XE & x86 allows for this freedom. This design allows us to bring the flexibility of file-based module configuration to a platform lacking traditional OS services, such as TG.

OCR implementation on x86 clusters

The OCR programming API built on top of the OCR execution model is amenable to seamlessly execute on distinct platforms without requiring source code modifications. The OCR concept of a Global Unique IDentifier (GUID) and the OCR memory-model are key to supporting this. Distributed OCR provides a GUID implementation that guarantees GUIDs to be unique across a distributed system as well as minimize communications by carrying information as part of the GUID value. The memory-model is enforced by turning relevant OCR API calls into synchronous communications whenever necessary. The implementation of distributed OCR leverages the OCR module architecture which promotes customization, extensibility and code reuse. First, it relies on a number of pre-existing modules that are oblivious to the distributed setting the runtime operates in. Second, the distributed logic is completely decoupled from the underlying communication library, which allows for easy integration and substitution of communication libraries. The implementation currently supports MPI and GASNET as communication libraries. Figure 8.10 and Figure 8.11 respectively show throughput results (in operations per second) for OCR on MPI and OCR on GASNet. The benchmark creates and executes EDTs issued locally and remotely across two machines of a cluster. The results highlight the benefit of using Deferred-EDT allowing for asynchronous creation and increase overlapping opportunities. Ongoing work includes relaxing blocking communications while preserving the memory-model semantic, providing a framework for distributed OCR objects to allow for customization.

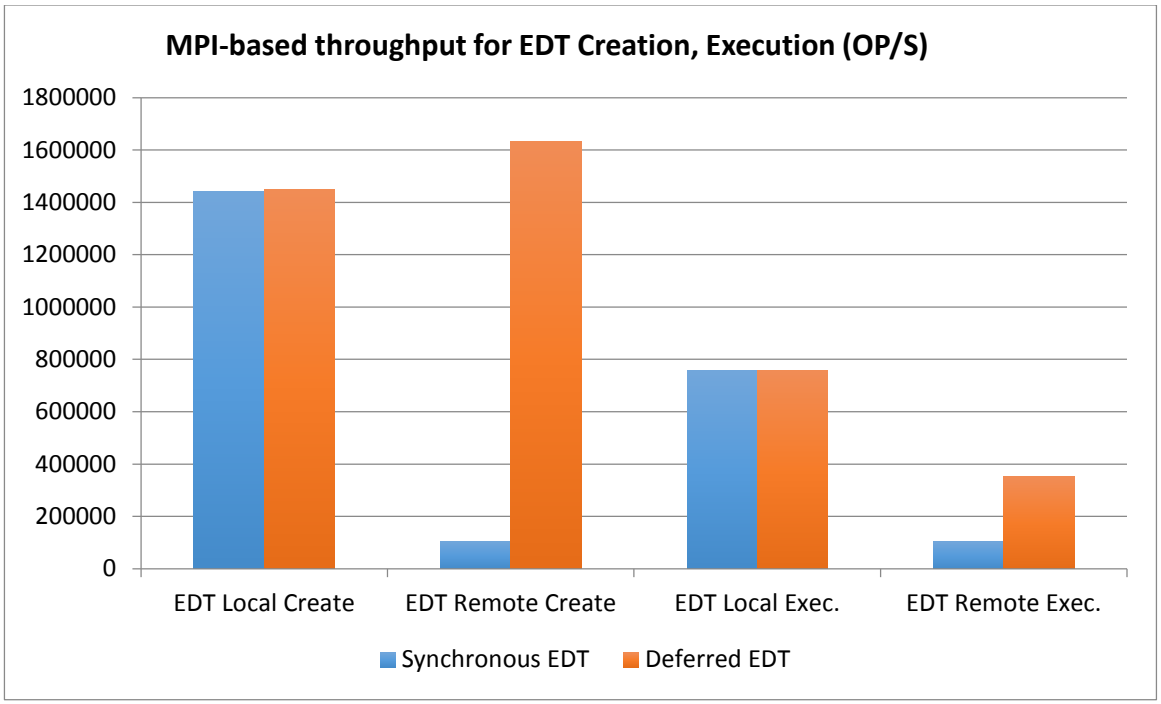


Figure 8.36: MPI-based throughput for EDT creation and execution in operations per seconds

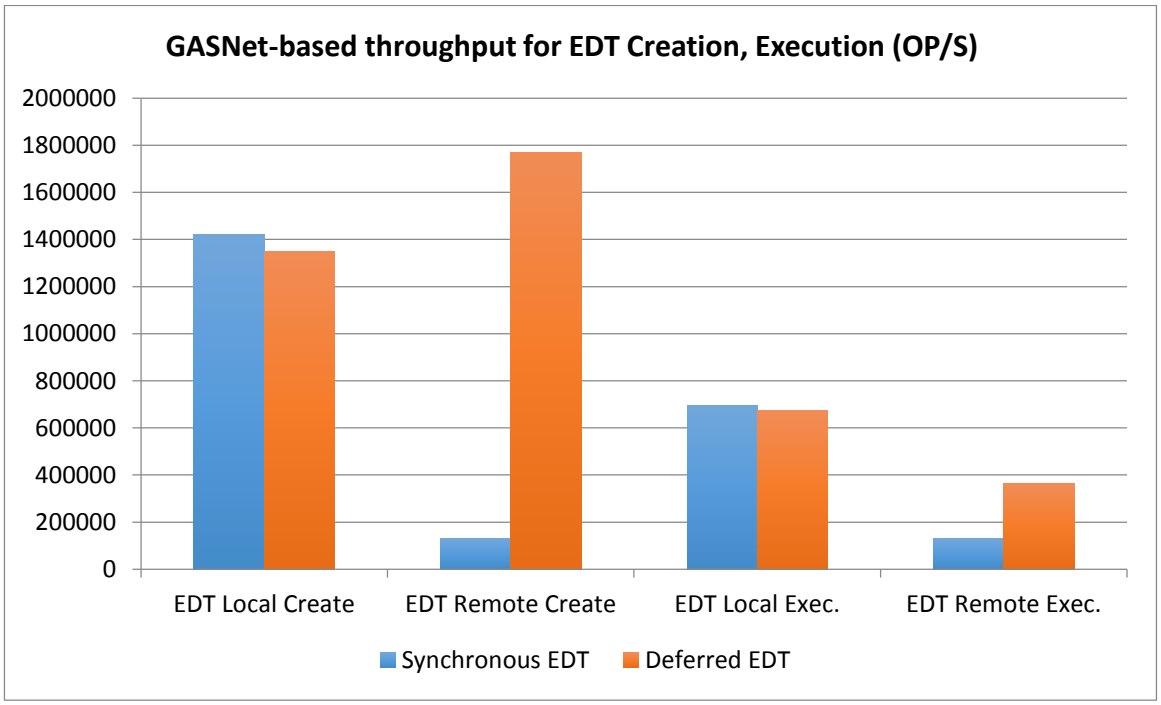


Figure 8.37: GASNet-based throughput for EDT creation and execution in operations per seconds

OCR implementation on emulated TG

To aid in the rapid development and debugging of runtime algorithms for the TG platform, we implemented an OCR target called 'TG-x86' which emulates the TG architecture. The code base between the TG target and the TG-x86 target is nearly identical with most of the differences being concentrated in the implementation of the low-

level communication, computation and memory resources, the former being very close to the bare hardware and the later relying on pthread for parallelism and the OS for memory chunks.

This version is not yet fully released as of August 31st 2015.

Metrics

Although TG-x86 emulates the TG architecture, it cannot provide the same detailed metrics as FSim (the simulator for the TG architecture) or accurate timing information.

OCR implementation on x86

We have a reference implementation on a single node x86 machine (potentially multiple workers) as this allows both us and application programmers to functionally debug their application. Please refer to other sections for performance results regarding OCR.

OCR implementation the new TG architecture

We currently have an implementation of OCR on an older version of the TG architecture. The results presented in this report make use of that version of OCR. We are currently working on finalizing the port of OCR to the latest version of the Traleika Glacier architecture.

Future work

As part of the follow-up work that will happen under the auspices of the XStack continuation, we intend to focus on the following aspects:

- Complete the port of OCR to the new TG architecture
- Improve the performance of the reference implementation
- Provide support for a higher level language. We are currently considering the LEGION programming model and associated tools.

We also intend to work with application developers to solve some of their pain points with the OCR API. The API was initially designed to be as minimalistic as possible to avoid constraining the programmer into predefined paradigms; it is, however, becoming very verbose and some higher-level features may need to be added to the APIs. Our goal is still to keep the API simple and only add APIs that not only help the programmer but also make the runtime better understand the programmer's intent and perform more efficiently.

9. Intel's Strawman Architecture and Simulator

by Joshua Fryman, Intel

Challenge Focus

With the goal of finding a software and hardware co-design solution to the Exascale initiative, Intel focused the prototype architecture on a minimalist platform that could meet the energy efficiency criteria while supporting a separation of concerns – the distinction between system needs, user algorithmic needs, and communications modules. From this hardware-oriented initial position, the co-design focus explored how to enhance the parallelism capabilities of the minimalist platform, combined with techniques to facilitate task-based execution models beyond the classical programming and execution methods.

Technical Approach

The XStack Traleika Glacier (XSTG) project started by forking the prototype architecture from the predecessor DARPA Ubiquitous High Performance Computing (UHPC) program. The UHPC effort created a bare-metal minimalist architecture and feature set that could reach the necessary energy efficiency targets. The UHPC

program's resulting architecture was not particularly friendly to developers or compilers, as it focused entirely on the energy efficiency and scale-out problems.

This baseline architecture was then modified to fix the known limitations that made programming the architecture overly complicated, creating a new baseline v2.0 architecture. This architecture then underwent 12 months of testing and implementation-related co-design evaluations, which identified several additional pain points. Correcting these additional problems in the architecture created an intermediate architecture v2.5. This architecture, internally captured in the ISA 4.0.x series of simulator tools, was extensively tested by co-design activities in applications and tools, and forms much of the basis of the work in the XStack project.

In the final 9 months of the XSTG project, all of the learnings from co-design and deep consultation with Intel design teams led to a final baseline design, the v3.0 architecture as encapsulated in the ISA 4.1.x series of simulators and tools. This v3.0 architecture is the recommended baseline for an exascale strawman prototype for evaluation, and forms the baseline of the open-source release of the simulators to the community.

Initial Prototype Architecture

The initial architecture, as borrowed from the conclusion of the UHPC program, included a heterogeneous ISA configuration, as shown in the Figure 9.1.

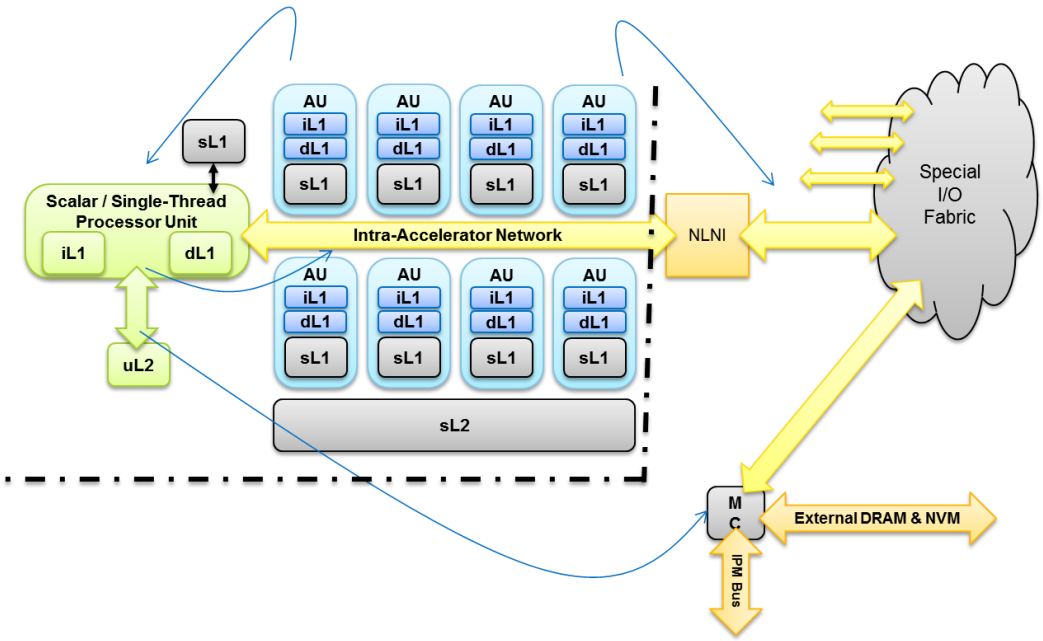


Figure 9.1

The control engine (CE), shown on the left in the above figure (Figure 9.1), runs the system software and related software technologies. In specific, the distributed task-based execution runtime executes predominantly on the CE core. Additional components of the software stack that are related to introspection of the dynamic behavior, resiliency controls, and OS/kernel functionality are all located on the CE core. User applications are targeted solely to the accelerator units, which are designed to be application-optimized execution engines (XE) shown as blue units in the above figure.

In this architecture, the CE is modeled to be a derivative of a small Intel x86 core, such as from the Intel Atom family of processors. The XE is a custom, RISC-based fixed-width encoded scalar instruction set that is optimized for energy efficiency for a target 1.0 GHz operational frequency and a low V_{dd}. Each of the CE and XE cores has

some common features, including per-core 16KB L1 instruction cache, 16KB L1 data cache, and 64KB L1 scratchpad.

This grouping of CE and multiple XEs, combined with a shared L2 scratchpad, is referred to as one architecture "block" in the design. A number of blocks are combined to form a cluster, with a shared L3 scratchpad, and the die is built out of as many clusters as will fit in the floorplan, as shown in below Figure 9.2.

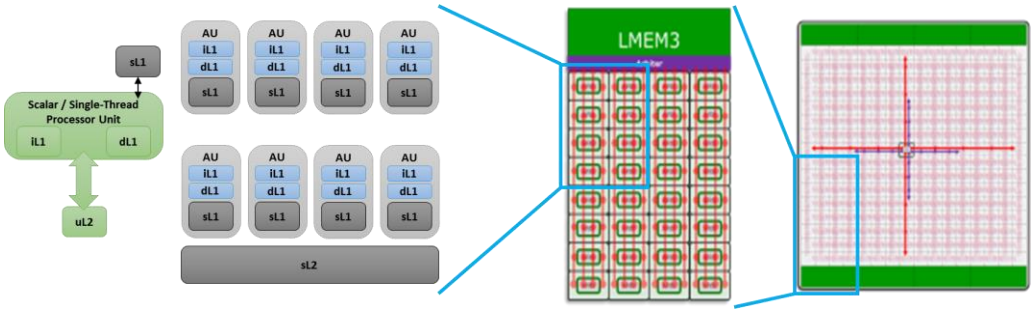


Figure 9.2

The actual organization of a node board involves additional structures and levels of memory hierarchy, as depicted in Figure 9.3 below. The processor die logic, shown in the center of the figure, encompasses the sea-of-clusters for compute and local memories. Within the package (blue region), multiple in-package memories (IPMs) are included to show energy efficient, high-bandwidth memory for local compute needs. Socket adjacent high-capacity memory is on the left side of the prototype node board, while the I/O fabric for connecting to the rest of the machine is on the right edge.

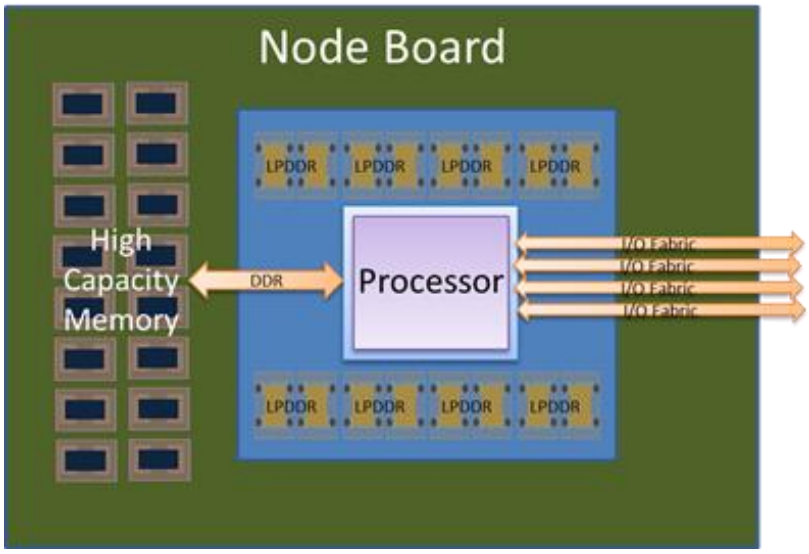


Figure 9.3

To provide further opportunities for the underlying system to optimize execution characteristics, this prototype system uses a relaxed memory consistency model, with a robust set of "fence" instructions added to the ISA for programmers to reason about the visibility of their memory accesses. The combination of scratchpads and in-package / out-of-socket memories construct a significantly more complicated memory hierarchy, as depicted below for a prototype exascale machine.

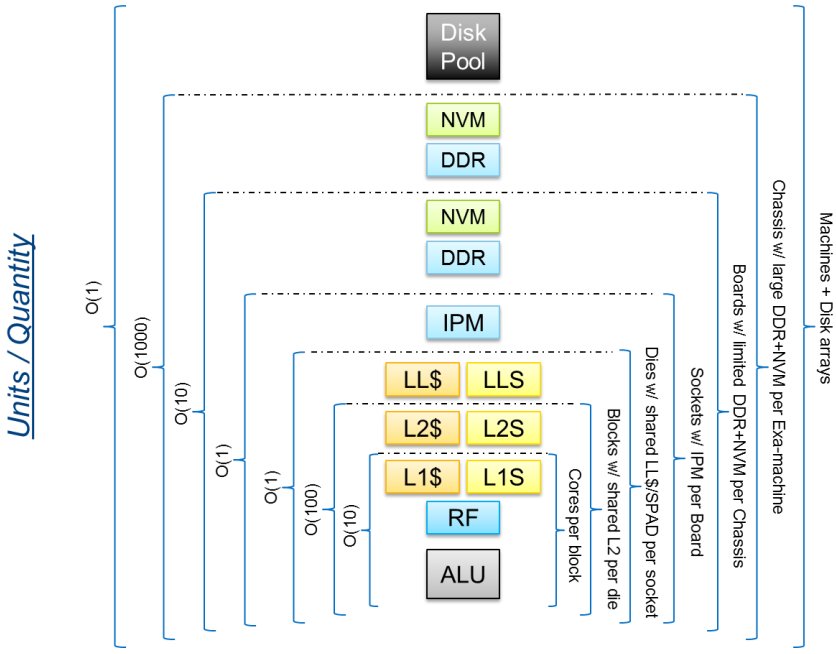


Figure 9.4

Evolution of Prototype Architecture

From the baseline UHPC model, several problems were identified for programmers, compilers, and hardware architects on the evolutionary march of the architecture from v1.0 to v3.0. The major changes are described below, which resulted in a co-designed architecture that appears to remain highly energy-efficient for the evaluated DOE proxy workloads, while the revised architecture is also straightforward to target using known compiler methods and simple ports of existing open-source compilers.

The memory map complexity was originally handled by opcode-based separation of operations between memory regions. Therefore, load, store, and atomic operations required a different opcode based on distance to the target memory (local, remote). This resulted in significant code bloat around pointers, many of which could not be known *a priori* to their use and therefore required dynamic codepaths to resolve. With assistance from the hardware engineers, an address-based scheme was constructed that allows for a single opcode and the hardware to dynamically steer memory operations to their destination. This reduced code sizes by > 20%, and facilitated better code generation from compilers.

A second problem arose due to the increasing scale of evaluation. In the original architecture, inter-processor interrupts (IPIs) allowed a very small number of events to transpire before they became lossy due to dropped temporally older events. The architecture evolution created a hardware-based queue system with software-allocated buffers that allows for extremely fast IPIs without loss. Once designed, this queue mechanism became highly flexible, and allowed for significant expanded use as a dedicated communication channel for the runtime as well as applications.

Each core had a micro-DMA unit for offloading data movement operations, including message buffers. Through the co-design process, this DMA unit was shown to have high value when applied to the scratchpads and in-package memories. These features lacked an ability to do “merge” operations of data values when copied, however, which is a common operation identified in many classes of DOE workloads. With co-design input from DOE application experts, the DMA unit developed additional capability for datum-level atomic merging of new values to existing values by a programmer-defined operation (add, subtract, multiply). This eliminated both an additional message buffer as well as the requirement for a user-created “merge loop” on the destination. As an unexpected benefit,

this also improved the dataset surface-to-volume ratio that the local memories could contain, further improving the efficiency of algorithm implementations on the strawman architecture.

A final significant improvement was to add the ability to off-load from a core an entire communications exchange sequence in an operation called *chaining*. The programmer specifies the start of an instruction chain sequence, and then issues a series of queue, DMA, and atomic operations to be executed in FIFO order. When the programmer marks the end of the chain, the entire sequence of operations is executed in an offload agent local to that core. While the core resumes processing of routine operations, the offload unit manages the entire sequence, therefore decoupling the overhead of communications management from the user algorithmic work. This allows a scalar core to behave, for communication purposes, as a superscalar core in an energy-efficient manner.

Simulation Technology

In order to simulate the radically different architecture, and to provide a high-speed simulation environment that could scale to thousands of cores, the UHPC program created a new simulator called FSim – short for Functional Simulator. Given the state of simulator technology for massively parallel simulation when UHPC started in 2010, the only viable option was to create a new framework. As the prototype architecture evolved during the XSTG program, the FSim infrastructure evolved to match, adding new features and enhancements on top of modeling the underlying platform.

The basic design of the FSim tool is based on the "block" concept in the architecture. Each block, being one CE and multiple XEs, is launched as a single Linux process in a cluster. Each block becomes its own process, and these processes are connected to a simulated network via their next-level network interface (NLNI) module. In real hardware, the NLNI module is the network switch connection to the on-die fabric for the block. In Fsim, the NLNI module is a loose wrapper to a Linux socket, whether accessed as Ethernet or Infiniband Verbs. The modeled larger memory structures, such as L3 scratchpads, IPM regions, and external DRAM / NVM are also launched as Linux processes, with their own NLNI module. A top-level server was launched which connected all of these processes together within the cluster, routing traffic from block to block via the sockets contained in the NLNI modules. This is depicted in Figure 9.5 below.

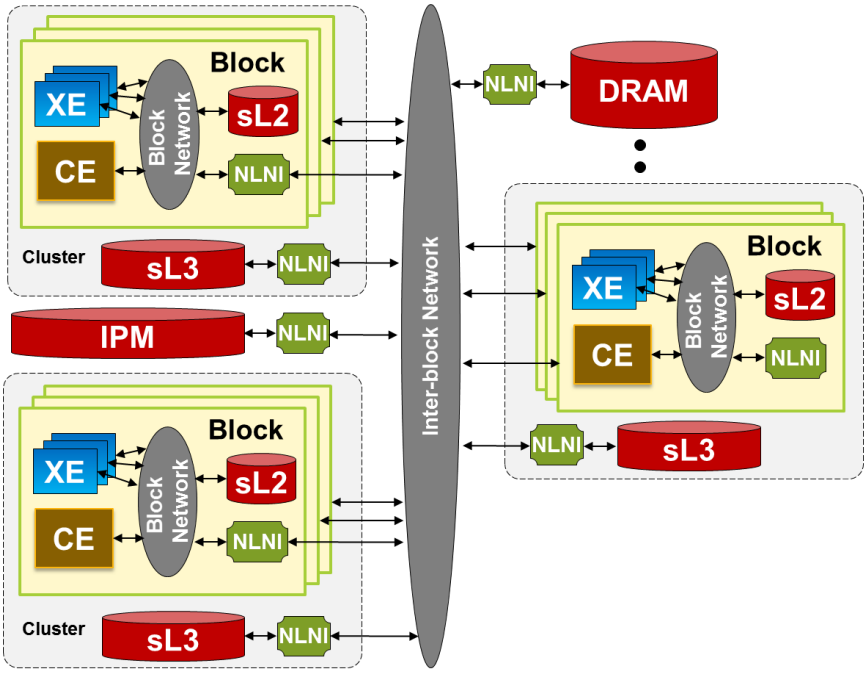


Figure 9.5

The emulation of the CE x86 cores was achieved by modifying the QEMU[7] platform emulator, integrating it into the FSim simulation environment. The QEMU changes enabled the architecture model of QEMU to be run as a library component inside of FSim, instead of as a full stand-alone emulation application for virtual machines. Additional changes were made to bypass the platform-level emulation components (I/O controllers, PCI bus, etc.) and instead strip QEMU down to just an x86 core model with a similarly reduced BIOS image. The XE component simulator was a custom execute-at-execute instruction-level simulator that matched the ISA evolution of the XSTG project.

Overall, a Lamport clocking[8] scheme was implemented to provide a simple proxy for timing and ordering between cores. Any more rigorous clocking or timing model would adversely impact the performance and resulting usefulness of the Fsim environment, so the Lamport clock runs could have their traces post-processed to asses more complicated models. In essence, the output from the FSim tool could be post-processed as a stimulus to more advanced network and core models, to study artifacts of interest in more depth.

For energy estimations, FSim was modified to use a basic dynamic capacitance model. The basis for this model is a simplified power model, where $P = afCV^2$. This focuses on dynamic power, and not static power – which is highly variable and controlled through processes different from dynamic power. Rewriting this equation slightly, the dynamic power can be expressed as $P = (aC)fV^2$. Here, we group the circuit activity factor a and the total switched capacitance C as a single unit, which we refer to as the dynamic capacitive load, C_{dyn}. Therefore, the power equation becomes $P = C_{dyn}fV^2$. By establishing the C_{dyn} term, in pF, for operations such as instruction fetch, various arithmetic operations, memory accesses, etc., the energy estimation becomes a simple summation of each C_{dyn} event. This allows for easy scaling of energy across different fabrication nodes (22nm, 14nm, etc.) and different voltage or frequency targets without making assumptions about the underlying architecture.

Results and Analysis

At the time of the 2015-Aug-31 release snapshot, FSim has a general performance characteristic as shown in Figure 9.6 below on a per-core average simulation speed. Two extremes, all local computation and all DRAM-dependent atomic memory ops, show the extreme edges of the simulator behavior on Intel Xeon E5-26900, running at a nominal 2.90GHz according to /proc/cpuinfo. In the all-DRAM remote atomic operation behavior, no memory re-ordering or out-of-order support is possible due to the dependencies: only one instruction is possible at a time to the full DRAM latency. The Cholesky kernel is shown as a representative comparison for a workload that involves both compute and communication. Comparison points against FPGA emulation platform speeds and RTL simulation speeds on workstations are shown for comparison.

[7] QEMU as a generic, open-source emulator can be found at: http://wiki.qemu.org/Main_Page
[8] Lamport clocks and timestamps are a common technique: https://en.wikipedia.org/wiki/Lamport_timestamps

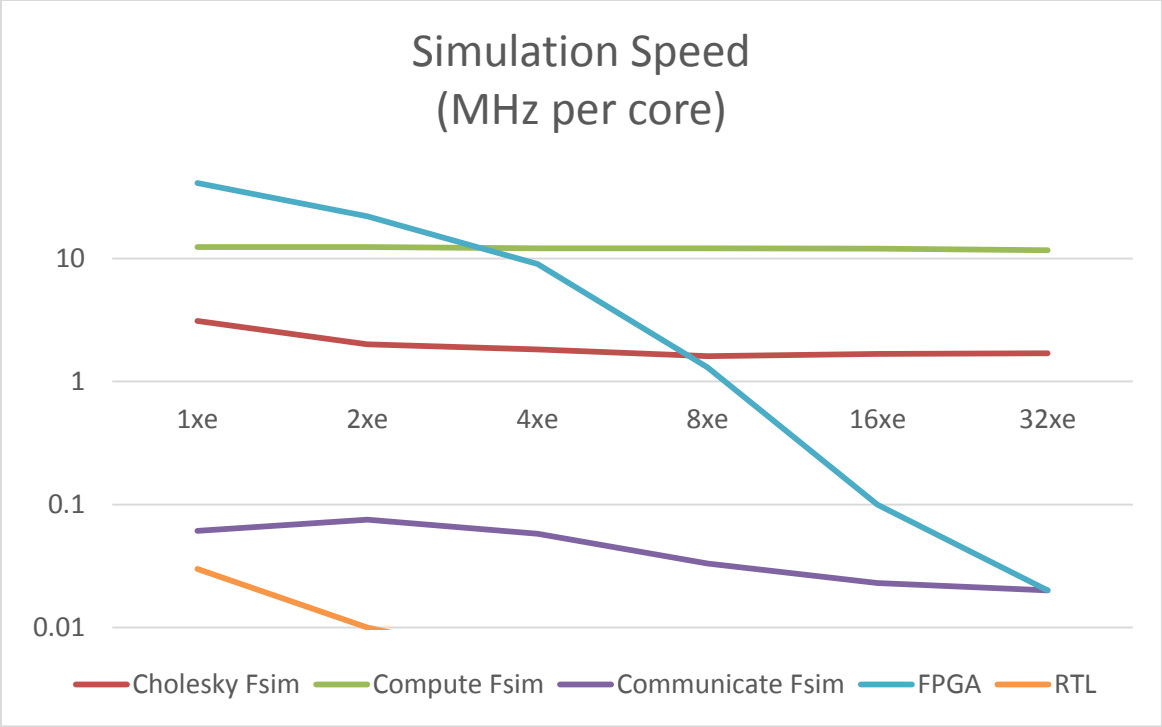


Figure 9.6

The primary observation of this data – that RTL and FPGA models do not scale – explains why a simulation infrastructure that was designed for parallelism and performance were necessary. RTL simulators do not scale because they are limited to one cache-coherent machine, which is effectively a single high-end workstation. FPGA platforms do not scale when the design exceeds the ability to be "fit" (placed and routed) inside one FPGA. As soon as multiple FPGAs are required, most designs for processing systems become I/O pin limited – particularly when moving 1,024-bit buses that are bi-directional and have multiple paths. The synthesized speed of the logic in the FPGA becomes limited by the required mux to strobe many logical pins over one physical pin, significantly reducing the observed clock rate.

In contrast, the performance of FSim is shown under three scenarios: an all-compute workload (~11 MHz); an all-communication workload, where every instruction invokes a distant memory access bypassing caches (~0.08 MHz); and a typical run of the naïve OCR-native version of Cholesky factorization, which is a blend of compute and communicate (~3 MHz).

When the physical resources to run these different types of environments is factored in, the value of FSim becomes even more apparent. Ignoring whether it is actually possible to attain certain speeds, the cost in dollars for one of the simulation tools to run at 1 MHz based on their real behaviors is shown in the graph below.

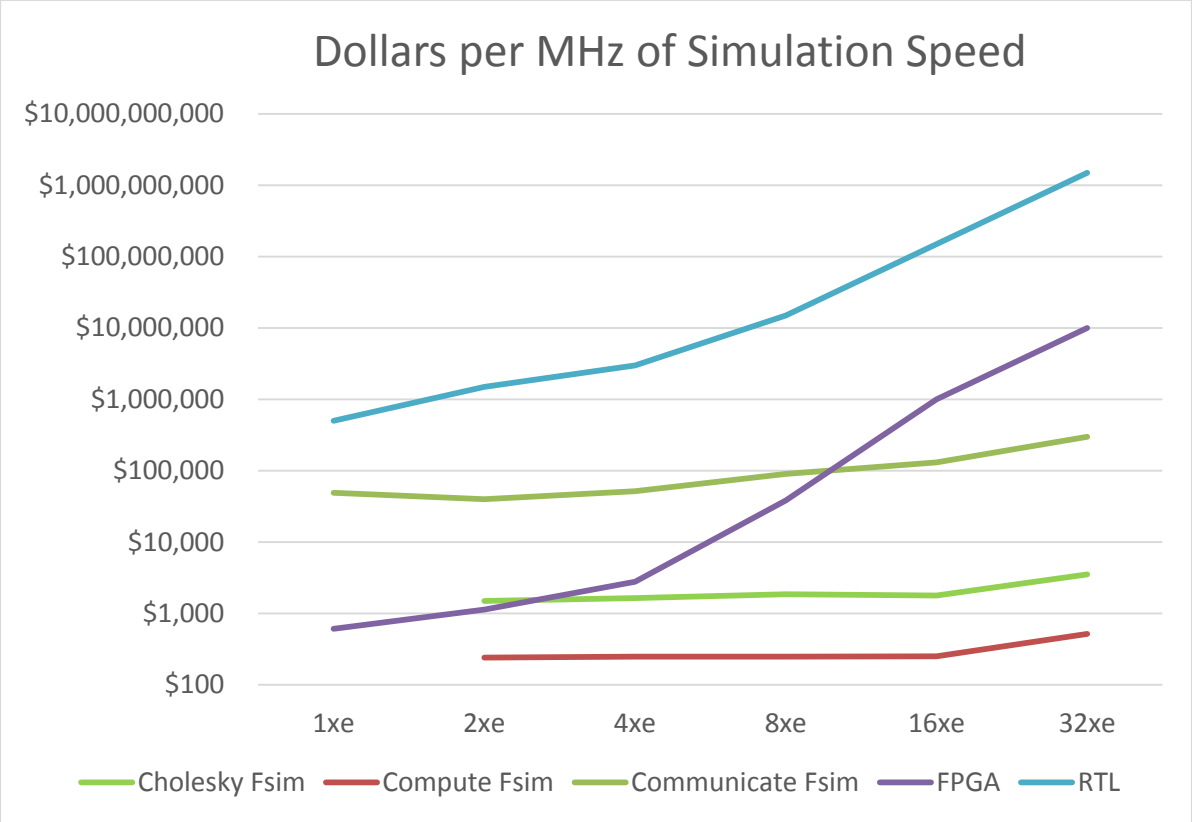


Figure 9.7

As shown in the graph (Figure 9.7), when the cost of FPGAs is factored into the reduced performance due to I/O pin limitations, they quickly become unrealistic for larger simulation sizes. FPGAs have value as long as the scope of the simulated machine remains small, and fits in 1-2 FPGA chips. The RTL simulation capability is limited by what can be done in a single cache coherent machine, which makes it impossible to afford. FSim scales with the simulated machine size by adding relatively modest dual-socket compute nodes to a cluster, where each compute node is inexpensive with low memory and local storage requirements. The major caveat, however, is that FSim will provide a simulation performance in inverse proportion to the degree of communication in the workload: more communication in the workload will slow the simulator accordingly.

Recommendations and Future Directions

1. The current design of FSim's central server is using one Linux process as the focal point that every spawned block (Linux process) must connect to for the simulation lifetime. With each spawned block requiring three separate ports, the central server as one standalone process is unlikely to scale beyond 1-2 sockets, which will require up to 2,000 ports per simulated socket. The physical limitations of network stacks on Linux machines and the overhead of so many concurrent connections is problematic.
   Recommendation: Breaking the server component of Fsim into a hierarchical design is advised, where the "leaf node" server manages one socket, a "middle" server manages one small chassis in a rack, a "high" server manages on rack, and the "top level" server connects all the racks together. Not only will this allow Fsim to scale to significantly larger levels, it also naturally models the hierarchical design and interconnect properties of real machines.
2. The use of C_{dyn} models corresponds well to standard industry practices to model dynamic power and make overall power projections, and providing these improved energy models and methods is of high value. However, these C_{dyn} models are flawed when used to model some structures, such as DRAM and NVM

technologies. These are better represented as a blend of background power and active power, instead of capacitive load. Additional problems arise in that the accumulation of the C_{dyn} term assumes a constant voltage when the final conversion to power or total energy is made. This does not reflect the reality that computational logic (pipelines, registers) may be at a different voltage from other structures (cache, scratchpads, interconnect, I/O pads, DRAM, NVM).
Recommendation: Modifying the energy accumulation behavior of FSim to account for pairs of (C_{dyn}, V_{dd}) rather than just the dynamic capacitance is a required first step to refine this model. In order to provide better support for technologies such as DRAM and NVM, FSim needs to perform a local-logging of traffic and the associated energy costs, rather than accumulating the round-trip costs and presenting that total to the originating requestor. This will enable technology-appropriate models to be used at each stage of evaluation. The drawback is that it will require post-processing to amalgamate charges to originators, when that level of understand is required.

3. Similar to the limitations of the C_{dyn} models, the use of Lamport clocking and simplistic bandwidth models hampers the ability to directly use the timing estimation information generated by FSim. While Lamport clocks impose a total order based on timestamp exchanges during communications, it is susceptible to high variance between mostly-computing and mostly-communicating behaviors. This leads in turn to significant clock warping on the communicating cores, combined with a lack of slowdown on the computing cores. The naïve network bandwidth limiting model also allows for unrealistic execution behaviors.
   Recommendation: For functional simulation purposes – the early bootstrapping of an entirely new software ecosystem – changing these behaviors around clocking and bandwidth will result in unacceptable performance loss in favor of more accurate models. At the same time, direct measurements of performance or estimations for future performance and not trustworthy without additional tools. Research and engineering effort should be spent to find a compromise between overall accuracy and simulation speed, particularly when FSim pipeline models and other components can easily be integrated with other tools such as detailed pipeline or network models.
4. Finally, there is a strong need for an ISA-aware integrated debugger – otherwise debugging an application requires stepping through the simulator itself at the same time. By creating an ISA-derivative port of a common debugger, such as GNU gdb, the basic ISA operations and software ABIs can be understood. Tools such as gdb include remote-control capability via a "stubs" remote-debug interface.
   Recommendation: To make the simulator friendly to end users, a gdb "stubs" port should be made and glued onto the FSim simulation infrastructure. Concurrently, adapting the "parallel gdb[9]" tools created by others and released as open source, such as "pgdb" by LANL, could be applied to facilitate better controls for the application developer in the presence of the heterogeneous ISA of the v3.0 architecture.

Products and Bibliography

Publications:

Nicholas P. Carter, Aditya Agrawal, Shekhar Borkar, Romain Cledat, Howard David, Dave Dunning, Joshua B. Fryman, Ivan Ganev, Roger A. Golliver, Rob C. Knauerhase, Richard Lethin, Benoît Meister, Asit K. Mishra, Wilfred R. Pinfold, Justin Teller, Josep Torrellas, Nicolas Vasilache, Ganesh Venkatesh, Jianping Xu. "Runnemede: An architecture for Ubiquitous High-Performance Computing." In the IEEE Symposium on High Performance Computer Architecture (HPCA), 2013.

Code:

FSim infrastructure and tools: https://xstack.exascale-tech.com/git/public/tg.git

[9] "pgdb" by LANL's Nikoli Dryden, available at: https://github.com/ndryden/PGDB

10. University of Illinois, Urbana Campus - Architecture and Sensitivity Analysis

By Wooil Kim, Josep Torrellas

Challenge Focus

The goal of this section is to evaluate the TG architecture running OCR-enabled applications. In particular, we want to evaluate: (1) the performance and energy scalability of the TG architecture, and (2) features of the OCR runtime system that exploit TG's large on-chip parallelism and hierarchical on-chip memory structures (blocks and units). Some interesting OCR runtime features are: (1) scheduling Event Driven Tasks (EDTs) for parallelism, and (2) allocating Data Blocks (DBs) for data locality. The former involves scheduling the execution of each EDT as soon as it is possible, given the structure of dependences between EDTs. Allocating DBs for data locality involves allocating each DB in a memory module close to the EDT that will use it, and de-allocating the DB when it is unneeded, to make space for other DBs.

Technical Approach

Since the TG architecture only exists as a simulation model, we evaluate it by running codes on the FSIM simulator of the TG architecture. As a result, given that the simulations take considerable time, we use kernels in our evaluation. Specifically, we use *Cholesky, Smith-Waterman, FFT,* and *SAR*. These kernels can be analyzed in great detail, and exhibit different behaviors. We perform runs with up to 64 XEs.

While our goal is to evaluate the architecture with the latest OCR version, we find that version 4.1.0 is not yet stable enough for our experiments. Instead, we perform the evaluation for version 4.0.8. Moreover, although the envisioned scenario involves a completely automated EDT scheduling for parallelism and DB allocation for locality, this is not the current state of the system. Instead, we have to perform some of the intelligent EDT scheduling and DB allocation with programmer-inserted hints.

1. Kernels Evaluated

We choose four kernel programs, namely *Cholesky*, *Smith-Waterman*, *FFT*, and *SAR*. They show different degrees of parallelism and various kinds of memory access patterns. While these benchmarks have been described before, here we focus on the aspects relevant to this chapter.

Cholesky is a decomposition method of a positive, definite matrix into a product of a lower triangular matrix and its conjugate transpose. The algorithm is iterative, and values produced in the previous iteration are used in the current iteration. The algorithm has $O(N^3)$ complexity because it iterates N (the matrix size in one dimension) times for matrix elements (N x N). The sequential algorithm is written as follows:

```
for (k = 0; k < N; k++) {
  A[k][k] = sqrt(A[k][k]);
  for (j = k+1; j < N; j++) {
    A[j][k] = A[j][k] / A[k][k];
    for (i = k+1; i < N; i++) {
      A[i][j] = A[i][j] - A[i][k] * A[j][k];
    }
```

```
  }
}
```

Smith-Waterman is a DNA sequence-alignment algorithm. The algorithm traverses an input data matrix from the top-left element to the bottom-right element with diagonal steps due to data dependence on left element, top element, and left-top element. In the parallel version, a tiled data layout is used to avoid per-element synchronization.

FFT implements the Fast Fourier Transform algorithm by Cooley-Tukey. In the parallel version, the algorithm works in a divide-and-conquer style until each EDT gets the minimum size of data.

SAR is a synthetic aperture radar algorithm, which is used to create images of an object, such as landscapes. An antenna, typically mounted on a moving platform such as an aircraft, forms a synthetic aperture and is used to capture multiple images. The application combines images from multiple antenna locations to create a finer resolution image.

2. EDT Structure of the Kernels

The four kernel programs have different degrees of parallelism, different memory access patterns, and involve different levels of effort in implementation. These variances expose what form of parallel programs run well on the TG architecture, and how much effort of parallelization or what aspect of effort is required for better performance. Table 10.1 shows a comparison of the four kernels in terms of parallelism and memory access behavior. Note that *FFT* has only a single DB allocated in *mainEdt*, which is a result of a direct translation from the sequential version of the program. Other programs have tiled data and associated EDTs.

Kernel	Parallelism Style	Degree of parallelism	Number of DataBlocks
Cholesky	Variable over time (decreasing)	max. $\frac{t(t-1)}{2}$	$\frac{t(t+1)}{2}$
Smith-Waterman	Variable over time (increasing in first half, decreasing in second half)	t	t^2
FFT	Variable over time (divide-and-conquer and merge)	$\frac{2^N}{minimum\ processing\ size}$	1
SAR	Fork-join style	t^2	t^2

Table 10.1. Comparison of kernel programs (N is the problem size in one dimension, T is the tile size in one dimension, and t is $\frac{N}{T}$, which is the number of tiles in one dimension.)

Figures 10.1-10.4 show the dependence structure between EDTs. Each program shows a distinct pattern, leading to different degrees of parallelism. Figure 10.1 shows the EDT dependence structure of *Cholesky*. *Cholesky* divides an entire input matrix into (t x t) tiles, where t (the number of tiles in one dimension) is N (the matrix size in one dimension) divided by T (the tile size in one dimension). An EDT for the current tile (blue) has dependences on EDTs for tiles being read (green).

Each EDT operates on a tile. The top row shows the EDTs executed in iteration i; the second row shows the EDTs executed in iteration i+1. In each iteration, there are three types of EDTs. The *Sequential* EDT is the single tile in blue in the leftmost plot of each iteration. It needs to execute first. The *Trisolve* Task EDTs are the blue tiles in the central plot of each iteration. They can only execute after the Sequential Task EDT completes. Finally, the *Update Non-*

Diagonal and *Update Diagonal* Task EDTs are the blue tiles in the rightmost plot of each iteration. They can only execute after the Trisolve Task EDTs complete.

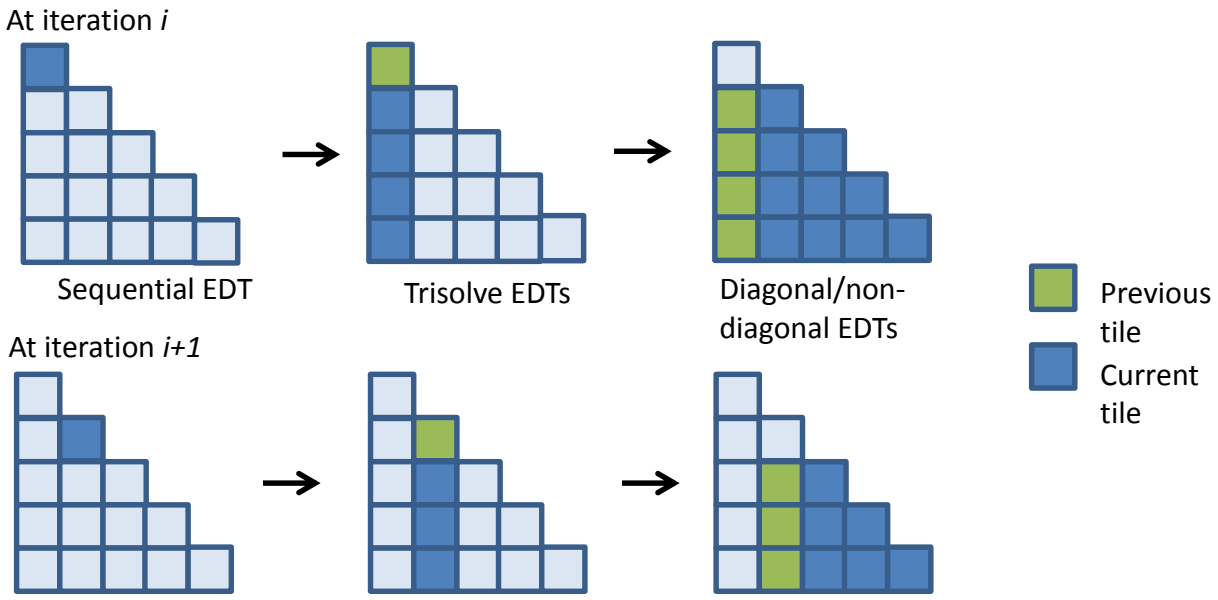


Figure 10.1. EDT dependences in Cholesky.

Figure 10.2 shows the EDT dependence of *Smith-waterman*. The EDT for a tile has dependences on the EDTs for the tiles in NW, W, N directions. In Figure 10.2(a), this is shown as the blue tile depending on the three green tiles. In turn, the tile processed in the current EDT creates data that is used in the EDTs for the tiles in the S, SE, E directions. This creates a wave-front style parallelism that leads to a maximum degree of concurrency of N when NxN tiles are available. A typical parallelization with a barrier leads to the parallel execution profile shown in Figures 10.2(b) and 10.2(c). EDTs are scheduled on available XEs, but the degree of parallelism is limited. Suppose that we have 4 XEs. In this case, we can only keep the 4 XEs busy in one of the inter-barrier steps.

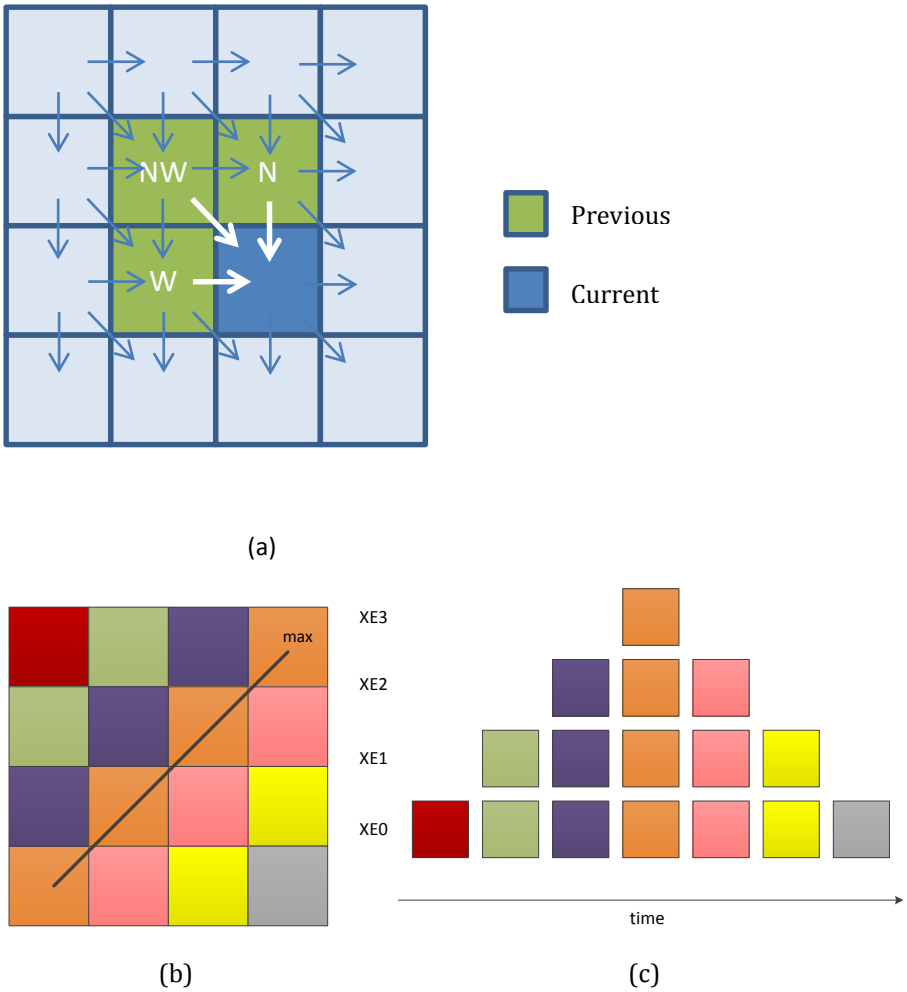


Figure 10.2. EDT dependences in Smith-Waterman.

Figure 10.3 depicts *FFT*'s divide-and-conquer style spawning and merging steps. While parallelism is exposed well, *FFT* does not use tiling. Thus, *FFT* is an example of insufficient effort to program for the TG architecture.

SAR has fork-join style parallelism as shown in Figure 10.4. *SAR* executes five parallel sections between sequential sections. Since the program uses a fixed tiling factor, the degree of parallelism is determined by the source image size.

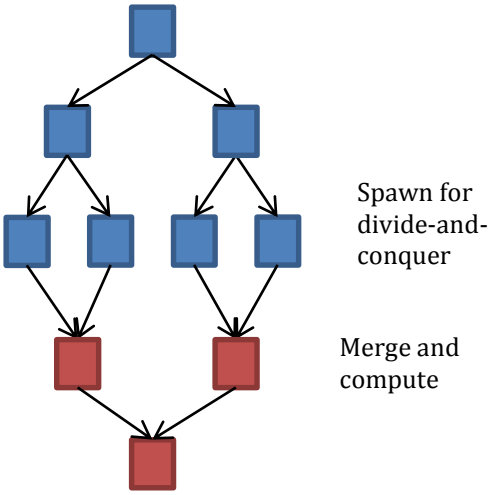


Figure 10.3. EDT dependences in FFT.

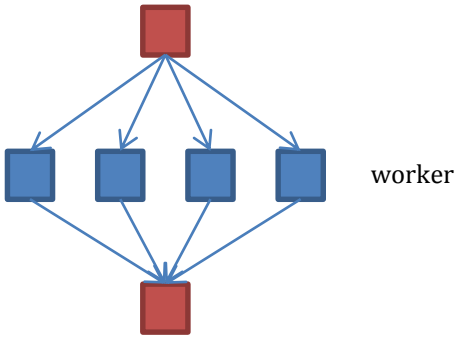


Figure 10.4. EDT dependences in SAR.

Results and Analysis

1. Key Findings

We start by summarizing the key findings of our work. First, in an architecture with as much on-chip parallelism as TG, it is important to schedule the EDTs for parallelism, so that individual EDTs execute as soon as possible, subject to the data dependences of the program. Typically, at any given time, there are many idle XEs that can be utilized.

Second, given the clustered-based, hierarchical organization of TG's on-chip memory modules, it is key to ensure that the DBs are close to the EDTs that use them. This means that we need DB allocation and migration for locality. Moreover, when the DB is not needed anymore, de-allocation to free-up space for other DBs is beneficial.

Third, intelligent DB allocation is a requirement not just to attain good performance; it is a requirement for energy savings. It makes a big difference in energy consumption whether the DB and its EDT are located close by.

Fourth, it is hard to perform EDT scheduling for parallelism and DB management for locality manually. The programmer has to understand the code well. It is still an open problem whether these operations can be effectively performed automatically.

Finally, the system evaluated (OCR on FSIM) still has certain overheads and inefficiencies that need to be improved in future versions. In particular, launching and migrating EDTs is slow. In addition, some of the statistics do not provide correct insights.

2. Performance Scaling

Figure 10.5 shows the execution time for different numbers of blocks and XEs per block, normalized to the execution time of the configuration with one block and one XE per block (b1x1). We use the b1x1 configuration as a baseline, which executes the EDTs sequentially.

Cholesky shows good performance scaling with more XEs up to b2x8 (two blocks with eight XEs each). Adding more XEs does not improve the performance for this problem size. Adding more blocks provides more computation resources, but it also increases the latency of memory accesses if inter-block accesses are required. All runs use a tiled version of *Cholesky* with data size 500 and tile size 50. If we used a non-tiled version for the baseline, the baseline performance would be worse because the single data block would be allocated in the main memory, rather than on the block memories.

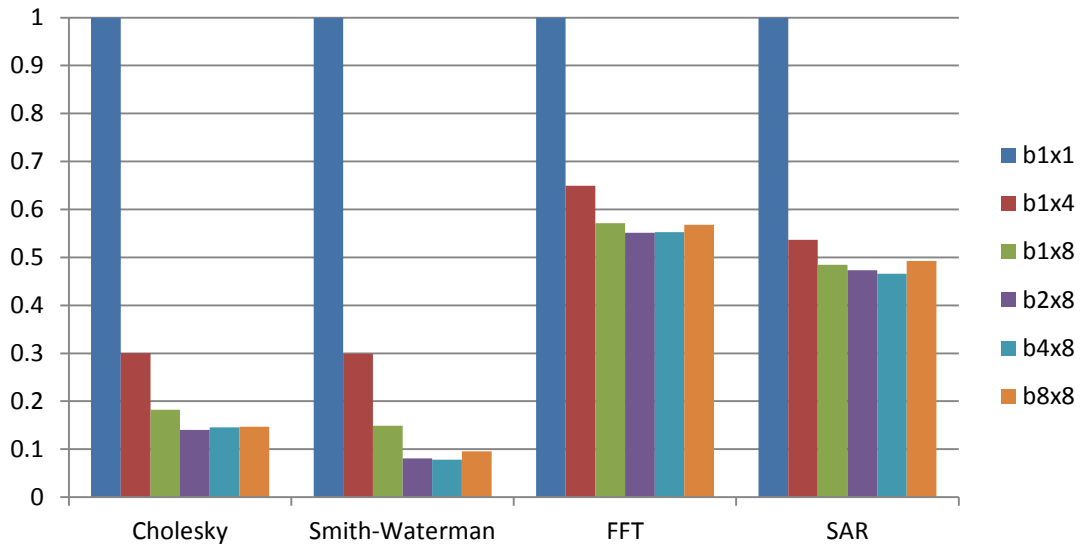


Figure 10.5. Normalized execution time of kernel programs with different numbers of blocks and XEs.

Smith-Waterman is run with data size 3200 and tile size 100. It also shows performance improvement up to b2x8. *FFT* is run with problem size 18. It shows performance improvement until b1x8. More than 8 XEs do not help the performance much. *SAR* runs with the 'tiny' problem size, and executes a 64x64 image size with 32x32 tile size. It shows meaningful speedups until b1x8.

3. Energy Consumption Breakdown

Figure 10.6 shows the breakdown of the energy consumption with varying numbers of blocks and XEs per block. The bars are normalized to the one XE configuration (b1x1). We use the same problem sizes and configurations as in the previous section. Our basic expectation is that the energy consumption is approximately constant as we increase the number of XEs because the work done is similar (except for some overhead of parallelization and OCR). However, it is also possible that, with more blocks, the energy consumption decreases. This is because we have more block memory (BSM) to allocate DBs, which is cheaper to access than DRAM memory.

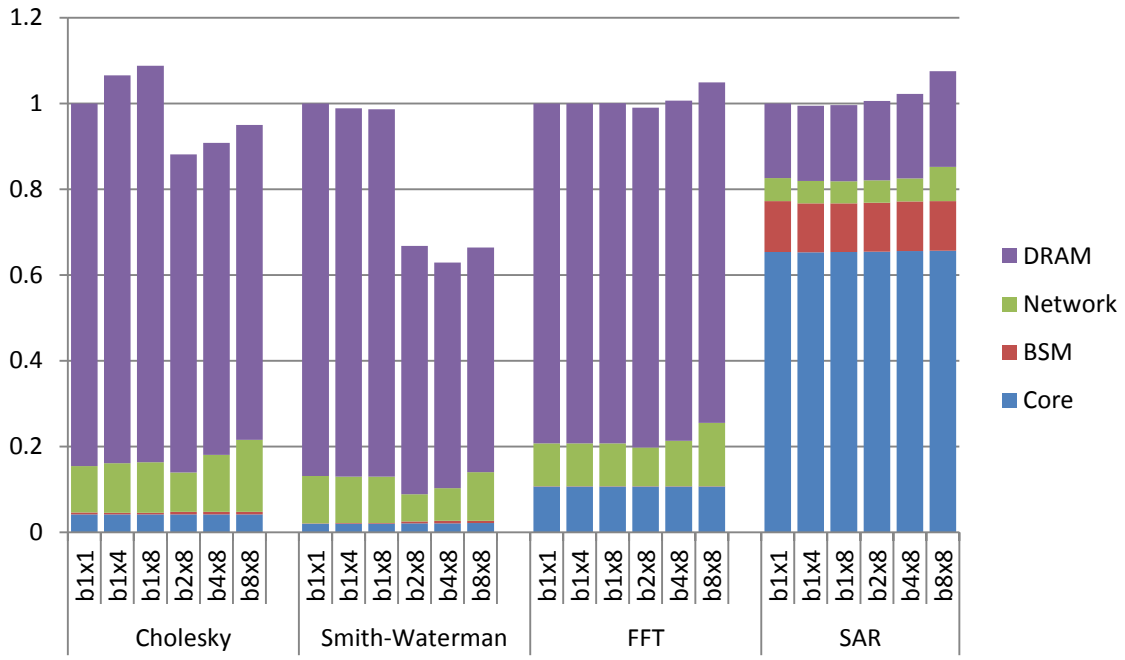


Figure 10.6. Normalized energy consumption breakdown with different numbers of blocks and XEs.

From the figure, we see that DRAM energy consumption dominates in *Cholesky*, *Smith-Waterman* and *FFT*. This is because accesses to DRAM are very costly in energy. With a single block, the data size used in the programs is larger than the 1.25MB block memory (BSM) size. Thus, even with tiling, not all DBs are located in on-chip memory; about half of them are allocated in DRAM. As we move to the b2x8 configuration, both *Cholesky* and *Smith-Waterman* show energy reduction. This is because of the increased BSM size. More DBs are allocated in the block memories, and the number of DRAM accesses decreases. Unfortunately, *FFT* has a single DB, and it is large enough to be allocated in DRAM. Thus, most accesses go to DRAM. *FFT* shows the necessity of DB tiling. *SAR* is a more computation-intensive kernel, and its small problem size is absorbed by the block memory.

In general, with more XEs, the DRAM energy and network energy has a tendency to increase. This is because the scratchpad memory is also used by both the OCR, and the memory usage of OCR increases with more cores.

6. Effect of the Granularity of Parallelism

Varying the tile size results in two effects: changing the number of concurrent EDTs and the location of the memory allocation. Smaller tile sizes enable more concurrent EDTs; however, more EDTs/DBs can incur more storage overhead for OCR internal data structures. Larger tile sizes have trade-offs between parallelism and EDT overhead.

To see the effect of tiling, we use *Cholesky* with a matrix size of 500 and vary the number of tiles, by changing the tile size from 25 to 250. The results are shown in Figure 10.7, normalized to b1x1 and tile size of 50. With bigger tiles, the parallelism is lower and, therefore, execution time is higher. With smaller tiles, we have more tiles and more parallelism. This reduces the execution time. In the case of tile size 25, we see speed-ups until the b4x8 configuration. However, for tile size 50, the execution time does not decrease after b2x8 because of EDT overhead.

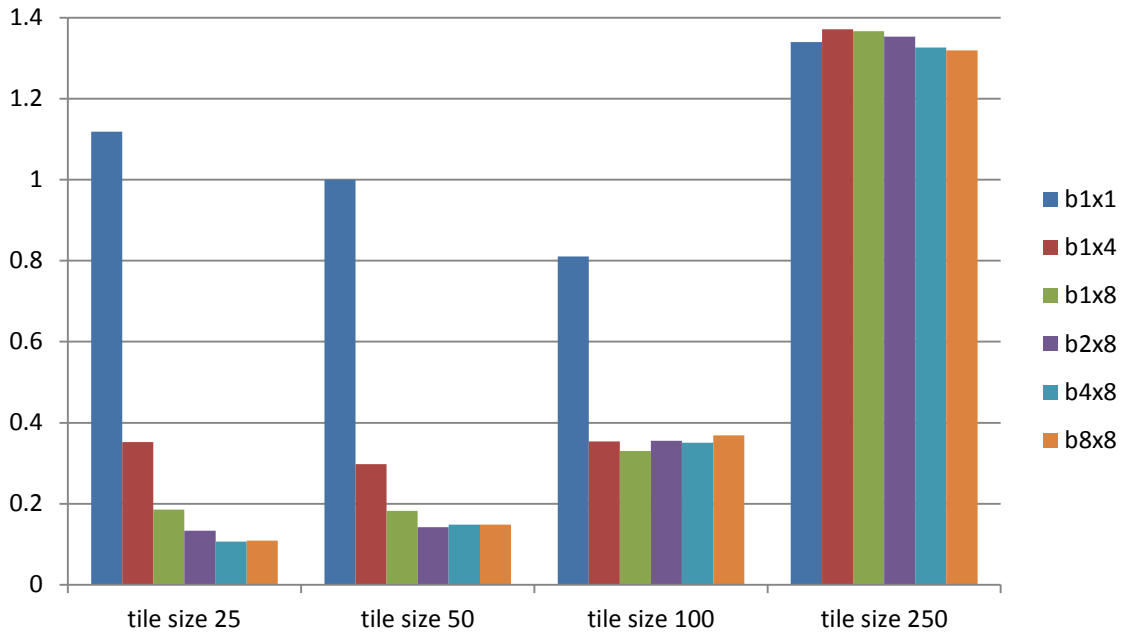


Figure 10.7. Normalized execution time with different tile sizes.

Going from tile size 50 to tile size 100, we reduce the total number of computing EDTs from 220 to 35. With tile size 100, the performance of b1x1 is better than the case of tile size 50 because of less overhead of EDTs. However, tile size 100 has lower scalability. Increasing the number of XEs hits scalability limits earlier than with tile size 25 or 50. In the case of tile size 250, there is no parallelism because all EDTs processing tiles have a dependence chain. Therefore, increasing the number of XEs is useless.

Figure 10.8 shows the energy consumption with varying tile size. Tile size 100 shows the lowest energy due to less complexity of EDT relations. For smaller tile sizes, the memory allocation of more events and OCR data structures reduces the size of the block memory available to the application.

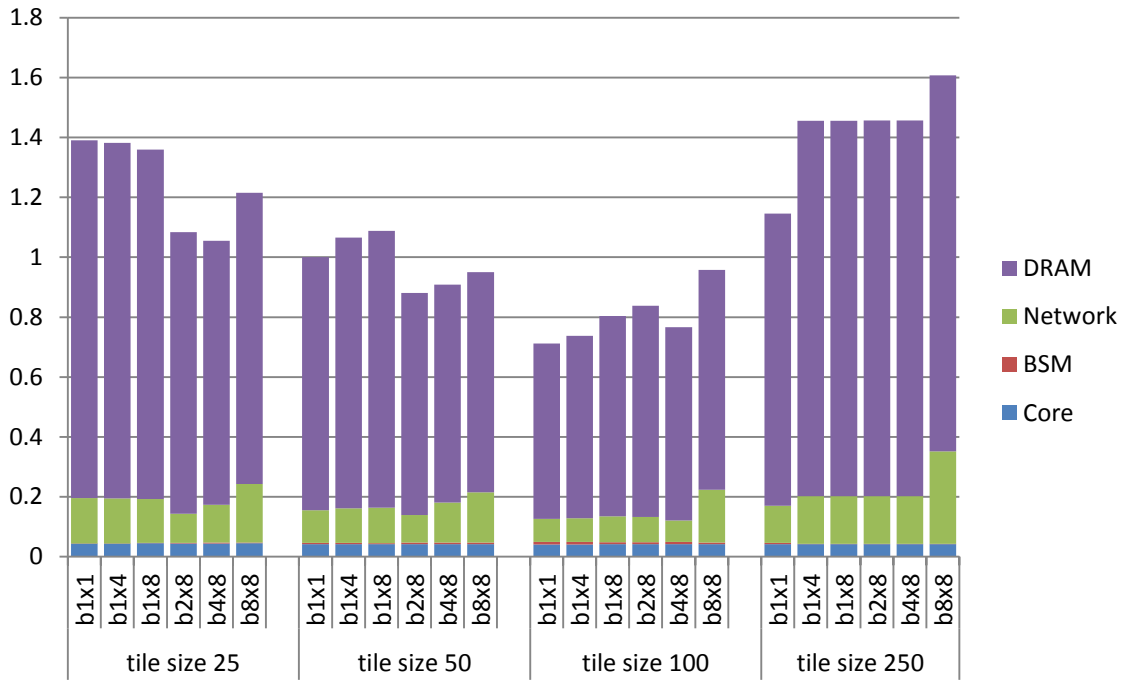


Figure 10.8. Normalized energy consumption breakdown with different tile sizes.

7. Scheduling EDTs for Parallelism

To see the impact of scheduling EDTs for parallelism, we consider *Cholesky* with the b1x4 configuration. In Figure 10.1, we showed the dependences between EDTs in *Cholesky*. With traditional barrier-based parallelization, iterations must be separated by barriers. Thus, all the EDTs in iteration *i* must be completed before any EDTs in iteration *i+1* execute.

With OCR, EDTs from different iterations can be executed concurrently if there is no dependence remaining for the EDT. OCR tracks dependences between EDTs at finer-granularity. In *Cholesky*, the EDT of a tile in iteration i+1 does not need to wait for the EDTs *of all the tiles* in iteration i to complete. Instead, a tile in iteration i+1 can be processed as soon as all those it depends on (in iterations i and i+1) complete. This is what OCR enables in a manner that is transparent to the programmer. With OCR, we can overlap EDTs from different iterations, as long as the dependences between individual tiles are respected.

Figure 10.9 shows a timeline of the EDT scheduling observed during execution. The figure shows which EDT is executing on each XE. The color code is as follows: the first gray block is the *MainEDT*, which creates all other EDTs, and sets dependences between them. Green blocks are *Sequential* Task EDTs, blue blocks are *Trisolve* Task EDTs, red blocks are *Update Non-Diagonal* Task EDTs, and purple blocks are *Update Diagonal* Task EDTs.

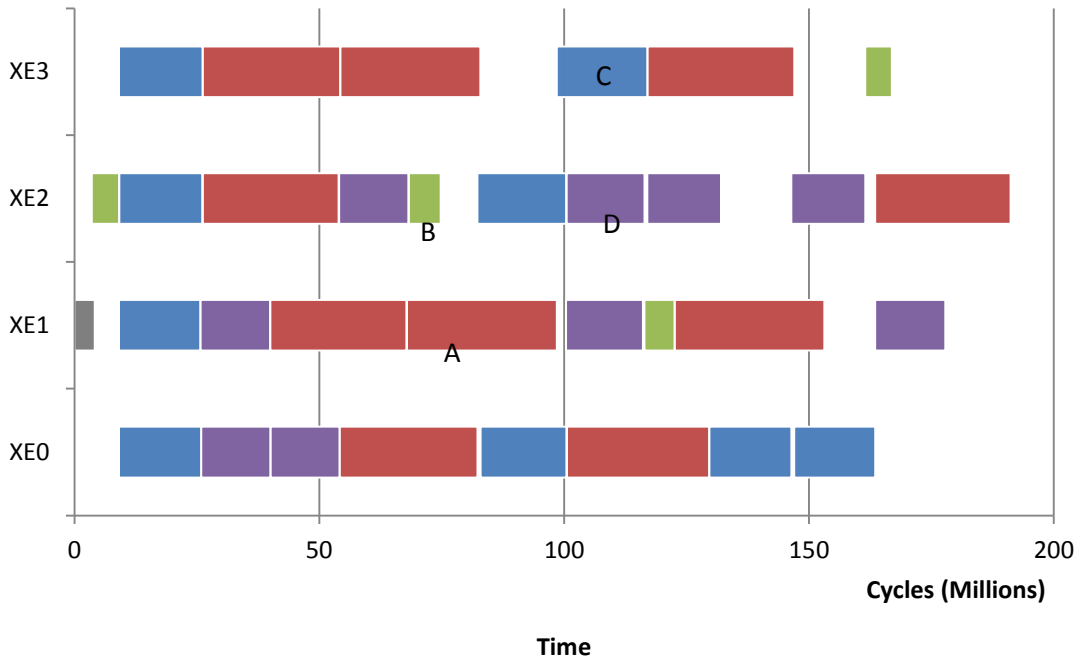


Figure 10.9. EDT scheduling timeline of Cholesky.

Figure 10.9 shows that EDTs do not wait for all the EDTs from the previous iteration to complete. In the figure, EDT A is an *Update Non-Diagonal* EDT from iteration 1, and EDT B, is a *Sequential* EDT in iteration 2. These two EDTs do not have any data dependence, even though they are in different iterations. Consequently, under OCR, they execute concurrently. The same occurs between independent EDTs in a single iteration. For example, EDT C is a *Trisolve* EDT from iteration 2 and EDT D is an *Update Diagonal* EDT from iteration 2. They happen not to have data dependences, and so OCR executes them concurrently. The result is more concurrency than under traditional barrier-based parallelization.

Figure 10.10 shows timeline of *Smith-Waterman* for 4x4 tiles running on 1 block with 4 XEs. We see that the execution is much more concurrent, and more XEs are busy on average. With global barriers, as in Figure 10.2, only tiles of the same color can run in parallel. In Figure 10.10, orange EDTs start execution before all the purple ones finish. Also, pink EDTs extend the time for which 4 XEs are running. With more overlapped execution, OCR enables better exploitation of fined-grained parallelism. The only effect that limits the concurrency is the true dependences between the EDTs.

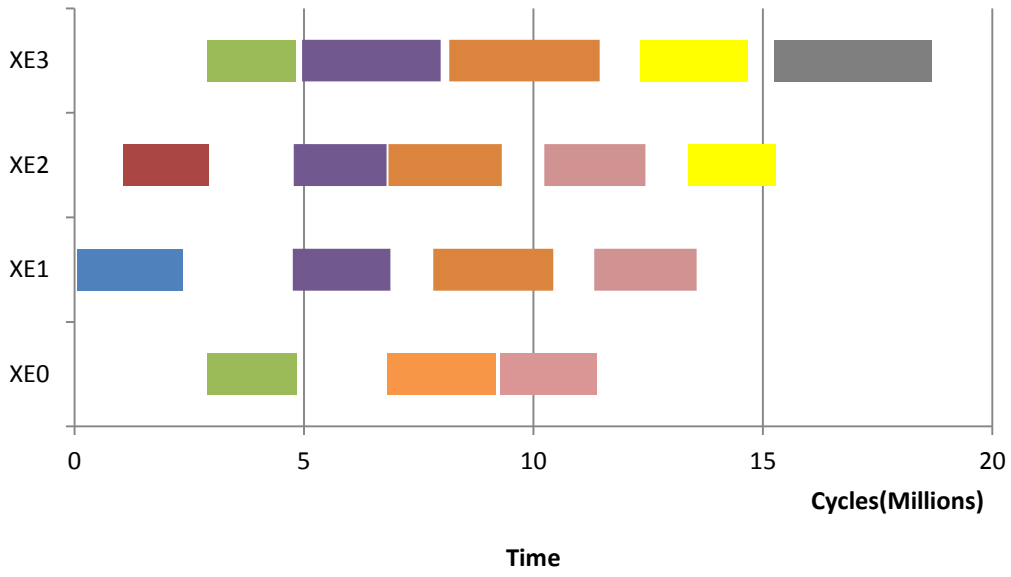


Figure 10.10. EDT scheduling timeline of Smith-Waterman.

Given this environment, we tried to improve the performance by using the hint framework, which schedules EDTs close to the memory module where the corresponding DBs are allocated. Programmers can give hints for scheduling, e.g. giving preference to the location of the DB in the first member in the dependence list. The effect of this hint-based scheduling should be reduced latency and energy. However, we did not find any noticeable benefits of using the hint framework. We observed that when all the XEs are all busy, the scheduler chooses an available XE regardless of the hint. Moreover, when DBs are allocated in the DRAM, the EDT always needs to access DRAM, irrespective of where it is scheduled. Finally, it is hard for programmers to add hints for an EDT with multiple dependences.

8. Managing DBs for Data Locality

In this section, we show the impact of managing the allocation and de-allocation of DBs for data locality. Exploiting locality leads to reduced memory access latency, and reduced energy in memory and network. To illustrate this optimization, we apply it to *Cholesky.*

Cholesky has three distinct DB allocation places in the code (via *ocrDbCreate*). Initially, it allocates DBs for input data. During execution, both *sequential* EDTs and *trisolve* EDTs allocate DBs. The problem size considered is 500 with tile size 50, and we run it on a b1x4 configuration. The program requires more storage than the capacity of the L2 memory in the block (i.e., the BSM memory). Thus, after the BSM is filled with DBs for the input data, all subsequent DB allocation goes to DRAM. Figure 10.11 shows the timeline of the DB allocation during execution. The figure only shows the DB allocation results of *sequential* EDT and *trisolve* EDT. Red blocks mean that the DB is allocated in DRAM. Since the BSM is full with DBs allocated for input data, all DBs that we see in the plot are allocated in DRAM.

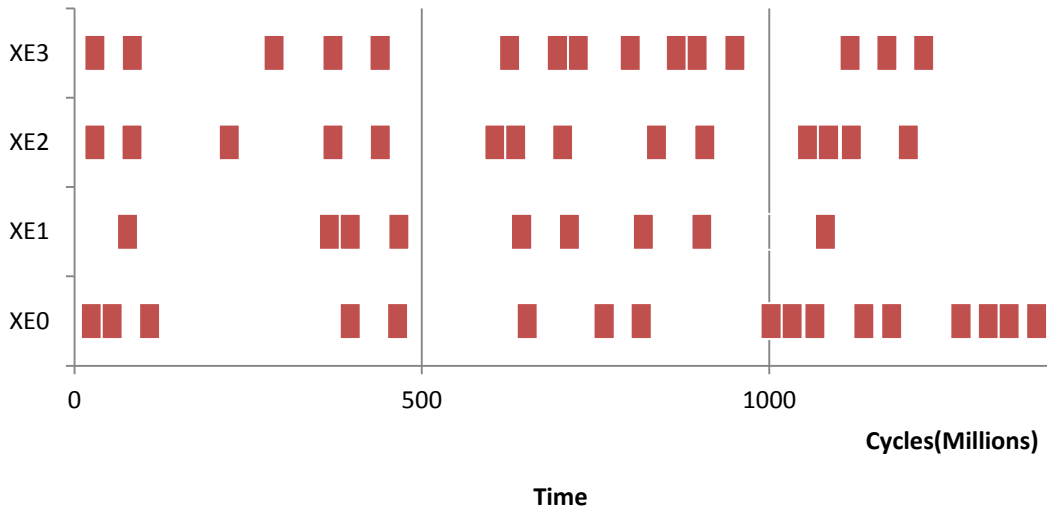


Figure 10.11. DataBlock allocation with the default policy.

With smarter management of on-chip memories, we can reduce DRAM accesses by allocating more DBs in the BSM. This requires securing more capacity in the BSM by moving previously-used DBs to other layers of memory (towards DRAM), or freeing previously-used DBs from the BSM. The former is DB migration; the latter is DB de-allocation. DB migration is not implemented yet in OCR, thus our evaluation is for DB de-allocation.

De-allocation requires careful analysis of future references to the DB. Only when the DB will not be referenced anymore can de-allocation be attempted. Otherwise, it will imply loss of useful DBs. For programs where EDTs allocate temporary storage, DB de-allocation is easy. We only have to insert *ocrDbDestroy* at the end of the EDT. For programs that have one-to-one communication patterns between EDTs, instrumentation is also easy because we know the producer-consumer pairs easily, and the consumer can identify when it can dispose of the DBs. For example, *Smith-Waterman* creates temporary DBs for computation, and also creates DBs for communication. The DBs for temporary use are destroyed by the EDT itself, and the DBs for communication are destroyed by the consumer after use.

Cholesky has more complicated producer-consumer pairs, and shows 1:N producer-consumer relation. We carefully instrument *Cholesky* for DB de-allocation with perfect knowledge of DB use, and see the effect of DB de-allocation. Figure 10.12 shows the new timeline of DB allocation. Blue blocks show BSM-allocated DBs, while red blocks still show DRAM allocation. We can see that some of the DBs are now allocated in the BSM. It can be shown that, after de-allocation from the BSM, we are unable to re-allocate *all* the freed space. This is because of the OCR behavior. OCR shares the same memory allocator as the application. After the application de-allocates a DB, the freed region is sometimes used by the OCR. It would be more intuitive for the programmer if the memory allocation for the OCR is separated from the space allocated by the application.

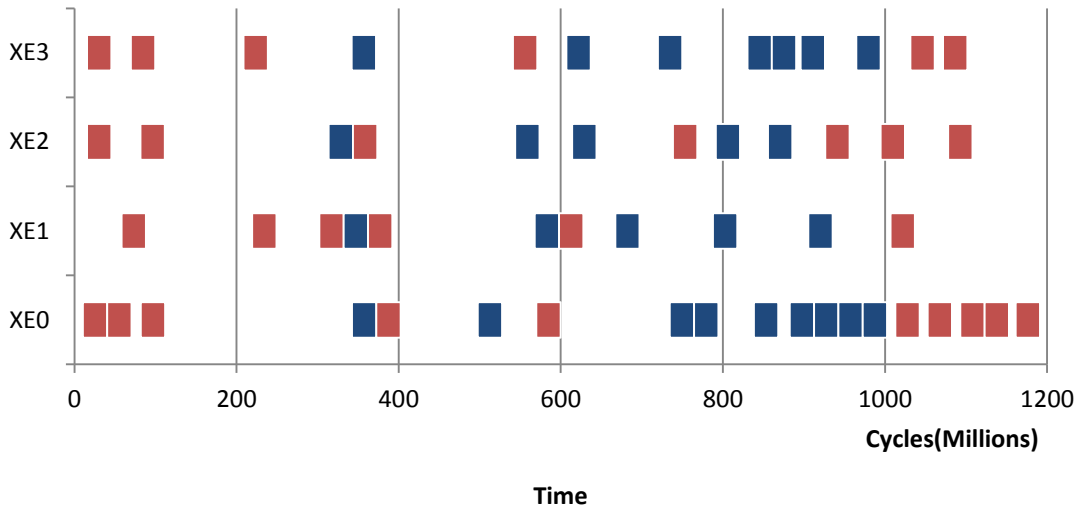


Figure 10.12. DataBlock allocation with de-allocation.

Figure 10.13 shows the breakdown of the memory accesses (loads and stores) with the baseline memory allocation and with the allocation that uses de-allocation first. With the second policy, we see more accesses to the BSM. The figure shows where memory requests are serviced from. Compared to the baseline policy without DB de-allocation, the new policy shows more accesses to the BSM, and reduces DRAM accesses.

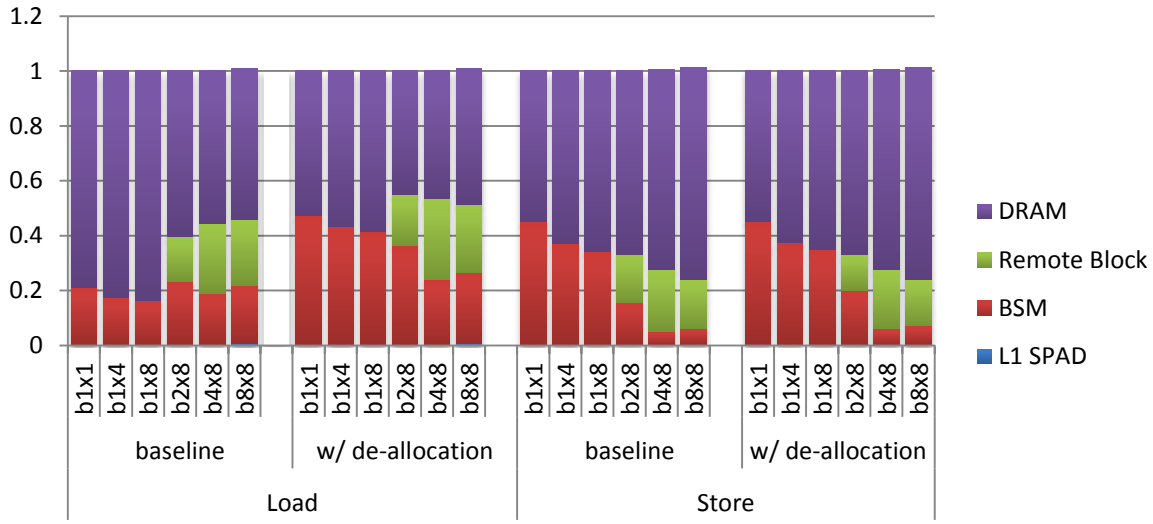


Figure 10.13. Memory access breakdown with different allocation policies.

The result of this now policy is that the program execution time is lower (Figure 10.14) and that the program consumes less energy (Figure 10.15).

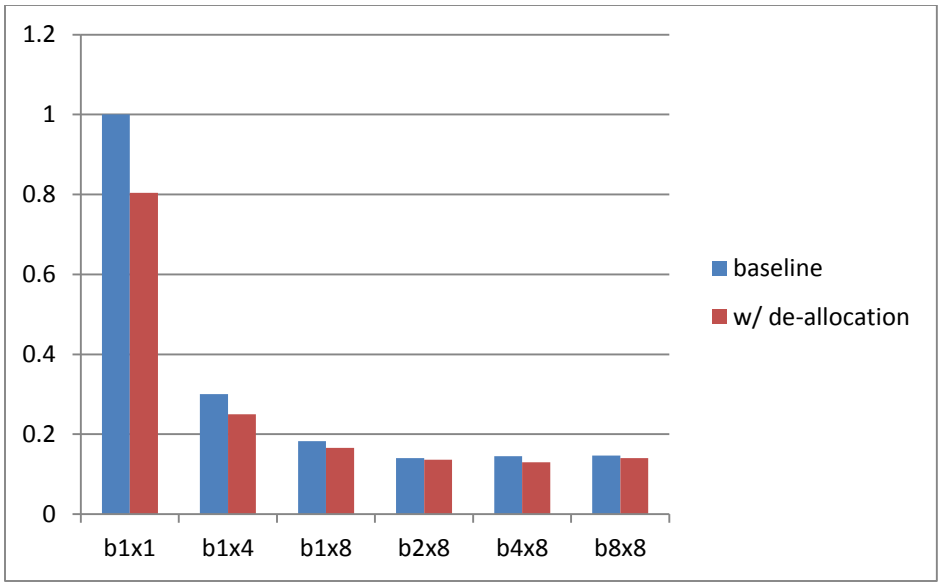


Figure 10.14. Execution time with different DB allocation policies.

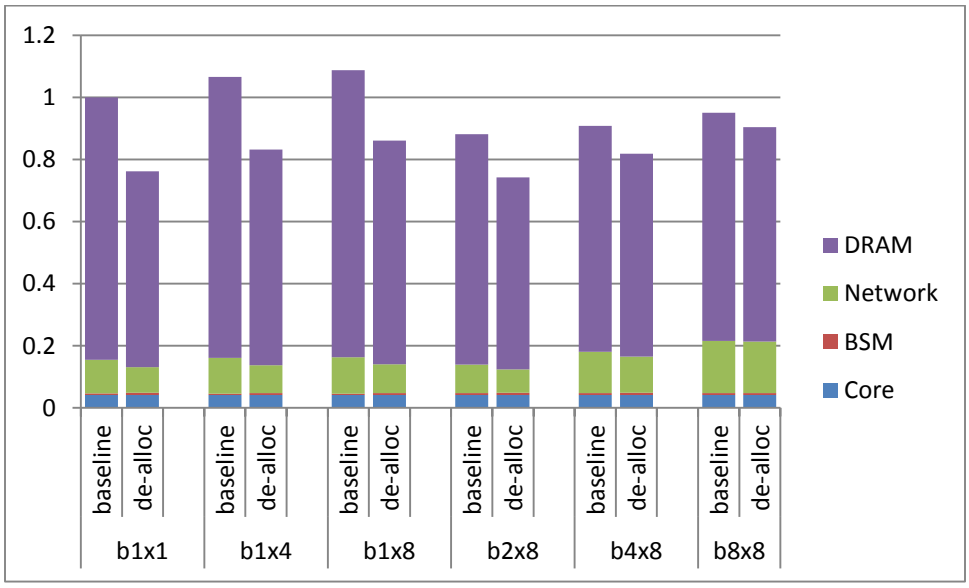


Figure 10.15. Energy consumption with different DB allocation policies.

Recommendations and Future Directions

Future work needs to focus on several directions. First and foremost, we need to improve the efficiency of the OCR implementation on FSIM. It is hard to extract conclusions on the recommended size of EDTs in TG or the

recommended frequency of DB migration in TG unless the system is reasonably tuned. We hope to see improvements over the next few months. Specifically, EDT launch and migration, and DB allocation and migration need to be supported with low overhead. Similarly, the procedure to gather some statistics, such as the execution time, needs to be improved.

It is important to show that effective EDT scheduling for parallelism and DB management for locality can be done *automatically*.

The FSIM simulator needs to be extended to support *Incoherent Caches*, so that a meaningful evaluation of incoherent caches versus scratchpads can be performed in the context of TG.

Products and Bibliography

Wooil Kim, Sanket Tavarageri, P. Sadayappan and Josep Torrellas, "*Architecting and Programming an Incoherent Multiprocessor Cache Hierarchy*", 10 pages, Submitted for publication, 2015.

Sanket Tavarageri, Wooil Kim, Josep Torrellas and P Sadayappan, "*Compiler Support for Software Cache Coherence*", 10 pages, Submitted for publication, 2015.