ARCC: A Non-Intrusive Auto-Tuner

Benoît Meister, Benoît Pradelle, Muthu Baskaran, Richard Lethin
Reservoir Labs

May 18, 2017

Abstract

This report describes ARCC, an auto-tuner motivated by practical compiler auto-tuning requirements. ARCC is non-intrusive, in that it enables the auto-tuning of any suitable tool used in the build process without modifying any of the build structure itself. ARCC defines a protocol and a syntax. Any tool can become auto-tunable by implementing the ARCC client protocol. A distinctive advantage of the ARCC protocol is that it lets the client, in which most of the problem knowledge is available, define the search space. We validated the approach by implementing the ARCC protocol within the R-Stream parallelization tool. While ARCC was written with compilers in mind, it is usable with any software that requires auto-tuning.

1 Motivation

The ARCC auto-tuning framework presented here was originally written for the R-Stream source-to-source compiler [7] (referred to as "RCC" in this document). Auto-tuning addresses a few major limitations encountered with static program optimization and parallelization tools.

First, there is an intrinsic limitation in the predictive power of cost models based on static analysis, because they must reason with any possible runtime scenario while only knowing information from the source code.

However, in the case of compilers supporting the polyhedral model [7, 8, 2, **?**], the runtime behavior of a class of programs (called "static control programs") can be predicted exactly. Sophisticated cost models can be built to predict the main program performance drivers (such as the number of instructions to be executed or the nmber of cache misses during the program execution [4, 10, 6]). Still, practical implementations of such cost models need to trade off prediction accuracy for faster compilation time. Auto-tuning can make up for the lost accuracy.

Finally, source-to-source parallelizing compilers rely on a back-end compiler to optimize the utilization of individual cores, by providing specialized instruction selection, register allocation and instruction-level parallelism. While this structure is modular and increases the ability to target new types of processors

quickly, one of its drawbacks is that the back-end behavior may strongly depend upon some relatively unpredictable factors.

Hence, tools for searching in some space of mapping solutions are desirable in the context of parallelizing compilers.

However, we also designed ARCC to be used outside of the domain of compilers, because building programs often involve a variety of tools (domain specific language front-ends, code generators, text processors, etc.), which may benefit from auto-tuning as well. Without any particular domain-specific knowledge, auto-tuning is subject to the curse of dimensionality: the number of combinations of parameters – and hence the duration of a likely-good search – increases exponentially as a function of the number of parameters in the search. ARCC addresses this problem by letting the auto-tuned software (which is called the "client" in this document) to define the search space. The client is often the entity that has the most knowledge about the domain-specific problem it has to solve. Following this assumption, the ARCC protocol lets the client define the set of parameters to be searched, as well as the values to be searched (i.e., the search space) as a function of its domain-specific knowledge of the problem.

In this paper, we present the ARCC tool and protocol using the RCC automatic parallelization compiler as an example client.

The approach we are choosing with ARCC is to tune advanced compiler options. Components of the RCC parallelization engine are named "tactics." As clients of ARCC, each tactic defines the space in which it should be searched. This article shows the main concepts of ARCC, and it can also be used as a user's manual. Note that the RCC user manual contains a section on ARCC usage as well.

2 Basic Operation

ARCC does not typically call specific clients directly. Instead, it communicates with the clients that happen to be active during the auto-tuning process. ARCC is called using the following command:

```
$ arcc --build=<cmd> --run=<cmd> --clean=<cmd> [options]
```

To use ARCC, users simply need to define:

- how to build the target program (e.g., "make foo")
- how to run the target program (e.g., "./foo input.dat")
- how to clean up the generated files (e.g., "make clean")

and to give this information to ARCC as command-line arguments.

More generally, the `build` option specifies how to construct a solution that can be evaluated, while the `run` option specifies how to evaluate the solution.

Auto-tuning with ARCC is done in two steps. In the first step (called *production*), meta-data is produced by the client during the first build. The produced meta-data defines the client's auto-tuning search space.

In the second step (called *consumption*), ARCC iteratively instantiates solutions in the search space defined by the meta-data and evaluates the solutions.

At each iteration, ARCC

- sets up options for the various RCC tactic options,
- (if needed) cleans previously generated files,
- builds the target program,
- runs it, and
- measures its performance.

Each instance is recorded along with the subsequent execution time. After a number of iterations, the program instance that minimizes execution time is built and ARCC returns.

A major advantage of ARCC communicating directly with the RCC tactics is that users don't need to modify their Makefile.

Note that a `clean` option is necessary when the build system will not rebuild the program if it is seen up-to-date, such as `make`. Otherwise, if the build system just overwrites the existing program, the clean option is not needed (i.e., assign its value with an empty string), as exemplified by the following command:

```
$ arcc --build="rcc --polymap --mm=core2duo --march=x86_64-linux
  --backend=\"icc\" --backendoptions=\"-O3 -openmp\" -o foo foo.c"
  --run="./foo input.dat" --clean=""
```

3 Options

ARCC options can be obtained by running `arcc --help`. They are listed below as of the time of publication.

```
\inputinput{arcc-usage.tex}
```

Notes:

1. The `--help` option asks ARCC to print a usage message on the screen.
2. To display what is happening during ARCC's auto-tuning process, use the `--verbose` option.
3. User can run the steps of producing the meta-data and consuming it separately. To run the former only, use `--produce`. User can customize the auto-tuning search space by modifying the produced meta-data file. To run the latter, use `--consume`.
4. For debugging purposes, the `--keep` option can be used to save all files generated or modified after each build in a subdirectory `arcc-codes`.

5. Advanced user can selectively decide a set of RCC's tactic options used in the auto-tuning. To view a list of all RCC's auto-tunable tactic options, use `--list` option. To simply tune *all* RCC's auto-tunable tactic options, use `--tune-all` option. See Section 6.3.

6. There is a default meta-data file name, which you can optionally override with the `--meta-data` option.

7. Similarly for log files. See Section 4.

8. Because the size of the search space can be huge, a brute force search space exploration technique (`--exhaustive` option) is not always feasible. Hence, heuristic algorithms (`--random` and `--simplex` options) are provided for reducing the search space size and thus the auto-tuning time. The search algorithm is set by default to the *Nelder Mead Simplex* method (the `--simplex` option), which is a popular non-derivative direct search method for optimization. The `--random-seed` option can be used to control the randomness of the results of the heuristic algorithms.

9. The `--max-trials` option can be used to limit the maximum number of instances in the search space empirically evaluated by ARCC.

10. The `--precise-perf` option is specific to RCC. It asks RCC to generate more exact timings information at the level of a mapped function, whereas `--rough-perf` just measures total execution time.

4 Logging and Reporting

ARCC includes logging and reporting mechanisms useful for gathering diagnostic information to track down the tuning results and failures. Logging is made in three different files:

- `arcc-main.log`: the *main* log file that records all steps of the auto-tuning process
- `arcc-result.log`: the log file that records all best results found at each search-space exploration step
- `arcc-error.log`: the log file that records all encountered failures

Note that the contents of `arcc-result.log` and `arcc-error.log` are also recorded in the main log file (`arcc-main.log`). The purpose of creating additional log files for reporting results and errors is to localize log messages, so that the user need not go through the long content of the main log file. These log file names can be prefixed by the user using the `--log` option.

Moreover, ARCC also offers the RCC-specific `--keep` option to store all files that are generated or modified after each build performed during the consumption mode. This feature is useful for keeping track of RCC-generated files such

as the mapped codes, the generated executables, the mapper log files, etc. All of the stored files are saved by default in a subdirectory, `arcc-codes`. An example of how the subdirectory structure of `arcc-codes` looks like after tuning a matrix multiplication code can be observed in the following.

```
% tree arcc-codes
arcc-codes
‘-- arcc-run-2011-3-30-16h-34m-47s
    |-- coord-1-2-0
    |   |-- matmult
    |   |-- matmult.gen.c
    |   ‘-- README
    |-- coord-0-2-0
    |   |-- matmult
    |   |-- matmult.gen.c
    |   ‘-- README
    |-- coord-0-3-0
    |   |-- matmult
    |   |-- matmult.gen.c
    |   ‘-- README
    |-- coord-1-3-0
    |   |-- matmult
    |   |-- matmult.gen.c
    |   ‘-- README
    ‘-- coord-2-0-1
        |-- matmult
        |-- matmult.gen.c
        ‘-- README

6 directories, 15 files
```

In the above example, the auto-tuning built and empirically evaluated five different program instances. Each code instance is represented as a Cartesian coordinate (i.e., `coord-X-Y-Z`) in the tuning search space. Multiple ARCC runs are recorded separately in subdirectories `arcc-codes/arcc-run-<timestamp>`. The files `matmult` and `matmult.gen.c` are the executable and the mapped code generated by RCC, respectively. Each of the generated `README` file consists of supplementary information such as the used build/clean/run commands, the program version (represented as a Cartesian coordinate in the search space), the used tactic options, etc.

5 The ARCC protocol

In the general use case for ARCC, a set of auto-tunable entities define their search space in production phase, and the search space is instantiated for the entities during the consumption phase.

In order to support a general set of auto-tunable entities, all the details of the search space are defined by the auto-tuned entities. Each entity must declare an identifier, a syntax for passing solutions from the search space, and a definition of the search space it wants to be auto-tuned for. The search space itself is defined as a set of named variables, as well as the set of values that each variable is allowed to have in the search. The search space can be refined using constraints among the search variables.

The grammar used for declaring the search space of an auto-tunable entity is as follows:

```
<ID> "{"
  "option" "=" <template-string> ";"
  ( "var" <var-name> "=" "[" <var-val> ( "," <var-val> )* "]" ";" )*
  [ "constraint" "=" <constraint-exp> ";" ]
"}"
```

Let us look at each section of the syntax, illustrated with the following example in the case of RCC. The meta-data for RCC's unroll-and-jam tactic that runs on function "matmult" when targeting an SMP target architecture will look like:

```
UJ_SMP_MATMULT {
  option = "max_unroll=$A$,max_unroll_per_dimension=$B$,innermost=$C$";
  var A = ["1","8","64","256"];
  var B = ["1","4","8","16"];
  var C = ["true","false"];
  constraint = (A>=B);
}
```

Each auto-tunable entity needs to identify itself with ARCC using an `ID`. `ID` must be unique within a search. In this example, we assumed that no two functions to be compiled would have conflicting names (a reasonable assumption in C). Several optimizations may require a search for each function of each compiled program. RCC supports mapping codes to hierarchies of parallel computing, and a particular optimization (which we call a "tactic") may be used at any level of the hierarchy. Hence, a unique name is formed for each (tactic, level, function) triplet by concatenating their names (joined by the _ character). `UJ` identifies the unroll-and-jam optimization, `SMP` a Symmetric Multi-Processing architecture, and `MATMULT` the name of the function being optimized.

The search space for our unroll-and-jam is defined by three variables:

- the total maximum unroll factor, represented by variable A;
- a per-loop maximum unroll factor, represented by variable B;
- a flag that toggles the unrolling of the innermost loop, represented by variable C

The `var` fields define the variable names as well as the values they are allowed to take in the search.

The search space of an auto-tunable entity can be additionally pruned using the `constraint` field. The goal of enforcing constraints on variables' values is to enable further pruning of the search space. As a concrete example, the number of unrolling factors must satisfy the machine's register capacity to avoid register spills.

It can also be convenient for ARCC to present particular auto-tuning solutions using a syntax that is specific to the auto-tunable entity. This is the role of the `option` field. In RCC, the main parameters for each tactic can be specified at the command line by the user. In order to make it easier to transpose a

good solution found by ARCC into a set of command-line options, auto-tunable tactics use the `option` field to specify the way a command-line option would be passed. Naturally, the tactic option parsers are used to read the tactic options from the `option` string made available by ARCC in the consumption phase.

In terms of syntax, `<template-string>` is a string containing to-be-tuned variables enclosed by $ signs, and `<constraint-exp>` is a set of constraint expressions defined in the Python scripting language.

Users can also specify a *global* constraint across tactics' meta-data. The variables used in a global constraint must be mangled with the tactic ID name. A use example of a global constraint is given below:

```
ID_1 {
  option = "opt1=$A$,opt2=$B$";
  var A = ["1","2","3","4","5"];
  var B = ["2","4"];
}
ID_2 {
  option = "opt1=$A$;
  var A = ["1","2","3","4"];
}
constraint = (ID_1_A==ID_2_A) and (ID_1_B>=ID_2_A);
```

6 Advanced Usage

This section is intended for an advanced user who wants to have more precise control over ARCC's auto-tuning configuration and search space.

6.1 Running Separate Production and Consumption Modes

Advanced users may run the production and consumption steps separately and modify the produced meta-data, which is concentrated in a single meta-data file (called `arcc-meta-data.md` by default). For example, the following command can be used to run ARCC in production mode:

```
$ arcc --build="make foo" --clean="make clean" --produce
```

Note that a `run` option is not needed in production mode.

After that, users can optionally alter the auto-tuning search space by modifying the meta-data file. Details on the syntax of the meta-data file are given in Section 6.2. With the search space defined in the obtained meta-data file, users can start the search for the best-performing program instance by running ARCC in consumption mode, using the command below:

```
$ arcc --build="make foo" --run="./foo input.dat"
  --clean="make clean" --consume
```

Obviously, the `build` and `run` options are required in consumption mode.

6.2 Meta-Data Syntax

The meta-data file consists of a sequence of tactic-specific meta-data. Each tactic provides:

- A syntactic template for the options it will accept to be auto-tuned for. The template contains variables, enclosed by $ signs, which will be instantiated by ARCC's search.
- The set of values allowed for each variable in the search is also provided.
- Finally, constraints on the variable's values can be formulated.

6.3 Selecting Auto-Tunable Tactic Options

This section presents the `--list` option, an ARCC feature that is currently specific to RCC.

A subset of the auto-tunable tactic options of RCC are auto-tunable *by default*. Others can be made auto-tunable through command-line options. Both the dimension and the size of the search space depend on the total number and the value ranges of the tactic option parameters, respectively. So, it would be beneficial for the user to explicitly select a subset of RCC's auto-tunable tactic options, to limit the search to a manageable size and therefore to reduce the empirical tuning time. Note that there is always a trade-off between tuning speed and quality of the solution – in the case of RCC, the tuned program performance. Users should be aware of possible performance changes when selecting auto-tunable options of RCC's tactics.

To view the list of all available auto-tunable tactic options in RCC, user can use the `--list` option. The following text shows the output format of RCC's auto-tunable tactic options list.

```
% arcc --list
[arcc] List of all autotunable tactic options:
-------------------------------------------------------------------------
| No. |  Tactic  |  Option  | Used by | Description
|     |          |          | default |
-------------------------------------------------------------------------
|  1. |  tactic1 |  option1 |       y | Description on tactic1-option1
|  2. |  tactic1 |  option2 |       n | Description on tactic1-option2
|  3. |  tactic1 |  option3 |       y | Description on tactic1-option3
|  4. |  tactic2 |  option4 |       y | Description on tactic2-option4
|  5. |  tactic2 |  option5 |       n | Description on tactic2-option5
|  6. |  tactic3 |  option6 |       y | Description on tactic3-option6
-------------------------------------------------------------------------
```

The selection of RCC's auto-tunable tactic options is done within RCC, to maintain a clear separation of knowledge and responsibility between ARCC and RCC. A list of used tactic options can be specified by user in the makefile and is given to RCC as command line arguments, as exemplified in the following.

```
% rcc --map:tactic1=autotune={option1,option2},tactic2=autotune=all input.c
```

The above example indicates that the options auto-tuned by ARCC are the first two options of `tactic1` and *all* options of `tactic2`. No options of `tactic3` will be auto-tuned.

Users are not required to specify any auto-tunable tactic options. In this case, ARCC will detect that no meta-data file is generated after the first build during the production mode, and then ARCC will rebuild and decide to use RCC's *default* list of auto-tunable tactic options to generate the meta-data, which later will be consumed by ARCC during the consumption mode. Optionally, user can also use the RCC option `--map:<tactic>=autotune=default` to explicitly select the default auto-tunable options for a particular tactic. As shown previously in Section 3, the `--tune-all` option of ARCC is essentially equivalent to applying `--map:<tactic>=autotune=all` to all RCC tactics.

7 Implementation notes

ARCC has been written and shipped closed-source as part of the R-Stream automatic parallelizer since 2011. It is now available as open source on github. The released source code contains the ARCC auto-tuner, written in Python (2.7), and a Java class that implements the parts of the ARCC protocol related to:

- Detecting whether ARCC is running auto-tuning;
- whether we are in the production or consumption phases; and
- communicating meta-data to and from ARCC.

8 Related work

Other efforts related to auto-tuning of compilation and automatic parallelization tools have been implemented. In the polyhedral optimization world, LetSee [9] focuses on auto-tuning polyhedral scheduling, an optimization that reorders iterations in loops using multi-dimensional linear functions. The principles explored in this tool (using the mathematical formulation of the search space, restricting the amplitude of schedule coefficients, etc.) are applicable to RCC's auto-tunable affine scheduling tactic.

In the context of autotuning program transformations, a few other systems exist. CHiLL and X-Tune [1] auto-tune the construction of program transformation scripts, producing a high-level description of how to parallelize and optimize loops. The approach in Orio [5] is different on that it is driven by annotations to input code (or pseudo-code) rather than a separate script.

In the field of tunable compilers, Milepost GCC [3] is an ambitious *collective tuning* project based around the GNU Compiler Collection. A key idea in Milepost GCC is that all compilations can contribute to a global, persistent database, and machine-learned from. Compilations using Milepost GCC can

then benefit from all the previous runs of Milepost GCC on any program, with any compiler flags and options.

9 Conclusion and Future Work

While we have found ARCC useful in the context of RCC, we have identified a few limitations to its current implementation, even in the particular context of compiler optimization:

- The ARCC process is currently sequential. A parallel version would enable ARCC to search larger spaces. Single-node as well as multi-node parallelization would be desirable.
- Currently available search algorithms are non-differential, i.e., they are not using evaluations of search points to direct the next solutions to be searched.
- The decisions of some RCC tactics have an influence on the optimal search space of subsequent tactics. This effect is well known in the compiler trade as the *phase ordering problem*. The current single-production auto-tuning cycle implemented by ARCC seems insufficient to take phase ordering into account.
- The performance of an auto-tuning search can be improved when clients have an independent impact upon the objective function. Let $S(A)$ and $S(B)$ the size of the search space for two ARCC clients A and B. Instead of searching a space of size $S(A) \times S(B)$, if the impacts of A and B on the objective function are independent, the search can be performed in time $O(S(A)) + O(S(B))$, which is much faster than in $O(S(A) \times S(B))$.

10 Acknowledgements

We would like to thank Albert Hartono (now with Intel), who wrote the reference implementation of ARCC (now available on github) back in 2010.

References

[1] Protonu Basu, Samuel Williams, Brian Van Straalen, Mary Hall, and Phillip Colella Leonid Oliker. Compiler-directed transformation for higher-order stencils. In *International Parallel and Distributed Processing Symposium (IPDPS)*, May 2015.

[2] U. Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Affine transformation for communication minimal parallelization and locality optimization of arbitrarily nested loop sequences. Technical Report OSU-CISRC-5/07-TR43, The Ohio State University, May 2007.

[3] Grigori Fursin, Cupertino Miranda, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Ayal Zaks, Bilha Mendelson, Edwin Bonilla, John Thomson, Hugh Leather, et al. Milepost gcc: machine learning based research compiler. In *GCC Summit*, 2008.

[4] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Cache miss equations: an analytical representation of cache misses. In *ICS '97: Proceedings of the 11th international conference on Supercomputing*, pages 317–324, New York, NY, USA, 1997. ACM.

[5] Albert Hartono, Boyana Norris, and Ponnuswamy Sadayappan. Annotation-based empirical performance tuning using orio. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–11. IEEE, 2009.

[6] B. Meister and S. Verdoolaege. Polynomial approximations in the polytope model: Bringing the power of quasi-polynomials to the masses. In *ODES-6: 6th Workshop on Optimizations for DSP and Embedded Systems*, April 2008.

[7] Benoît Meister, Nicolas Vasilache, David Wohlford, Muthu Manikandan Baskaran, Allen Leung, and Richard Lethin. R-Stream compiler. In David A. Padua, editor, *Encyclopedia of Parallel Computing*, pages 1756–1765. Springer, 2011.

[8] S. Pop, A. Cohen, C. Bastoul, S. Girbal, P. Jouvelot, G.-A. Silber, and N. Vasilache. GRAPHITE: Loop optimizations based on the polyhedral model for GCC. In *Proc. of the 4th GCC Developper's Summit*, pages 179–198, Ottawa, Canada, June 2006.

[9] L.-N. Pouchet, C. Bastoul, A. Cohen, and N. Vasilache. Iterative optimization in the polyhedral model: Part I, one-dimensional time. In *IEEE/ACM Fifth International Symposium on Code Generation and Optimization (CGO'07)*, pages 144–156, San Jose, California, March 2007. IEEE Computer Society press.

[10] S. Verdoolaege, R. Seghir, K. Beyls, V. Loechner, and Maurice Bruynooghe. Analytical computation of Ehrhart polynomials: enabling more compiler analyses and optimizations. In *Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 248–258. ACM Press, 2004.