Domain Specific Language Support for Exascale

Final Report

Rice University Subproject

Cooperative Agreement No. DE-SC0008882

September 2012–August 2016

John Mellor-Crummey

Principal Investigator at Rice University

Department of Computer Science, MS 132
Rice University
P.O. Box 1892
Houston, TX 77251-1892
Voice: 713-348-5179
FAX: 713-348-5930
Email: johnmc@rice.edu

Contents

1 Introduction **1**

2 Summary Description of the Research Performed **1**
2.1 Abstract Machine Model for Code Generation . . . 1
2.1.1 Autotuning for Complex Memory Hierarchies . . . 2
2.1.2 Static Cost Estimation for Data Layout Selection on GPUs . . . 2
2.2 Rosebud DSL Framework . . . 3
2.2.1 Design . . . 4
2.2.2 Implementation . . . 5
2.3 Compiler Technologies . . . 5
2.3.1 Data Layout Transformations for Multicore CPU platforms. . . . 5
2.3.2 Data Layout Transformations for CPU+GPU platforms . . . 6
2.3.3 Polyhedral Optimizations of Explicitly Parallel Programs . . . 7
2.3.4 Polyhedral Optimizations for a Data-flow Graph Language . . . 7
2.3.5 Integrating Polyhedral and AST-based Transformations in ROSE . . . 7
2.3.6 Test-driven Repair of Data Races in Structured Parallel Programs . . . 8
2.3.7 Inter-iteration Scalar Replacement Using Array SSA Form . . . 8
2.3.8 Automatic Parallelization of Pure Method Calls . . . 9
2.4 Runtime Technologies . . . 9
2.4.1 Heterogeneous work-stealing across CPU and DSP cores . . . 9
2.4.2 Dynamic Determinacy Race Detection for Task Parallelism with Futures . . . 9

3 Research Objectives Remaining **10**

4 Findings **10**
4.1 Successes . . . 10
4.2 Lessons . . . 11

5 Products of the Research **13**
5.1 Rosebud . . . 13
5.2 Contributions to Community Open-source Infrastructure . . . 13
5.3 Contributions to Community Standards . . . 13
5.3.1 Contribution of doacross construct to OpenMP 4.5 . . . 13
5.4 Publications . . . 13
5.4.1 Papers . . . 13
5.4.2 Theses . . . 15
5.4.3 Reports . . . 15

1 Introduction

Today, parallel computers are typically programmed using message passing and coarse-grained threads, e.g., MPI and OpenMP. These models are widely used because they can be made to deliver high performance on current platforms. While working with such low-level programming abstractions has been tolerable to date, the expectation is that they will be increasingly problematic as the move to exascale causes software applications to become orders of magnitude more complex. In fact, the 2010 ASCAC report on exascale computing [3] predicts that exascale application developers will face numerous difficult challenges including vastly increased fine-grained parallelism and stringent requirements for exploiting heterogeneous processors, communication avoidance, latency tolerance, adaptive load balancing, fault tolerance, energy efficiency, and performance portability across diverse and changing architectures. It concludes that going to exascale will require radical changes to compilers.

Human programmers cannot hope to manage all of the details necessary to meet the challenges of exascale. Traditionally optimizing compilers have been responsible for managing performance details, but as parallel architectures became more complex, conventional compiler technology for general-purpose programming languages has been unable to keep up. Harnessing the full power of exascale systems with a feasible level of effort by application developers will require a new approach. The ASCAC report concludes that *extreme-scale computing will require performance optimization to be based on a knowledge-oriented process.*

To address this challenge, a multi-institutional project known as D-TEC (short for "Domain-specific Technology for Exascale Computing") set out to explore technologies to support the construction of Domain Specific Languages (DSLs). DSLs employ automated code transformation to shift the burden of delivering portable performance from application programmers to compilers. Two chief properties contribute: DSLs permit expression at a high level of abstraction so that a programmer's intent is clear to a compiler and DSL implementations encapsulate human domain-specific optimization knowledge so that a compiler can be smart enough to achieve good results on specific hardware. Domain specificity is what makes these properties possible in a programming language. If leveraging domain specificity is the key to keep exascale software tractable, a corollary is that many different DSLs will be needed to encompass the full range of exascale computing applications; moreover, a single application may well need to use several different DSLs in conjunction. As a result, developing a general toolkit for building domain-specific languages was a key goal for the D-TEC project.

2 Summary Description of the Research Performed

Different aspects of the D-TEC research portfolio were the focus of work at each of the partner institutions in the multi-institutional project. D-TEC research and development work at Rice University focused on on three principal topics: understanding how to automate the tuning of code for complex architectures, research and development of the Rosebud DSL engine, and compiler technology to support complex execution platforms.

2.1 Abstract Machine Model for Code Generation

An key goal of the D-TEC project is to develop compilers for domain-specific languages that map high-level programs onto future exascale systems systems. To address this issue, the vision for D-TEC called for development of a system that employs rule-based transformations and search guided by machine models. To realize this approach, we must develop a deep understanding of how to tailor programs for key architectural features that we expect to be present in exascale systems. Here, we describe two research efforts that explored aspects of this problem: mapping code onto

complex memory hierarchies and choosing an appropriate data layout to ensure that a code will perform well on GPU architectures.

2.1.1 Autotuning for Complex Memory Hierarchies

In today's parallel systems, tailoring codes for complex memory hierarchies is one of the most difficult code optimization tasks for data-intensive codes. To develop some insights into how to tackle this problem in general, we chose to study the problem of automatically generating high-performance code for tensor transposition. Tensor transposition is a generalization of matrix transposition and is a key library primitive used by the Tensor Contraction Engine. (The Tensor Contraction Engine was developed to provide DSL support for quantum chemistry codes, including NWCHEM.) Tensor transposition is of interest for developing machine models since (1) it is a memory intensive, bandwidth-limited operation that requires careful tailoring to make best use of a machine's memory hierarchy, and (2) next-generation systems are expected to have deeper and more complex memory hierarchies. Efficient implementation of tensor transposition for modern node architectures depends on various architecture capabilities such as cache and memory hierarchy, threads, and SIMD parallelism.

To efficiently map tensor calculations onto node architectures with complex memory systems, we developed a framework that uses static analysis and empirical autotuning to produce optimized parallel tensor transposition code for node architectures using a rule-based code generation and transformation system. Efficiently using a platform's memory hierarchy involves using a collection of techniques (and associated rule-based transformations), including multi-level tiling, in-cache buffers, SIMD operations, non-temporal stores, loop unrolling, and software prefetching. By exploring combinations of these optimizations with various parameter settings, our framework achieves 85–94% of the bandwidth of a multithreaded implementation of the STREAM benchmark on a platform based on Intel Westmere processors. On a Power7-based platform, performance of our generated code is somewhat uneven. While our framework generates code that roughly matches the performance of the STREAM benchmark for several problem sizes; for very large problems, performance drops to 80% of STREAM because of thrashing in the Power7's ERAT. A lesson from this work is that radically-different approaches are sometimes needed to generate efficient code for problems at different scales.

2.1.2 Static Cost Estimation for Data Layout Selection on GPUs

Performance modeling provides mathematical models and quantitative analysis for designing and optimizing computer systems. In high performance architectures, high-latency memory accesses often dominate execution time in many classes of applications. Thus, performance modeling for memory accesses of high performance architectures has been an important research topic. In high performance computation, data layout can significantly affect the efficiency of memory access operations. In recent years, the problem of data layout selection has been well studied on various parallel CPU and some GPU architectures. GPUs have memory hierarchies different from multicore CPUs. While data layout selection on GPUs has been inspected by several existing projects, there is still a lack of a mathematical cost model for data layout selection on GPUs. This motivated us to investigate static cost analysis methods that could better guide future data layout selection work, and perhaps even designing new SIMT architectures. To address this problem, we developed a comprehensive cost analysis for data layout selection for GPUs. We built a cost function based on knowledge of a GPU's memory hierarchy and developed an algorithm that enables developers to perform compile-time cost estimation for a given data layout. Our approach employs a new vector-based representation to represent the estimated cost, which can better estimate the cost of applications with loops that have unknown trip counts. In an evaluation of our cost analysis

approch with benchmarks from past publications on data layout selection, we found that our cost analysis accurately predicts the relative costs of different data layouts, which makes it useful for layout selection.

2.2 Rosebud DSL Framework

Rosebud attacks the two main obstacles to HPC's adoption of DSLs by making them easier to build and to easier to use. *Building* a DSL compiler, like any compiler, is a lot of work and requires special skills. A few parts are automated (e.g. parser construction) and there is some reusable infrastructure (e.g. ROSE). But the hardest parts of a compiler are still coded by hand, existing infrastructure often isn't applicable to new languages, and gluing all the pieces together is tedious and error prone. *Using* a DSL can be a lot of work too, because the implementations are often very poor compared to the production compilers we generally use. DSL source code may have peculiar syntax or present arbitrary restrictions, coupling between DSL and standard code may be cumbersome, compilation may be slow, error messages may be useless or missing, the compiler may be unreliable or hard to maintain, and using a debugger or profiler with DSL code may be frustrating. These deficiencies are usually caused by inadequate resources and skills for the large effort of building a compiler. We need a better way to build better DSLs.

Our approach was to build a comprehensive integrated DSL construction system. Rosebud addresses all aspects of DSL compilation (parsing, semantics, optimization, code generation, runtime support, and tool interfacing), automating what's practical and generating glue code for the rest. Rosebud can be used to build DSLs in any style, with or without custom syntax. In addition, Rosebud was to provide a rich collection of compiler building blocks by wrapping ROSE components into uniform packaging and making it easy to compose them in a DSL compiler, although this was not accomplished. Our hope was that Rosebud's cheaper and better DSLs would result in wider adoption of the DSL programming model and a proliferation of community-developed scientific DSLs, just as today's community develops and shares libraries like Chombo and Zoltan.

During the reporting period at Rice University, we made good progress toward building Rosebud. Our accomplishments include:

- a plugin-based architecture for developing, distributing, combining, and using domain-specific language extensions to standard programming languages (Rosebud)
- a high level declarative notation for defining all aspects of a DSLs implementation including parsing, AST building, optimization, and code generation (Rosebud Definition Language)
- a novel method for extensible language parsing using existing non-extensible language front ends (two phase parsing)
- a novel method for type checking domain-specific languages by leveraging the type system of a surrounding host language (proxy type checking)
- design and partial implementation of a reusable object-oriented framework for generating Rosebud plugins and translating mixed-language source files with them
- preliminary RDL definitions for two host-language plugins (C++ and Fortran 2008)
- integration of the Open Fortran Parser and ROSE software tools to support design and implementation of the Rosebud framework for Fortran related DSLs.

Domain Specific Languages. Many DOE applications use C++ so DSLs on top of the C++ host language are particularly relevant to DOE. We have developed a complete SDF C++11 grammar for Rosebud. The C++ grammar consists of 634 productions and also covers GNU extensions. The grammar will be used as basis for DSL extensions of C++. The rest of this section provides additional detail about our accomplishments.

2.2.1 Design

Two-phase parsing. We devised a two-phase mechanism to use ROSE's existing language front ends for parsing and static analysis of source files containing passages written in multiple DSLs, despite the front ends lack of extensibility. Phase 1 first parses the source file with a merged grammar for the host language and DSLs. A technique similar to island parsing discards the structure of host language passages while building full ASTs for the DSL passages. Then, guided by source position information from the parser, the input file is copied in order to replace DSL passages by markers, yielding a pure host-language source file acceptable to the rose front end. Markers are distinctive host language constructs of the same syntactic classes as the DSL fragment they replace; for instance, a DSL expression passage might be replaced by a host function call to a specially named function. Rosebud adds declarations for these markers so the rose front end sees semantically correct source code. Phase 2 invokes the rose front end to re-parse the marked-up source file, perform static semantic analysis, and build a rose AST. Lastly, it replaces marker AST subtrees by corresponding DSL ASTs from Phase 1 to produce the final mixed-language rose AST.

Proxy type checking. We devised a new DSL type checking mechanism which provides a simple declarative definition style for DSL authors, automatic consistency of a DSL's type system with that of the host it extends, and a simple, robust, and efficient implementation exploiting roses existing language front ends. Our approach is to restrict DSL type semantics so that domain-specific type checking can be mapped to generic overload resolution in the host language. This limits the expressiveness of Rosebud DSLs somewhat, but we believe it is a reasonable tradeoff for simplicity and inter-language consistency. The same marker mechanism used in two-phase parsing is the basis for proxy type checking. We map a DSL's domain-specific types to host language proxy types and construct generic marker declarations corresponding to the DSLs declarative type rules. When the rose front end performs host language type checking on marker passages, it effectively performs domain-specific type checking on corresponding DSL passages. To obtain the DSL type checking results we simply map inferred proxy types of markers back to domain-specific types. Rosebud Definition Language (RDL). We designed the integrated DSL specification language RDL to serve as source code from which Rosebud plugins are generated. RDL is syntactically patterned after the SDF and Stratego languages, with module structure for encapsulation and module syntax based on sections introduced by keywords. Currently RDL has nine kinds of sections:

1. DSL metadata	6. Compile-time code
2. Concrete syntax	7. Rewriting system
3. Embedded syntax	8. Run-time code
4. Abstract syntax	9. Prettyprint rules
5. Static semantics	

Concrete syntax is specified in SDF, embedded abstract syntax in regular tree grammars, static semantics in a C++-like template notation, rewriting system in Stratego, pretty printing in the gpp language, and code in C++.

2.2.2 Implementation

We made substantial progress toward an initial version of Rosebud. Below we outline the status of our implementation:

- Infrastructure. Rosebud exploits existing open source code for much of its functionality. We have incorporated the following: Boost Library for data structures and utilities; SDF2 for parsing and tree building; ROSE for host language processing, analysis, and transformation; Stratego/XT for rewriting systems; POCO Zip for structured storage; and POCO Class Loader for executable components of plugins.

- Autotools build system. We completed construction of a GHU Autotools build system for: Rosebud's Generator, Translator, and Plugin Utility tools; Host and Building Block plugins; LATEX documentation; example DSL definitions; test cases; and the Rosebud infrastructure. All code is housed in an svn repository at Rice University.

- Executable tools. We completed implementation of the RDL grammar and IR and the Rosebud plugin architecture. We completed detailed top-down skeletons of all Rosebud tools which compile and run; each tool implements command line processing, error handling, plugin io, and a high level sketch of its algorithm. In addition, the tools completely implement RDL definition parsing, generation of DSL parsers from RDL grammar sections, plugin-based two-level parsing of mixed-language source files, and selective listing and dumping of plugin contents for debugging.

- Host language plugins. We completed preliminary implementation of host language plugins for C++ and Fortran 2008 based on SDF grammars for these languages built by the LLNL and Oregon teams.

Example DSLs. We completed the RDL definition of a pedagogic DSL named Stacks which extends a host language with custom syntax for working with pushdown stacks. We can generate a plugin from its RDL and use the plugin to parse C++ and Fortran source files containing Stacks passages; however, its type checking and rewriting systems cannot yet be tested. We began RDL definition of a plugin for the SDSL stencil language contributed by OSU; so far we have converted its original ANTLR grammar to RDL but have not yet used it for parsing.

2.3 Compiler Technologies

In this research thrust, we explored a range of compiler technologies that employ sophisticated program analysis and transformations to map programs efficiently to parallel systems. The goal of this work was to develop building block technologies to support efficient mapping of programs to exascale architectures.

2.3.1 Data Layout Transformations for Multicore CPU platforms.

As part of the D-TEC project, Rice continued a collaboration with Ian Karlin, Jeff Keasler, and James McGraw at LLNL to develop a new approach to managing array data layouts to optimize performance for scientific codes executing on multicore CPU nodes. This project led to a redesign and reimplementation of TALC—a data layout tool previously developed by Jeff Keasler. It is well known that changing data layouts can lead to significant performance improvements. However, there have been two major reasons why such optimizations are not currently used in production applications: (1) the need to select different layouts for different computing platforms, and (2) the cost of re-writing codes to use new layouts. This work addresses both obstacles, and enables data layout optimization to also be applied to code generated from a DSL. Our new data layout tool

is built upon LLNL's ROSE compiler infrastructure, which provides a source-to-source translation process that allows us to generate codes with different array interleavings from the same source code, based on different data layout specifications. The ability to specify different data layout specifications can be viewed as an embedded mini-DSL in the context of the D-TEC project. We used our implementation to generate 19 different data layouts for an ASC benchmark code (IRSmk) and 32 different data layouts for the LULESH mini-application. Performance results for multicore versions of the benchmarks with different layouts showed significant benefits on four computing platforms (IBM POWER7, AMD APU, Intel SandyBridge, IBM BG/Q). For IRSmk, our results showed performance improvements ranging from 22.23% on IBM POWER7 to 1.10% on Intel SandyBridge. For LULESH, we saw improvements ranging from 1.82% on IBM POWER7 to 1.02% on Intel SandyBridge. We also developed a new automatic optimization algorithm to recommend a layout for an input source program and specific target machine characteristics.

Our results show that the performance of this automated layout algorithm outperforms the manual layouts in one case and performs within 10% of the best architecture-specific layout in all other cases but one. Our implementation in ROSE has been checked into the trunk of the ROSE project, and a technical paper is available on this work [7]. This work also formed a central part of Ph.D. student Kamal Sharma's dissertation at Rice [6].

2.3.2 Data Layout Transformations for CPU+GPU platforms

In addition to optimizing data layouts for CPUs, we also explored the impact of data layoust on heterogeneous CPU+GPU platforms. This work extends our previously mentioned collaboration with LLNL by taking into account the fact that the optimal layout for a computational kernel depends on whether the kernel executes on a CPU core, a discrete GPU, or on an integrated GPU. For instance, the GPU memory performance is impacted by upon the number of coalesced memory accesses, whereas CPU memory performance is impacted by factors such as false sharing and data reuse. Since changes in data layout can impact CPU vs. GPU performance, in general, the programmer has to write different versions of CPU and GPU kernels for different architectures and has to select optimal memory layouts for each. This places a severe constraint on code portability.

In this work, we first implemented a compiler-driven data layout transformation framework for heterogeneous platforms. We introduce a meta-data framework which enables the same source code to be compiled with different data layouts without involving the programmer in the data layout transformations. Our compiler and runtime infrastructure generates efficient code for different architectures based on the meta information. Our experimental results show significant benefits from this approach, and demonstrate that the best data layout for a program is different for CPU vs. GPU execution. On an average, the data layout transformation alone impacted the performance by 7.33% (up to 27.11%) on AMD 4-core A10-5880K CPU, 2.84% (up to 5.57%) on AMD Radeon integrated GPU, 8.32% (up to 29.5%) on NVIDIA Tesla M2050 GPU, 2.19% (up to 5.32%) on Intel 12-core X5660 CPU and 1.9% (up to 3.89%) on Intel integrated i7-3770 GPU for a set of 5 distinct benchmarks. We then introduced a two-level hierarchal formulation of the data layout problem for modern heterogeneous architectures. The lower level formulation deals with optimal data layout problem for a parallel code region (section). Our analysis of this problem showed it to be NP-Hard. For that reason, we developed a greedy algorithm that uses an affinity graph to obtain approximate solutions. The higher level formulation targets data layouts for the entire program, for which we provide an optimal solution using a graph-based shortest path algorithm that uses the data layouts for the code regions computed in the lower level. We have implemented solutions to this two-level hierarchal formulation of the data layout problem in the Habanero-C parallel programming system, implemented in the ROSE compiler infrastructure, and demonstrated significant performance benefits of up to 6.92% (on average 3.11%) compared to the

manually specified layouts for a set of parallel programs running on a CPU+GPU heterogeneous platform. A key contribution of this work was extending ROSE to automatically generate OpenCL code from the Habanero-C forasync parallel loop construct.

2.3.3 Polyhedral Optimizations of Explicitly Parallel Programs

The polyhedral model is a powerful algebraic framework that has enabled significant advances to analysis and transformation of sequential affine (sub)programs, relative to traditional AST-based approaches. However, given the rapid growth of parallel software, there is a need for increased attention to using polyhedral frameworks to optimize explicitly parallel programs. An interesting side effect of supporting explicitly parallel programs is that doing so can also enable optimization of programs with unanalyzable data accesses within a polyhedral framework. In this work, we explored how to extend polyhedral frameworks to enable analysis and transformation of programs that contain both explicit parallelism and unanalyzable data accesses. As a first step, we focused on OpenMP loop parallelism and task parallelism, including task dependences from OpenMP 4.0. Our approach first enables conservative dependence analysis of a given region of code. Next, we identified happens-before relations from the explicitly parallel constructs, such as tasks and parallel loops, and intersected them with the conservative dependences. Finally, we pass resulting set of dependences to a polyhedral optimizer, such as PLuTo and PolyAST, to enable transformation of explicitly-parallel programs with unanalyzable data accesses. We evaluate our approach using eleven OpenMP benchmark programs from the KASTORS and Rodinia benchmark suites. We show that 1) these benchmarks contain unanalyzable data accesses that prevent polyhedral frameworks from performing exact dependence analysis, 2) explicit parallelism can help mitigate the imprecision, and 3) polyhedral transformations with the resulting dependences can further improve the performance of the manually-parallelized OpenMP benchmarks. Our experimental results show geometric mean performance improvements of 1.62x and 2.75x on the Intel Westmere and IBM Power8 platforms respectively (relative to the original OpenMP versions).

2.3.4 Polyhedral Optimizations for a Data-flow Graph Language

This paper proposes a novel optimization framework for the Data-Flow Graph Language (DFGL), a dependence-based notation for macro-dataflow model which can be used as an embedded domain-specific language. Our optimization framework follows a dependence-first approach in capturing the semantics of DFGL programs in polyhedral representations, as opposed to the standard polyhedral approach of deriving dependences from access functions and schedules. As a first step, our proposed framework performs two important legality checks on an input DFGL program checking for potential violations of the single-assignment rule, and checking for potential deadlocks. After these legality checks are performed, the DFGL dependence information is used in lieu of standard polyhedral dependences to enable polyhedral transformations and code generation, which include automatic loop transformations, tiling, and code generation of parallel loops with coarse-grain (fork-join) and fine-grain (doacross) synchronizations. Our performance experiments with nine benchmarks on Intel Xeon and IBM Power7 multicore processors show that the DFGL versions optimized by our proposed framework can deliver up to 6.9x performance improvement relative to standard OpenMP versions of these benchmarks. To the best of our knowledge, this is the first system to encode explicit macro-dataflow parallelism in polyhedral representations so as to provide programmers with an easy-to-use DSL notation with legality checks, while taking full advantage of the optimization functionality in state-of-the-art polyhedral frameworks.

2.3.5 Integrating Polyhedral and AST-based Transformations in ROSE

The polyhedral model is an algebraic framework for affine program representations and transformations for enhancing locality and parallelism. Compared with traditional AST-based transformation

frameworks, the polyhedral model can easily handle imperfectly nested loops and complex data dependences within and across loop nests in a unified framework. On the other hand, AST-based transformation frameworks for locality and parallelism have a long history that dates back to early vectorizing and parallelizing compilers. They can be used to efficiently perform a wide range of transformations including hierarchical parametric tiling, parallel reduction, scalar replacement and unroll- and-jam, and the implemented loop transformations are more compact (with smaller code size) than polyhedral frameworks. While many members of the polyhedral and AST-based transformation camps see the two frameworks as a mutually exclusive either-or choice, we demonstrate that both frameworks can be integrated in a synergistic manner. In this work, we obtained early results with integrating polyhedral and AST-based transformations in ROSE. Our preliminary experiments demonstrate the benefits of the proposed combined approach relative to Pluto, a pure polyhedral framework for locality and parallelism optimizations.

2.3.6 Test-driven Repair of Data Races in Structured Parallel Programs

A common workflow for developing parallel software is as follows: 1) start with a sequential program, 2) identify subcomputations that should be converted to parallel tasks, 3) insert synchronization to achieve the same semantics as the sequential program, and repeat steps 2) and 3) as needed to improve performance. Though this is not the only approach to developing parallel software, it is sufficiently common to warrant special attention as parallel programming becomes ubiquitous. In this work, we focus on automating step 3), which is usually the hardest step for developers who lack expertise in parallel programming.

Past solutions to the problem of repairing parallel programs have used static-only or dynamic-only approaches, both of which incur significant limitations in practice. Static approaches can guarantee soundness in many cases but are limited in precision when analyzing medium or large-scale software with accesses to pointer-based data structures in multiple procedures. Dynamic approaches are more precise, but their proposed repairs are limited to a single input and are not reflected back in the original source program. To address this problem, we developed a hybrid static+dynamic test-driven approach to repairing data races in structured parallel programs. Our approach includes a novel coupling between static and dynamic analyses. First, we execute the program on a concrete test input and determine the set of data races for this input dynamically. Next, we compute a set of "finish" placements that prevent these races and also respects the static scoping rules of the program while maximizing parallelism. Empirical results on standard benchmarks and student homework submissions from a parallel computing course establish the effectiveness of our approach with respect to compile-time overhead, precision, and performance of the repaired code.

2.3.7 Inter-iteration Scalar Replacement Using Array SSA Form

Scalar replacement is an strategy for mappin array elements into scalar variables to facilitate analysis and optimization of loops with complex array indexing. In this work, we developed novel simple and efficient analysis algorithms that scalar replacement and dead store elimination across loop iterations. Our approach leverages Array SSA form, which is a uniform representation for capturing control and data flow properties at the level of array or pointer accesses. We use extensions to the original Array SSA form representation to capture loop-carried data flow information for arrays and pointers. A core contribution of our algorithm is a subscript analysis that propagates array indices across loop iterations. Compared to past work, our new algorithm can handle control flow within and across loop iterations and degrade gracefully in the presence of unanalyzable subscripts. We also introduce code transformations that can use the output of our analysis algorithms to perform the necessary scalar replacement transformations (including the insertion of loop prologues

and epilogues for loop-carried reuse). Our experimental results show performance improvements of up to 2.29x relative to code generated by LLVM at -O3 level. These results promise to make our algorithms a desirable starting point for scalar replacement implementations in modern SSA-based compiler infrastructures such as LLVM.

2.3.8 Automatic Parallelization of Pure Method Calls

We developed a novel approach for using futures to automatically parallelize the execution of pure method calls. Our approach is built on three new techniques to address the challenge of automatic parallelization via future synthesis: candidate future synthesis, parallelism benefit analysis, and threshold expression synthesis. During candidate future synthesis, our system annotates pure method calls as async expressions and synthesizes a parallel program with future objects and their type declarations. Next, the system performs a parallel benefit analysis to determine which async expressions may need to be executed sequentially due to overhead reasons, based on execution profile information collected from multiple test inputs. Finally, threshold expression synthesis uses the output from parallelism benefit analysis to synthesize predicate expressions that can be used to determine at runtime if a specific pure method call should be executed sequentially or in parallel.

We implemented our approach and the results from an experimental evaluation of the complete system on a range of sequential Java benchmarks are very encouraging. Our evaluation shows that our approach can provide significant parallel speedups of up to 7.4x (geometric mean of 3.69x) relative to the sequential programs when using 8 processor cores, with zero programmer effort beyond providing the sequential program and test cases for parallelism benefit analysis.

2.4 Runtime Technologies

In addition to our work on compiler technologies for parallel programs, we also investigated a collection of complementary runtime technologies that include scheduling and error detection.

2.4.1 Heterogeneous work-stealing across CPU and DSP cores

Due to the increasing power constraints and higher and higher performance demands, many vendors have shifted their focus from designing high-performance computer nodes using powerful multicore general-purpose CPUs, to nodes containing a smaller number of general-purpose CPUs aided by a larger number of more power-efficient special purpose processing units, such as GPUs, FPGAs or DSPs. While offering a lower power-to-performance ratio, unfortunately, such heterogeneous systems are notoriously hard to program, forcing the users to resort to lower-level direct programming of the special purpose processors and manually managing data transfer and synchronization between the parts of the program running on general-purpose CPUs and on special-purpose processors. In this paper, we present HC-K2H, a programming model and runtime system for the Texas Instruments Keystone II Hawking platform, consisting of 4 ARM CPUs and 8 TI DSP processors. This System-on-a-Chip (SoC) offers high floating-point performance with lower power requirements than other processors with comparable performance. We present the design and implementation of a hybrid programming model and work-stealing runtime that allows tasks to be created and executed on both the ARM and DSP, and enables the seamless execution and synchronization of tasks regardless of whether they are running on the ARM or DSP. The design of our programming model and runtime is based on an extension of the Habanero-C programming system. We evaluate our implementation using task-parallel benchmarks on a Hawking board, and demonstrate excellent scaling compared to sequential implementations on a single ARM processor.

2.4.2 Dynamic Determinacy Race Detection for Task Parallelism with Futures

Existing dynamic determinacy race detectors for task-parallel programs are limited to programs with strict computation graphs, where a task can only wait for its descendant tasks to complete. In

this paper, we present the first known determinacy race detector for non-strict computation graphs, constructed using futures. The space and time complexity of our algorithm are similar to those of the classical SP-bags algorithm, when using only structured parallel constructs such as spawn-sync and async-finish. In the presence of point-to-point synchronization using futures, the complexity of the algorithm increases by a factor determined by the number of future task creation and get operations as well as the number of non-tree edges in the computation graph. The experimental results show that the slowdown factor observed for our algorithm relative to the sequential version is in the range of 1.00x 9.92x, which is in line with slowdowns experienced for strict computation graphs in past work.

3 Research Objectives Remaining

The end goal of the D-TEC project was to develop a sophisticated framework for accelerating the construction of new implementations of domain-specific languages. Unfortunately, it was infeasible to realize this grand vision with the limited time and resources allocated to this project. Implementing a compiler and runtime system for any domain-specific language, even a small one, is a significant undertaking. Developing a general framework for automating the synthesis of parallel implementations of arbitrary domain-specific languages was a step too far given the current state of technology. While research and development efforts in the D-TEC project at Rice University yielded valuable component technologies necessary to realize our grand vision, our grasp was shorter than our reach. At present, all of the component technologies require significant further research for them to be useful for tailoring domain-specific languages to exascale platforms. Our result in each of the three dimensions of the project fell short of what was needed.

First, while D-TEC research to understand how to map programs efficiently to complex architectures using empirical autotuning was successful in within restricted domains, significantly more work is needed to develop a more general autotuning framework necessary to tailor applications for all of the complex features expected in exascale platforms.

Second, although Rosebud provides a rich framework for specifying many aspects of domain-specific extensions to conventional programming languages, the lack of time and resources along with inadequate preparation of the compiler framework intended as its host left us unable to fully integrate Rosebud with a compiler framework to support analysis and optimization of its domain-specific abstractions.

Finally, while compiler and runtime technologies developed in the course of this project provide useful building blocks for efficiently mapping parallel programs onto complex parallel architectures, an incomplete palatte of compiler technologies and the lack of a general framework for cost estimation left us well short of the project goal of supporting a rule-based DSL compiler framework that employed cost models and domain knowledge to tailor applications to complex parallel platforms.

4 Findings

4.1 Successes

Rosebud's front end architecture. Rosebud implements a custom-syntax multi-DSL mixin programming model inspired by the Stratego/XT group's Metaborg project [**?**]. Our plugin based design for extensible parsing and static semantic analysis of such programs is technically feasible and a major advance over Metaborg's implementation strategy. It is straightforward to implement and sufficiently fast and robust for practical use. Organizing the front end in phases and using an object-oriented style for plugins does permit easy run-time extension with new lexical, syntactic, and semantic analyses for a DSL. SGLR parsing is fast enough for interactive use even on the very large source files found in legacy HPC codes and is capable of detecting and parsing DSL syntax

extensions even when nested. Building a unified AST for a source file with multiple DSL mixins is both natural and useful.

Reuse of compiler front ends. Our proxy type checking strategy for employing an existing compiler front end for host language analysis and to type-check interactions between host and DSL code is effective in practice. This important improvement over Metaborg permits us to use complex modern host languages like C++ without reimplementing the language's entire front end, which would be prohibitive. Our research demonstrated Rosebud reuse of both C++ and Fortran compiler front ends, essential for HPC. We expect reusing other languages' compilers to be equally straightforward.

Rosebud Definition Language. Using a single meta-DSL to define all aspects of a mixin DSL, another improvement over Metaborg, is technically feasible. It is expressive and easy to use, to implement, and to extend as research uncovers new requirements.A unified notation simplifies modular development by enabling aspect-oriented factoring of lexical, syntactic, semantic, optimization, code generation, and tool interface details. Basing the notation on well-delimited sections, each with a custom micro-syntax, provides clarity and flexibility. Including notation to specify interfaces to externally developed components makes it easy for a DSL to incorporate existing code for complex or costly computations such as polyhedral analysis or multiprecision arithmetic.

Abstract DSLs. Rosebud introduces the notion of an "abstract" DSL, one whose programs are expressed in abstract syntax (as AST fragments) rather than concrete syntax. Such programs are not written by humans but generated by rewriting an existing AST during Rosebud's translation process. Abstract DSLs are useful to encapsulate back end optimization and code generation mechanisms. For instance, rewriting a Fortran program might convert MPI subroutine calls into custom AST fragments in an abstract MPI DSL. Isolated in this way, MPI operations could be examined, optimized, and converted to code by the MPI DSL using its language-independent encapsulated expertise. Other mechanisms which might benefit from abstract-DSL encapsulation are OpenMP, polyhedral optimization, coprocessor acceleration, resilience, symbolic algebra, and so on. Although we were unable to build any abstract DSLs, we are convinced by our design and implementation experience and discussions with other D-TEC and X-Stack researchers that this idea is powerful and practical.

4.2 Lessons

Compiler integration requirements. We knew at the outset that reuse of an existing host language compiler would require some Rosebud mechanisms to be integrated tightly with the compiler's front end and AST; examples are converting AST fragments to and from the rewriting engine's format and reanalyzing the static semantics of AST fragments after rewriting. We addressed this by targeting the ROSE Compiler Infrastructure, written and maintained by our D-TEC collaborators at LLNL, rather than an open source production compiler like Clang. Our work plan was predicated on LLNL's making the necessary changes to ROSE, but that was never done. Consequently we were unable to build the promised Rosebud back end, and so unable to evaluate experimentally the productivity and portable performance gains Rosebud is designed to provide. The lesson is that compiler integration can be a show stopper if adequate resources are not available.

Collaboration and effort. We also knew at the outset that building and evaluating Rosebud would require much help from the other institutions involved in D-TEC. For instance, LLNL and IBM would help with front end work for C++/Fortran and for X10 respectively; Berkeley would help build and evaluate a stencil DSL in Rosebud; UCSD would help build an MPI DSL; and OSU would help build a stencil optimizer for Rosebud. None of these tasks were accomplished despite repeated attempts to enlist cooperation. These omissions were aggravated by a pervasive

shortage of software development staff; our already very lean proposed budget was decimated in the DOE funding process without corresponding reduction in work scope, resulting in a severely undermanned effort at Rice.

Reuse of program analysis infrastructure. We did not appreciate at the outset just how difficult it would be to reuse an existing compiler's program analysis and optimization mechanisms. The chief difficulty is that existing optimizers hard-code assumptions about what constitutes a constant, variable, or expression. In Rosebud's mixin-DSL model, these constructs also include domain-specific variants (e.g. a DSL's sparse array references used as variables). Even if a DSL's author specifies the data and control flow semantics of its constructs, the inflexibility of existing optimizer implementations make it challenging to modify one for use by Rosebud. We didn't get far enough into Rosebud's back end to need this, but we now recognize it as an issue. Part way through D-TEC a collaborator began looking at this issue with promising early results, so we do not see it as an obstacle to eventually achieving Rosebud's goals.

Ambiguity in the front end. We also failed to appreciate at the outset how big an issue syntactic ambiguity would turn out to be. It was obvious that adding two DSL mixins to a source program might cause inter-DSL ambiguities, and we had a plan to handle this. However, we assumed that the host language – some existing production programming language – would be unambiguous in practice. This is certainly true from a user's viewpoint, but unfortunately it is not true for the BNF grammars used to describe their concrete syntaxes. For instance, C++ is highly ambiguous at the context-free level and disambiguation is done during static analysis, too late to help Rosebud's mixed language parser. These intra-host ambiguities interact badly with DSL syntax extensions and interfere with identification of DSL code passages. Fortunately, we were able to understand this issue and devise a straightforward, efficient technique for handling it. In brief, the parser tracks all possible parses for each construct (for free in an SGLR parser) and discards insignificant ambiguities – ones which affect only the parsing of host-language code. These may be ignored because the host compiler disambiguates them later. Most ambiguities encountered in practice are this kind. Significant ambiguities are those within a single DSL, between two DSLs, or between a DSL and the host language. We treat the presence of a significant ambiguity as a syntax error. The record of possible parses allows Rosebud to emit reasonable error messages and the user can fix these errors with appropriate parenthesization. Pruning of insignificant ambiguities is done on the fly during parsing, preventing a combinatorial explosion of space and time costs.

Defining DSL type semantics. To type check DSL code and its interaction with host code, Rosebud uses a novel technique we call proxy type checking. We map a Rosebud type checking problem to a corresponding host language proxy, let the host compiler resolve it, and map the proxy answer back to the Rosebud problem. This works well and greatly simplifies the specification of type semantics: instead of writing formal type semantics, an author just provides a host-language proxy template for each DSL construct. Unfortunately, even this tuns out to be hard to express in RDL at present. We need to look for a simpler, more expressive notation for such proxy mappings.

Interleaving cost based searches. Although we never got a chance to work on Rosebud's back end, one of its most interesting and challenging aspects was our proposed use of interleaved cost based searches for joint optimization of multiple DSLs in a single source file. In our vision, each DSL construct is translated by search over possible translations using estimated execution cost to guide the search, and searches for distinct constructs are interleaved so that a (tentative) rewriting decision for one construct can be taken into account in the cost metric of another construct's search. For instance, if in one alternative world a stencil loop has decided to load some array onto the GPU, then the search trying to translate a subsequent stencil loop might assign low cost to having that

array on GPU but a higher cost to having other arrays on the GPU (because the latter but not the former would require additional GPU io). As we looked at possible implementations we realized that to make the technique practical we'd have to find a way of structuring the rewriting system to use some kind of transactions. A single decision such as putting an array on GPU may be implemented through a sequence of rewrite rules, but for interleaving to achieve the desired effect each interleaved order must take or leave that sequence as a whole. This research issue remains to be explored in future work.

Multiple host languages. Finally, we overlooked at the outset that many practical HPC applications involve running code written in two or more languages; for instance, Chombo programs are mostly C++ but employ loop kernels written in Fortran, and GPU programs may consist of host language code plus kernels written in CUDA. Thus Rosebud needs to be able to translate a single mixed-language source program into multiple output files written in different languages. Our original design, like Metaborg before it, assumed that one input source file is translated into one output file. We have worked out most of the details of this extension and believe it is straightforward to implement.

5 Products of the Research

5.1 Rosebud

Rosebud is open source with a permissive license. Its source code and documentation are available at `https://svn.rice.edu/r/rosebud`.

5.2 Contributions to Community Open-source Infrastructure

Mature results of the compiler work were incorporated into the ROSE compiler infrastructure, available from `http://rosecompiler.org`.

5.3 Contributions to Community Standards

5.3.1 Contribution of doacross construct to OpenMP 4.5

During the period of this project, two members of the D-TEC team at Rice, research scientist Dr. Jun Shirako and co-PI Prof. Vivek Sarkar, worked with IBM on developing a proposal for adding the *doacross* construct to OpenMP [5]. This proposal was presented and refined over the course of multiple OpenMP committee meetings, before it appeared as "multidimensional extensions to ordered constructs" in the OpenMP 4.5 standard in November 2015. In addition to the various benefits of this construct known from past work [1, 8], our research has shown that the doacross construct can be a useful target for code generated from DSLs [4] or from polyhedral optimizers [2].

5.4 Publications

5.4.1 Papers

- Kamal Sharma, Ian Karlin, Jeff Keasler, James R. McGraw, Vivek Sarkar. User-Specified and Automatic Data Layout Selection for Portable Performance, Technical Report TR13-03, Department of Computer Science, Rice University, 2013.

- Deepak Majeti, Rajkishore Barik, Jisheng Zhao, Max Grossman, and Vivek Sarkar. Compiler-Driven Data Layout Transformation for Heterogeneous Platforms, International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Platforms (HeteroPar 2013, co-located with EuroPar 2013), August 2013.

- Jun Shirako, Vivek Sarkar. Oil and Water can mix! Experiences with integrating Polyhedral and AST-based Transformations, 17th Workshop on Compilers for Parallel Programming (CPC), July 2013.

- Deepak Majeti, Kuldeep S. Meel, Rajkishore Barik, and Vivek Sarkar. 2014. ADHA: automatic data layout framework for heterogeneous architectures. In Proceedings of the 23rd international conference on Parallel architectures and compilation (PACT '14). ACM, New York, NY, USA, 479-480. DOI: http://dx.doi.org/10.1145/2628071.2628122

- Lai Wei and John Mellor-Crummey. 2014. Autotuning Tensor Transposition. In Proceedings of the 2014 IEEE International Parallel & Distributed Processing Symposium Workshops (IPDPSW '14). IEEE Computer Society, Washington, DC, USA, 342-351. DOI: http://dx.doi.org/10.1109/IPDPSW.2014.43.

- Rishi Surendran, Raghavan Raman, Swarat Chaudhuri, John Mellor-Crummey, and Vivek Sarkar. Test-driven repair of data races in structured parallel programs. In Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14). ACM, New York, NY, USA, 15-25. DOI: http://dx.doi.org/10.1145/2594291.2594335.

- Rishi Surendran, Rajkishore Barik, Jisheng Zhao, Vivek Sarkar: Inter-iteration Scalar Replacement Using Array SSA Form. A. Cohen (Ed.): CC 2014, LNCS 8409, pp. 4060, 2014. DOI: https://doi.org/10.1007/978-3-642-54807-9_3

- Rishi Surendran and Vivek Sarkar. Dynamic Determinacy Race Detection for Task Parallelism with Futures. In: Falcone Y., Snchez C. (eds) Runtime Verification. RV 2016. Lecture Notes in Computer Science, vol 10012. Springer, Cham DOI: https://doi.org/10.1007/978-3-319-46982-9_23

- Xu Liu, Kamal Sharma, John Mellor-Crummey, ArrayTool: a lightweight profiler to guide array regrouping, Proceedings of the 23rd International Conference on Parallel Architectures and Compilation, August 24-27, 2014, Edmonton, AB, Canada. DOI: https://doi.org/10.1145/2628071.2628102.

- Jun Shirako, Louis-Noël Pouchet, and Vivek Sarkar. Oil and water can mix: an integration of polyhedral and AST-based transformations. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '14). IEEE Press, Piscataway, NJ, USA, 287-298. DOI: https://doi.org/10.1109/SC.2014.29.

- Kamal Sharma, Ian Karlin, Jeff Keasler, James R McGraw, Vivek Sarkar. Data Layout Optimization for Portable Performance. Proceedings of Euro-Par 2015: Parallel Processing: 21st International Conference on Parallel and Distributed Computing, Vienna, Austria, August 24-28, 2015. Lecture Notes in Computer Science book series (LNCS, volume 9233). Springer Berlin Heidelberg. DOI: https://doi.org/10.1007/978-3-662-48096-0_20

- Prasanth Chatarasi, Jun Shirako, and Vivek Sarkar. Polyhedral Optimizations of Explicitly Parallel Programs. In Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT) (PACT '15). IEEE Computer Society, Washington, DC, USA, 213-226. DOI: https://doi.org/10.1109/PACT.2015.44

- Alina Sbîrlea, Jun Shirako, Louis-Noël Pouchet, and Vivek Sarkar. Polyhedral Optimizations for a Data-Flow Graph Language, Languages and Compilers for Parallel Computing: 28th International Workshop, LCPC 2015, Raleigh, NC, USA, September 9-11, 2015, Revised Selected Papers 2016, pp. 57–72. Springer International Publishing, Cham. https://doi.org/10.1007/978-3-319-29778-1_4

- Y. Peng, M. Grossman and V. Sarkar. Static Cost Estimation for Data Layout Selection on GPUs. 7 th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS), Salt Lake, UT, 2016, pp. 76-86. DOI: https://doi.org/10.1109/PMBS.2016.013.

- Prasanth Chatarasi, Jun Shirako, Martin Kong, and Vivek Sarkar. An Extended Polyhedral Model for SPMD Programs and Its Use in Static Data Race Detection The 29th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2016), September 28–30, 2016. Rochester, NY. ChapterinLecture Notes in Computer Science 10136:106-120. In book: Languages and Compilers for Parallel Computing, pp.106-120 DOI: https://doi.org/10.1007/978-3-319-52709-3_10.

- Rishi Surendran and Vivek Sarkar. Automatic parallelization of pure method calls via conditional future synthesis. In Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016). ACM, New York, NY, USA, 20-38. DOI: https://doi.org/10.1145/2983990.2984035.

5.4.2 Theses

- Kamal Gopal Sharma. Locality Transformations of Computation and Data for Portable Performance. Ph.D. Dissertation. Department of Computer Science, Rice University, August 2014. http://hdl.handle.net/1911/88159.

- Lai Wei. Autotuning Memory-intensive Software for Node Architectures. Master's Thesis. Department of Computer Science, Rice University. May 2015. http://hdl.handle.net/1911/88422.

- Sbirlea, Alina. High-level execution models for multicore architectures. Ph.D. Dissertation. Department of Computer Science, Rice University, November 2015. http://hdl.handle.net/1911/88142.

- Rishi Surendran. Debugging, Repair, and Synthesis of Task Parallelism. Ph.D. Dissertation. Department of Computer Science, Rice University, March 2017 http://hdl.handle.net/1911/96003.

5.4.3 Reports

- K. Sharma, I. Karlin, J. Keasler, J. McGraw, V. Sarkar. User-Specified and Automatic Data Layout Selection for Portable Performance. Technical Report LLNL-TR-637873, Lawrence Livermore National Laboratory, May 30, 2013.

References

[1] R. Cytron. Useful Parallelism in a Multiprocessing Environment. *Proc. 1985 International Conference on Parallel Processing*, pages 450–457, 1985.

[2] Prasanth Chatarasi, Jun Shirako and Vivek Sarkar. Polyhedral Optimizations of Explicitly Parallel Programs. In *Proc. of The 24th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, San Francisco, CA, USA, 2015.

[3] R. Rosner et al. The opportunities and challenges of exascale computing. *US Dept. of Energy Office of Science, Summary Report of the Advanced Scientific Computing Advisory Committee (ASCAC) Subcommittee, November*, 2010. http://science.energy.gov/~/media/ascr/ascac/pdf/reports/Exascale_subcommittee_report.pdf.

[4] A. Sbîrlea, J. Shirako, L.-N. Pouchet, and V. Sarkar. Polyhedral optimizations for a data-flow graph language. In *The 28th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, Sept 2015.

[5] D. Sbîrlea, A. Sbîrlea, K. B. Wheeler, and V. Sarkar. The Flexible Preconditions Model for Macro-Dataflow Execution. In *The 3rd Data-Flow Execution Models for Extreme Scale Computing Workshop (DFM)*, Sep 2013.

[6] K. Sharma. *Locality Transformations of Computation and Data for Portable Performance*. PhD thesis, Rice University, Aug 2014.

[7] K. Sharma, I. Karlin, J. Keasler, J. R. McGraw, and V. Sarkar. Data layout optimization for portable performance. In *Euro-Par 2015: Parallel Processing*, pages 250–262. Springer, 2015.

[8] P. Unnikrishnan, J. Shirako, K. Barton, S. Chatterjee, R. Silvera, and V. Sarkar. A practical approach to doacross parallelization. In *Proceedings of the 18th International Conference on Parallel Processing*, Euro-Par'12, pages 219–231, Berlin, Heidelberg, 2012. Springer-Verlag.