Experiences Developing the OpenUH Compiler and Runtime Infrastructure

Barbara Chapman and Deepak Eachempati
University of Houston

Oscar Hernandez
Oak Ridge National Laboratory

***Abstract*—The OpenUH compiler is a branch of the open source Open64 compiler suite for C, C++, Fortran 95/2003, with support for a variety of targets including x86_64, IA-64, and IA-32. For the past several years, we have used OpenUH to conduct research in parallel programming models and their implementation, static and dynamic analysis of parallel applications, and compiler integration with external tools. In this paper, we describe the evolution of the OpenUH infrastructure and how we've used it to carry out our research and teaching efforts.**

I. Introduction

At the University of Houston, we are pursuing a pragmatic agenda of research into parallel programming models and their implementation. Our research interests span language support for application development on high-end systems through embedded systems. Our practical work considers both the need to implement these languages efficiently on current and emerging platforms as well as support for the application developer during the process of creating or porting a code. These activities are complemented by coursework, primarily at the graduate level, that explores the use of programming languages for parallel computing as well as their design and implementation.

Starting roughly ten years ago, we began a program of research into language enhancements and novel implementation strategies for OpenMP [30], a set of compiler directives, runtime library routines and environment variables, that is the de-facto programming standard for parallel programming in C/C++ and Fortran on shared memory and distributed shared memory systems. We also were interested in learning how to exploit compiler technology to facilitate the process of OpenMP application development, with the goals of reducing the human labor involved and helping avoid the introduction of coding errors. Since that time, our research interests have broadened to encompass a range of parallel programming models and their implementation, as well as strategies for more extensive support for parallel application creation and tuning.

In order to enable experimentation, to ensure that we understand the implementation challenges fully, and to demonstrate success on real-world applications, we strove to implement our ideas in a robust compiler framework. Moreover, we decided to realize a hybrid approach, where portability is achieved via a source-to-source translation, but where we also have a complete compiler that is able to generate object code for the most widely used ABIs. This permits us to evaluate our results in a setting that is typical of industrial compilers. Within the context of OpenMP, for instance, our ability to generate object code helps us experiment to determine the impact of moving the relative position of the OpenMP lowering within the overall translation, and allows us to experiment with a variety of strategies for handling loop nests and dealing with resource contention. It is of great value in our research into feedback optimizations. Given the high cost of designing this kind of compiler from scratch, we searched for an existing open-source compiler framework that met our requirements. We chose to base our efforts on the Open64 [1] compiler suite, which we judged to be more suitable for our purposes than, in particular, the GNU Compiler Collection [13] in their respective states of development.

In this paper, we describe the experiences of our research group in building and using OpenUH, a portable open source compiler based on the Open64 compiler infrastructure. OpenUH has a unique hybrid design that combines a state-of-the-art optimizing infrastructure with the option of a source-to-source approach. OpenUH supports C/C++ and Fortran 90, includes numerous analysis and optimization components, and offers a complete implementation of OpenMP 3.0 as well as near-complete implementations of Unified Parallel C (UPC) and Coarray Fortran (CAF). It includes a CUDA translation to NVIDIA's PTX format and supports automated instrumentation as well as providing additional features for deriving dynamic performance information and carrying out feedback optimizations. It is also the basis for a tool called Dragon that supplies program information to the application developer and is designed to meet the needs of program maintenance and porting. We hope that this compiler (which is available at [31]) will complement other existing compiler frameworks and offer a further attractive choice to parallel application developers, language and compiler researchers and other users.

The reminder of this paper is organized as follows. Section 2 describes the Open64 compiler infrastructure and Section 3 gives an overview of our OpenUH compiler. It presents some details of the research that it has enabled, while the following section briefly discusses our experiences using it in teaching and training.

II. Overview of Open64

Open64 is a well-written, modularized, robust, state-of-the-art compiler with support for C/C++ and Fortran 77/90. The major modules of Open64 are the multiple language front-ends, the inter-procedural analyzer (IPA) and the middle end/back end, which is further subdivided into the loop nest optimizer (LNO), global optimizer (WOPT), and code generator (CG). Five levels of a tree-based intermediate representations (IR) called WHIRL exist to support the implementation of different analysis and optimization phases. They are classified as being Very High, High, Mid, Low, and Very Low levels, respectively. Open64 also includes two IR-to-source translators named *whirl2c* and *whirl2f* which can be useful for debugging and also, potentially, leveraged for source-to-source compiler translation.

Open64 originated from the SGI MIPSPro compiler for the MIPSR10000 processor, and was open-sourced as Pro64 in 2000 under the GNU public license. The University of Delaware became the official host for the compiler, now called Open64, in 2001 and continue to host the project today. Over the past 10 years, Open64 has matured into a robust, optimizing compiler infrastructure with wide contributions from industry and research institutions. Intel and the Chinese Academy of Sciences partnered early on to develop the Open Research Compiler (ORC) which implemented a number of code generator optimizations and improved support for the Itanium target. A number of enhancements and features from the QLogic PathScale compiler was also merged in, including support for an x86 back-end.

Open64 has an active developer community including participants from industry and academic institutions. For example, NVIDIA used Open64 as a code optimizer in their CUDA toolchain. AMD is active in enhancing the loop nest optimizer, global optimizer, and code generator. HP has long been active in maintaining the compiler and supporting related research projects using Open64. Universities currently working on Open64 projects include, but are not limited to, University of Houston, Tsinghua University, the Chinese Academy of Sciences, National Tsing-Hua University, and University of California, Berkeley. For the past several years, an annual Open64 workshop has been held to provide a forum for developers and users to share their experiences and on-going research efforts and projects. As a member of the Open64 Steering Group (OSG), we engage other lead Open64 developers in the community to help make important decisions for the Open64 project including event organization, source check-in and review policies, and release management.

III. Overview of OpenUH

The OpenUH [24] compiler is a branch of the open source Open64 compiler suite for C, C++, Fortran 95/2003, supporting the IA-64, IA-32, Opteron Linux ABI, and PTX generation for NVIDIA GPUs. Fig. 1 depicts an overview of the design of OpenUH based on Open64. It consists of the front-ends with support for OpenMP 3.0 and Coarray Fortran (CAF), optimization modules, back-end lowering phases for OpenMP and coarrays, portable OpenMP and CAF runtimes, a code generator and IR-to-source tools. Most of these modules are derived from the corresponding original Open64 modules. OpenUH may be used as a source-to-source compiler for other machines using the IR-to-source tools.

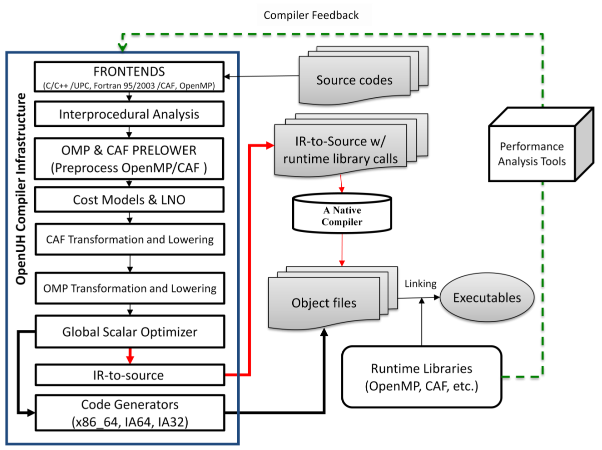

Fig. 1: The OpenUH Compiler/Runtime Infrastructure

We have undertaken a broad range of infrastructure development in OpenUH to support important topics such as language research, static analysis of parallel programs, performance analysis, task scheduling, and dynamic optimization [2, 19, 21, 14, 16]. We also investigated techniques for retargeting OpenMP applications to distributed memory architectures and more recently systems with heterogeneous cores [11, 8]. OpenUH also includes support for a tool called Dragon [9], which supports the export of application information in the forms of call graphs/trees, procedure control flow, call-site details, OpenMP usage, data dependence information, and more. Dragon is able to respond to user requests for information about an application code and responds by producing graphical displays along with the corresponding source code.

In the following sections we describe some of the research infrastructure we've developed in OpenUH to support our projects.

A. *OpenMP Support*

OpenMP is a fork-join parallel programming model with bindings for C/C++ and Fortran 77/90 to provide additional shared memory parallel semantics. The OpenMP extensions consist primarily of compiler directives (structured comments that are understood by an OpenMP compiler) for the creation of parallel programs; these are augmented by user-level runtime routines and environment variables. Its popularity stems from its ease of use, incremental parallelism, performance portability and wide availability. Recent research at language and compiler levels, including our own, has considered how to expand the set of target architectures to include recent system configurations, such as SMPs based on Chip Multithreading processors [25], as well as clusters of SMPs [17]. However, in order to carry out such work, a suitable compiler infrastructure must be available. In order for application developers to be able to explore OpenMP on the system of their choice, a freely available, portable implementation would be desirable.

Many compilers support OpenMP today, including proprietary products such as the Intel compilers, Sun Studio compilers, and SGI MIPSpro compilers. However, their source code is mostly inaccessible to researchers and they cannot be used to gain an understanding of OpenMP compiler technology or to explore possible improvements to it. Several open source research compilers (including Omni [32], Mercurium [3], Cetus [23], and Rose [26]) have been developed. But none of them translate all of the source languages that OpenMP supports, and most of them are source-to-source translators with reduced scope for optimization. Our goal in developing OpenUH was to provide a robust OpenMP compiler for C/C++/Fortran which can generate high-level source code and also generate optimized binaries.

We describe a selection of recent developments in OpenUH to support OpenMP research.

1) Data Flow Analysis: In past work [18, 21], we have designed and implemented an extension to the data flow analysis framework (PDFA) in OpenUH to describe data flow between concurrently executing threads in an OpenMP parallel region. For this work, we implemented a Parallel Control Flow Graph (PCFG) for representing OpenMP programs in order to enable aggressive optimizations, while guaranteeing correctness. The PCFG is not unlike the Program Execution Graph and the Synchronized Control Flow Graph proposed by other researchers [4, 7]. The distinction between our PCFG and their flow-graph is that ours is based upon the relaxed memory consistency model of OpenMP, and its barrier and flush synchronizations instead of event-based synchronizations (such as signal-wait). We have also added support for Parallel SSA (PSSA) form, an extension of SSA that represents more complex data that occurs in an OpenMP parallel program. We incorporate ψ- and π-functions into our representation, based in part on work by Lee et al. [22].

2) Locality Language Extension: We are currently working to implement our proposed location feature in the OpenUH compiler and its runtime. We have extended the OpenMP runtime to support location and data layout management [20]. We are currently building our runtime on top of libnuma, which is a library in Linux systems to support NUMA systems, to implement thread binding and location creation, and will look for more portable library support for our implementation in the next step. We are working on minimizing the overheads of our current implementation of the location feature. The compiler handles the syntax: its front end has been extended to parse the current syntax, and the middle end translates the new OpenMP directives and clauses to multithreaded code by modifying the code and inserting calls to the extended runtime.

3) Tasking Implementation: OpenUH includes support for OpenMP 3.0 tasks. This consists of front-end support ported from the GNU C/C++ compiler, back-end translation we implemented jointly with Tsinghua University, and an efficient task scheduling infrastructure we have developed in our runtime library. We have implemented a configurable task pool framework that allows the user to choose at runtime an appropriate task queue organizations to use. This framework also allows for fast prototyping of new task pool designs. Furthermore, the user may control the order in which tasks are removed from a task queue for greater control over task scheduling. We have merged these recent improvements in our runtime (including improved nested parallelism and tasking support) into the official OpenMP 3.0 branch in the Open64.net source repository.

4) Runtime Enhancements: We have implemented a collector tool API for OpenMP applications within our runtime [6, 15]. Performance tools may issue requests to the runtime library as the application is running to query important state information. To satisfy these requests, we have added support for (1) initiate/pause/resume/stop even generation, (2) responding to queries for the ID of the current/parent parallel region, and (3) responding to queries for the current state of the calling thread. By running an application with a performance tool that uses this API, the user can uncover important information about their program, for example synchronization costs. Another feature we've added is to make available several different implementations of barrier operations. We have extended our runtime to accept a user-specified barrier algorithm best suited for the application's needs on a specific architecture and given number of threads.

B. Instrumentation

OpenUH provides a complete compile-time instrumentation module covering different compilation phases and different program scopes. Advanced feedback-guided analyses and optimizations are part of the compiler for sequential tuning. We have designed a compiler instrumentation API that can be used to instrument a program. It is language independent to enable it to interact with performance tools such as TAU [27] and KOJAK [28] and support the instrumentation of Fortran, C and C++.

Compile-time instrumentation has several advantages over both source-level and object-level instrumentation. Compiler analysis can be used to detect regions of interest before instrumenting and measuring certain events to support different performance metrics. For instance, analysis can be used to detect parallel loops and combine this analysis with the instrumentation, to provide feedback-guided parallelization. In addition, the compiler analysis is used to provide the correct mapping for instrumenting optimized code and how it is related to the source code. OpenUH allows the user to perform instrumentation at different compilation phases, allowing some optimizations to take place before the instrumentation. These capabilities play a significant role in the reduction of instrumentation points, improve user's ability to deal with program optimizations, and reduce the instrumentation overhead and size of performance traces or data collected.

The instrumentation module in OpenUH can be invoked at six different phases during compilation, which come before and after three major stages in the translation: inter-procedural analysis, loop nest optimizations, and SSA optimizations. For each phase, the following kinds of user regions can be instrumented: functions, conditional branches, switch statements, loops, call sites, and individual statements. Each user-region type is further divided into subcategories when possible.

For instance, a loop may be of type do-loop, while-loop. Conditional branches may be of type if-then, if-then-else, true-branch, false-branch, or select. MPI operations are instrumented via PMPI so that the compiler does not instrument these call sites. OpenMP constructs are handled via runtime library instrumentation, where it captures the fork and joint events, implicit and explicit barriers. Procedure and control ow instrumentation is essential to relate the MPI and OpenMP-related output to the execution path of the application, or to understand how constructs behave inside these regions.

The compiler instrumentation is performed by first translating the intermediate representation of an input program to locate different program constructs. The compiler inserts instrumentation calls at the start and exit points of structured control flow operators such as procedures, branches and loops. If a region has multiple exit points, they will all be instrumented; for example, `GOTO`, `STOP` or `RETURN` statements may provide alternate exit points.

C. Coarray Fortran Support

CAF support in OpenUH [12] comprises three areas: (1) an extended front-end accept the coarray syntax and related intrinsic functions, (2) back-end optimization and translation, and (3) a portable runtime library.

1) Front-end: We modified the Cray Fortran 95 front-end used by OpenUH to support our coarrays implementation. Cray had provided some support for CAF syntax, but its approach was to perform the translation to the underlying runtime library in the front-end. It accepted the [] syntax in the parser, recognized certain CAF intrinsics, and it targeted a SHMEM-based runtime with a global address space. In order to take advantage of the analysis and optimizing capabilities in the OpenUH back-end, we needed to preserve the coarray semantics into the back-end. To accomplish this, we adopted a similar approach to that used in Open64/SL Fortran front-end from [10], where co-subscripts are preserved in the IR as extra array subscripts. We also added support for CAF intrinsic functions such as `this_image`, `num_images`, `image_index`, and more as defined in the Fortran 2008 standard.

2) Back-end: We have in place a basic implementation for coarray lowering in our back-end and are in the midst of adding an analysis/optimization phase. One of the key benefits of the CAF programming model is that programs are amenable to aggressive compiler optimizations. The back-end also consists of a prelowering phase which normalizes the IR emitted from the front-end to facilitate dependence analysis. This will enable many optimizations, including hoisting potentially expensive coarray accesses out of loops and generating non-blocking communication calls where it is feasible and profitable.

3) Runtime: The implementation of our supporting runtime system relies on an underlying communication subsystems provided by ARMCI [29] or GASNet [5]. We have adopted both the ARMCI and GASNet libraries for most communication and synchronization operations required by the CAF execution model. This work entails memory management for coarray data, communication facilities provided by the runtime, and support for synchronizations specified in the CAF language. We have also added preliminary implementation of reductions in the runtime.

IV. OpenUH in Teaching and Learning

We have used our compiler infrastructure to support our instructional efforts in a graduate course offered to Computer Science students. In that context, the richness of this infrastructure has made it a valuable resource. For example, we have illustrated our discussion of the roles and purposes of various levels of intermediate representation by showing how OpenUH represents selected constructs and simple executable statements. We are able to get students to apply certain features and then output source code. Even though some help is needed to explain the structure of this output code, it offers insight into the manner in which the compiler applies transformations. Moreover, students have routinely successfully carried out minor adaptations to the compiler, or retrieved specific information from its internal structures. Last, they have used it to explore strategies for implementing state-of-the-art parallel programming models, including but not limited to our own work.

References

[1] The Open64 compiler. http://www.open64.net, 2011.

[2] C. Addison, J. LaGrone, L. Huang, and B. Chapman. OpenMP 3.0 tasking implementation in OpenUH. In *Open64 Workshop in Conjunction with the International Symposium on Code Generation and Optimization*, 2009.

[3] J. Balart, A. Duran, M. Gonzalez, X. Martorell, E. Ayguade, and J. Labarta. Nanos Mercurium: a research compiler for OpenMP. In *the 6th European Workshop on OpenMP (EWOMP'04)*, Stockholm, Sweden, October 2004.

[4] V. Balasundaram and K. Kennedy. Compile-time detection of race conditions in a parallel program. In *ICS '89: Proceedings of the 3rd international conference on Supercomputing*, pages 175–185, Crete, Greece, June 1989. ACM Press.

[5] D. Bonachea. Gasnet specification, v1.1. Technical report, Berkeley, CA, USA, 2002.

[6] V. Bui, O. Hernandez, B. Chapman, R. Kufrin, D. Tafti, and P. Gopalkrishnan. Towards an implementation of the openmp collector api. In *PARCO*, 2007.

[7] D. Callahan, K. Kennedy, and J. Subhlok. Analysis of event synchronization in a parallel programming tool. In *PPOPP '90: Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming*, pages 21–30, Seattle, Washington, USA, March 1990. ACM Press.

[8] S. Chandrasekaran, O. Hernandez, D. Maskell, B. Chapman, and V. Bui. Compilation and parallelization techniques with tool support to realize sequence alignment

algorithm on fpga and multicore. In *Workshop on New Horizons in Compilers*, 2007.

[9] B. Chapman, O. Hernandez, L. Huang, T.-H. Weng, Z. Lui, L. Adhianto, and Y. Wen. Dragon: An Open64-based interactive program analysis tool for large applications. In *Proceedings of the Fourth International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT'2003)*, pages 792–796. IEEE Press, 2003.

[10] Y. Dotsenko, C. Coarfa, and J. Mellor-Crummey. A multi-platform co-array fortran compiler. In *PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 29–40, Washington, DC, USA, 2004. IEEE Computer Society.

[11] D. Eachempati, L. Huang, and B. M. Chapman. Strategies and implementation for translating OpenMP code for clusters. In R. H. Perrott, B. M. Chapman, J. Subhlok, R. F. de Mello, and L. T. Yang, editors, *HPCC*, volume 4782 of *Lecture Notes in Computer Science*, pages 420–431. Springer, 2007.

[12] D. Eachempati, H. J. Jun, and B. Chapman. An Open-Source Compiler and Runtime Implementation for Coarray Fortran. In *PGAS'10*, New York, NY, USA, Oct 12-15 2010. ACM Press.

[13] the GNU compiler collection. http://gcc.gnu.org, 2005.

[14] O. Hernandez and B. Chapman. Compiler support for efficient profiling and tracing. In *Parallel Computing 2007 (ParCo 2007))*, September 2007.

[15] O. Hernandez, R. C. Nanjegowda, B. M. Chapman, V. Bui, and R. Kufrin. Open source software support for the openmp runtime api for profiling. In *ICPP Workshops*, pages 130–137, 2009.

[16] O. R. Hernandez. *Efficient performance tuning methodology with compiler feedback*. PhD thesis, Houston, TX, USA, 2008. AAI3313493.

[17] L. Huang, B. Chapman, and R. Kendall. OpenMP on distributed memory via Global Arrays. In *Parallel Computing 2003 (PARCO 2003)*, DRESDEN, Germany, 2003.

[18] L. Huang, D. Eachempati, M. W. Hervey, and B. Chapman. Extending global optimizations in the openuh compiler for openmp. In *Open64 Workshop at CGO 2008, In Conjunction with the International Symposium on Code Generation and Optimization (CGO)*, Boston, MA, April 2008.

[19] L. Huang, H. Jin, L. Yi, and B. Chapman. Enabling locality-aware computations in OpenMP. *Scientific Programming*, 18(3):169–181, 2010.

[20] L. Huang, H. Jin, L. Yi, and B. M. Chapman. Enabling locality-aware computations in openmp. *Scientific Programming*, 18(3-4):169–181, 2010.

[21] L. Huang, G. Sethuraman, and B. Chapman. Parallel data flow analysis for openmp programs. In *Proceedings of IWOMP 2007*, June, 2007.

[22] J. Lee, D. A. Padua, and S. P. Midkiff. Basic compiler algorithms for parallel programs. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'99)*, pages 1–12, Atlanta, Georgia, USA, August 1999. ACM SIGPLAN.

[23] S. I. Lee, T. A. Johnson, and R. Eigenmann. Cetus - an extensible compiler infrastructure for source-to-source transformation. In *LCPC*, pages 539–553, 2003.

[24] C. Liao, O. Hernandez, B. Chapman, W. Chen, and W. Zheng. OpenUH: An optimizing, portable OpenMP compiler. In *12th Workshop on Compilers for Parallel Computers*, January 2006.

[25] C. Liao, Z. Liu, L. Huang, and B. Chapman. Evaluating OpenMP on chip multithreading platforms. In *First international workshop on OpenMP*, Eugene, Oregon USA, June 2005.

[26] C. Liao, D. J. Quinlan, T. Panas, and B. R. de Supinski. A rose-based openmp 3.0 research compiler supporting multiple runtime libraries. In M. Sato, T. Hanawa, M. S. Müller, B. M. Chapman, and B. R. de Supinski, editors, *IWOMP*, volume 6132 of *Lecture Notes in Computer Science*, pages 15–28. Springer, 2010.

[27] A. D. Malony, S. Shende, R. Bell, K. Li, L. Li, and N. Trebon. Advances in the tau performance system. *Performance analysis and grid computing*, pages 129–144, 2004.

[28] B. Mohr and F. Wolf. KOJAK - a tool set for automatic performance analysis of parallel applications. In *Proc. of the European Conference on Parallel Computing (EuroPar)*, pages 1301–1304, 2003.

[29] J. Nieplocha and B. Carpenter. ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems. In *Proceedings of the 11 IPPS/SPDP'99 Workshops Held in Conjunction with the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, pages 533–546. Springer-Verlag, 1999.

[30] OpenMP: Simple, portable, scalable SMP programming. http://www.openmp.org, 2006.

[31] The OpenUH compiler project. http://www.cs.uh.edu/~openuh, 2005.

[32] M. Sato, S. Satoh, K. Kusano, and Y. Tanaka. Design of openmp compiler for an smp cluster. In *In EWOMP '99*, pages 32–39, 1999.