Mercurium:
Design Decisions for a S2S Compiler

Roger Ferrer[1], Sara Royuela[1,2], Diego Caballero[1,2], Alejandro Duran[1], Xavier Martorell[1,2], and Eduard Ayguade[1,2]

[1]Barcelona Supercomputing Center, Nexus II, Jordi Girona 29, Barcelona, Spain
[2]Departament d'Arquitectura de Computadors, Univ. Politecnica de Catalunya, Jordi Girona, 1–3, Barcelona, Spain

I. Introduction

Current research on heterogeneous multi- and many-core architectures requires powerful compilers. They must provide flexibility to try new language constructs, and be extensible, to increase the capacity of experimentation with different program transformations. And they must be easy to use, to reduce as much as possible the introduction of defects in the transformations, and allow rapid prototyping.

Since 1996, our group has been doing research on program transformations useful to extract parallelism from serial codes, and also improve serial execution. In this period, we have mainly used three compiler platforms to base our work: Parafrase-2 [12], from the Univ. of Illinois at Urbana-Champaign; we did an attempt to use the Open64 infrastructure; and currently we are using Mercurium, our own approach for research on program transformations [4], [6]. Mercurium has gone through several revisions, starting with a C compiler (mcc, 2004), a Fortran95 infrastructure (mf95, 2005), and to the current approach for C/C++ (mcxx, since 2007), and our current work on Fortran 95 (mf95, 2011).

We have used this infrastructure in the research for the proper OpenMP extensions to deal with Tasking, Error Management, and User-Defined Reductions, in collaboration with the OpenMP ARB. Ours was the first implementation of OpenMP Tasking. Regarding Error Management and User-Defined Reductions, we expect our contributions to be eventually included in a later specification of OpenMP.

Mercurium has supported in-house developments, such as the support of the StarSs collection of extensions – CellSs [11] for the heterogeneous Cell/B.E. architecture, SMPSs [13], [3] for SMP environments, GPUSs for environments with GPUs, and a prototype generating code for FPGAs. Currently, these approaches are being integrated into OmpSs [2], [5], supporting accelerators and cluster environments. We also used it to implement a first approach for a transactional memory OpenMP [9]. The compiler has also been used to generate various kinds of code instrumentation, for later analysis with Paraver [1].

We have also used this infrastructure in a variety of projects: Acotes; SARC; Encore; TEXT; TERAFLUX; and in Mareincognito, the collaboration between BSC and IBM.

II. Structure of the Compiler

Figure 1 shows the structure of the Mercurium compilation process. The compiler takes C/C++ or Fortran 95 input files, and processes them through the frontend, getting an initial high level representation of the program units. Then, a set of transformation phases modify such representation, introducing the transformations guided by directives. As a possible consecuence of the transformations, part of the code can be outlined to separate files, that are reintroduced in the compilation flow (see the Secondary input source files box). Primary and Secondary input files can be later processed by different backends, thus supporting heterogeneous architectures. This has been used to generate code for the Cell/B.E., GPUs, and FPGAs.

III. Design decisions

This section presents the design decisions taken for the development of the Mercurium S2S compiler. We think that those principles have sense for a compiler devoted to research. In our research environment, we are interested in the quick prototyping of extensions, and the possibility of make them stable in the compiler

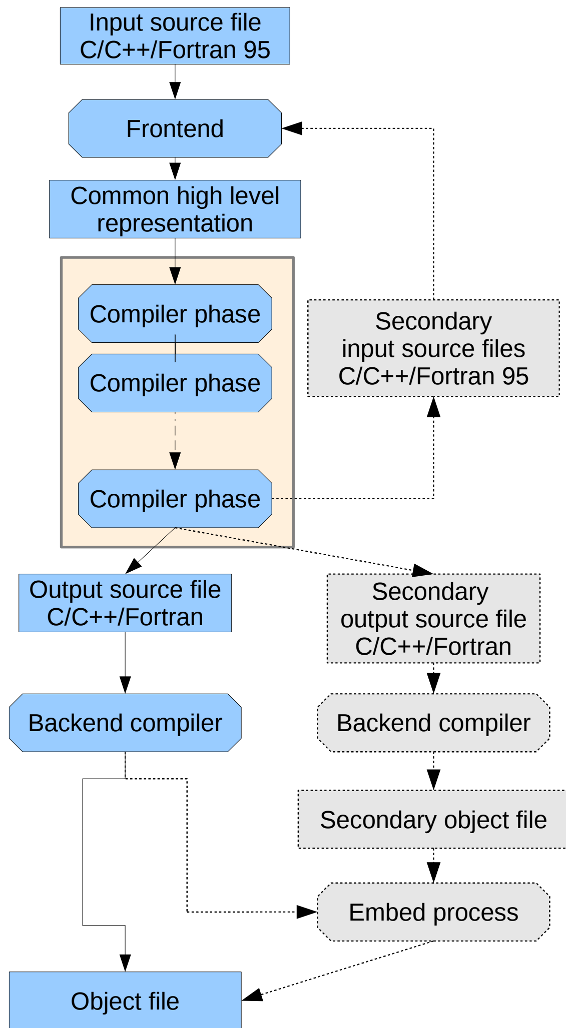

Fig. 1. Structure and phases of the Mercurium compilation process

later, when they show a benefit. The following design decisions are not targeted towards the development of a commercial compiler, as in this case there are additional requirements on speed and accurate diagnostic reporting, that may lead to different design decisions.

A. *Extended parsing & later disambiguation*

Researchers writing program transformations in a compiler need an extra level of extensibility of the compiler parser, and on the initial semantic analysis passes. Extensibility is good to be able to introduce certain language features, like new types, built-in functions, etc., and to fully extend the common pragmas/directives used to express the proposed new program transformations.

The Mercurium compiler accepts a relaxed form of pragma/directive syntax, where the initial parsing only recognizes tokens (clauses, expression lists associated to the clauses, values, etc.) which may not have any actual meaning. And then, a compiler pass implemented later gives meaning to the directive and clauses, that are then checked for correctness, and passed on, to be applied in a subsequent program transformation.

B. *Write source & subparsing*

Researchers implementing a transformation pass are usually familiar with the native target language (C/C++, Fortran), but not very familiar with the compiler internals.

One aspect of the ease of use of the Mercurium compiler is that most of the time, a program transformation can be expressed by means of the target language. This means that it is useful to allow the compiler writer to use literal sentences expressed in the same language. The usual alternative is to code the program transformation on the internal compiler representation (IR), a method that is usually tedious and error-prone.

For example, consider building the simple expression:

```
1 nth_num_proc = nth_cpus();
```

Working with a traditional compiler IR API, the programmer will need to add the following sentences to the compiler:

```
1 ...
2 cpus_statement = ...
3 expr_num_proc = build_0ary_op(VARIABLE, "nth_num_proc");
4 expr_cpus = build_0ary_op(VARIABLE, "nth_cpus");
5 expr_call_cpus = build_Nary_op(FUNCTION,
6                                expr_cpus, NULL);
7 expr_assign = build_binary_op(BIASSIGNMENT,
8                               expr_call_cpus,
9                               expr_num_proc);
10 cpus_statement = cat_stmt_ls(cpus_statement,
11                        build_expr_statement(expr_assign));
12 ...
```

Mercurium supports a specific type to easy the work of the researcher, *Source*, which allows to add such a sentence, just using the native language. Continuing with the example, see line 4 in the next listing:

```
1 ...
2 // current_ast, current_scope_link are the IR for the
3 // part of the program that needs to be transformed.
4 Source stmt << "nth_num_proc␣=␣nth_cpus();"
5 AST_t cpus_statement = stmt.parse_statement (
6                  current_ast, current_scope_link);
7 ...
```

With the introduction of the *Source* datatype in the compiler, Mercurium allows users to express the transformation pass with the source code that will be produced by it. At any point during the transformation, the compiler writer can mix *Source* code with the internal compiler information.

When the *Source* code has been generated, it is provided back to the compiler, to be parsed, and the internal representation is generated again, for processing on further transformation passes. We call this feature *Subparsing*, as it uses a portion of the grammar to recognize a subset of the full grammar of the language. For example, subparsing is able to understand a single declaration, an expression, or a collection of statements. Those will be collected and inserted at the point indicated in the IR of the compiler. As an example, see lines 5–6 in the previous listing.

C. Generic Driver & plug-ins

During a compilation, the compiler is driven by a configuration file. The compiler is invoked using a generic driver, which automatically knows which configuration file to use, given its invocation command name. The configuration file used describes the way that particular invocation of the compiler must proceed. For example, for the OpenMP transformations, there is a rule in the configuration file, indicating the set of compiler passes that need to be loaded and executed in order. A pass can be conditionally activated given a compiler command line option. This is the case of the instrumentation passes, that are activated only if the compiler is instructed to do so with the corresponding compiler flag. The following listing shows a snippet of this feature:

```
...
# if --instrument is given, activate the internal
#                  compiler variable indicating so
{instrument} options = --variable=instr:1

# load the proper compiler plug-in
{instrument} compiler_phase = libtlinstr.so

# and link against the proper (instrumented) libraries
{instrument} linker_options = \
           -L@NANOX_LIBS@/instrumentation -lnanox
...
```

The compiler frontend and initial semantic passes offer the basic IR to the rest of the passes. Each transformation phase can enrich the IR with new information, that can be later used by subsequent passes. This is very generic in our implementation. One pass can add new directives to the source level, that can be processed later in another pass, thanks to the subparsing feature explained before. Each compiler pass is implemented as a shared library, and it is loaded into the compiler process as a plug-in.

D. Drop-in replacement

When users of the compiler want to compile an application, usually they are faced with the challenge of adapting the application *Makefiles* (or autoconf, or CMake)-files, in order to invoke Mercurium in the proper way. To avoid such a need, we try hard to have Mercurium offering the flavor of just any other compiler, and specifically, to be compatible with the way of invoking GCC.

This means that Mercurium acts as a real compiler. It can generate object files, from source files; link object files to get a binary file; or do both at a time. But it will not need or leave other files than those known to Make et al. Mercurium also accepts the most common compiler flags used by GCC unmodified, so that there is no need to change those for the Make tools.

Being a drop-in replacement has been a challenge for the support of heterogenous environments, where portions of the code need to get outlined to separate files for later compilation with the specific SPE compiler, or GPU compiler. This is the topic of the following section.

E. Have multi-file support

Since 2006, with the raise of heterogeneous computing, our Mercurium infrastructure has been facing the challenge to support different compilation backends, while compiling and generating code from a single compilation unit. Initially, the effort was dedicated to support the Cell/B.E. processor [8], [7], [10]. Currently, we are using the infrastructure to support code generation for GPUs.

Multi-file support means that, as part of our compilation with Mercurium, some portions of the source code are identified to belong to an *accelerator* architecture. Then they are outlined, and written into separate files. Those files get automatically incorporated in the compilation process. As they belong to a different target, they are listed for later compilation following a different configuration file. Under such new configuration, Mercurium will compile them using the native *accelerator* compiler (the NVidia CUDA compiler, for example). Then, the resulting files are incorporated into the compilation flow in a way that mimics the usual compilation process.

F. Common representation for languages

In the area of High Performance Computing, applications written in Fortran are not unusual. So supporting both C/C++ and Fortran is highly desirable. Even if the two languages are very different it is possible to work on a superset that includes both languages. This approach is common in many compilers but it usually implies that the common representation is of lower level, which may suit a commercial compiler, but it is not desirable in a S2S compiler.

Mercurium shares a common representation for the symbolic information, typesystem and the abstract syntax tree. The abstract syntax tree is a high level representation of the program, with control structures retained, but without redundant information like declarations.

By using this approach, it is possible to write analysis and transformation passes valid for both languages so the amount of language-specific cases are minimized. When implementing a transformation, as long as it does not involve specific constructs of one of the languages, it could be written in any of the three languages.

IV. Conclusions

In this paper, we have shown the most relevant design decisions taken for the development of the Mercurium compiler. Mercurium is a source-to-source compilation infrastructure that has the goal to be flexible and extensible to accomodate new compiler transformations based on language directives and pragmas.

Mercurium has been used succesfully in a number of projects (Acotes, SARC), and it is currently being used in several others (Encore, TEXT, TERAFLUX), showing that the infrastructure is useful for research in program transformations.

Acknowledgments

We thankfully acknowledge the support of the European Commission through the ENCORE project (FP7-248647), the TERAFLUX project (FP7-249013), the TEXT project (FP7-261580), and the HiPEAC-2 Network of Excellence (FP7/ICT 217068), the support of the Spanish Ministry of Education (TIN2007-60625 and CSD2007-00050) and the Generalitat de Catalunya (2009-SGR-980 and 2009-SGR-1372), and the BSC-IBM MareIncognito project.

References

[1] http://www.bsc.es/paraver.

[2] Optimizing the Exploitation of Multicore Processors and GPUs with OpenMP and OpenCL. In *Proceedings of the 23rd International Workshop on Languages and Compilers for Parallel Computing (LCPC2010)*, October 2010.

[3] R.M. Badia, J.R. Herrero, J. Labarta, J.M. Pérez, E.S. Quintana-Ortí, and G. Quintana-Ortí. Parallelizing Dense and Banded Linear Algebra Libraries Using SMPSs. *Concurrency and Computation: Practice and Experience*, 21, August 2009.

[4] J. Balart, A. Duran, M. Gonzalez, X. Martorell, E. Ayguade, and J. Labarta. Nanos Mercurium: a Research Compiler for OpenMP. In *Sixth European Workshop on OpenMP*, pages 103–109, 2004.

[5] J. Bueno, L. Martinell, A. Duran, M. Farreras, X. Martorell, R.M. Badia, E. Ayguade, and J. Labarta. Productive Cluster Programming with OmpSs, August 2011. to appear.

[6] A. Duran, R. Ferrer, J.J. Costa, M. Gonzàlez, X. Martorell, E. Ayguadé, and J. Labarta. A Proposal for Error Handling in OpenMP. *Intl. Journal of Parallel Programming*, 35(4):393–416, August 2007.

[7] Alexandre E. Eichenberger, Kevin O'Brien, Kathryn M. O'Brien, Peng Wu, Tong Chen, Peter H. Oden, Daniel A. Prener, Janice C. Shepherd, Byoungro So, Zehra Sura, Amy Wang, Tao Zhang, Peng Zhao, Michael Gschwind, Roch Archambault, Yaoqing Gao, and Roland Koo. Using advanced compiler technology to exploit the performance of the Cell Broadband Engine(tm) architecture. *IBM Systems Journal*, 45(1):59–84, 2006.

[8] IBM. Cell Broadband Engine Resource Center, 2008. http://www.ibm.com/developerworks/power/cell/downloads.html.

[9] M. Milovanovic, R. Ferrer, V. Gajinov, O.S. Unsal, A. Cristal, E. Ayguade, and M. Valero. Nebelung: Execution Environment for Transactional OpenMP. *International Journal of Parallel Programming*, 36(3):326–346, 2008.

[10] Kevin O'Brien, Kathryn M. O'Brien, Zehra Sura, Tong Chen, and Tao Zhang. Supporting openmp on cell. *International Journal of Parallel Programming*, 36(3):289–311, 2008.

[11] Josep M. Perez, Pieter Bellens, Rosa M. Badia, and Jesus Labarta. CellSs: Making it easier to program the Cell Broadband Engine processor. *IBM Journal of Research and Development*, 51(5):593–604, September 2007.

[12] C.D. Polychronopoulos, M.B. Girkar, M.R. Haghighat, C.L Lee, B.P. Leung, and D.A. Schouten. The structure of Parafrase-2: an advanced parallelizing compiler for C and FORTRAN, 1990.

[13] E. Ayguadé V. Marjanovic, J. Labarta and M. Valero. Effective communication and computation overlap with hybrid MPI/SMPSs. In *Proceedings of the 15th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 337–338, New York, NY, USA, 2010. ACM.