Reservoir Labs

R-Stream Parallelizing C Compiler
Getting Started Guide

For Software Version 3.15.0
Preface

Contacting Reservoir Labs
Call: 212.780.0527
Fax: 212.780.0542
Email: rstream-support@reservoir.com
Web-based inquiry form:
https://www.reservoir.com/support/r-stream-support/
To report a bug:
https://www.reservoir.com/cgi-bin/bugzilla3/

Copyrights
Copyright © 2009-2015 Reservoir Labs, Inc. All rights reserved.

Disclaimer
The content of this document is provided for informational use only, is subject to change without notice, and should not be construed as a commitment by Reservoir Labs, Inc.

Contents

Preface ................................................................. ii
   Contacting Reservoir Labs ............................... ii
   Copyrights ..................................................... ii
   Disclaimer ..................................................... ii

1 Introducing R-Stream ................................................... 1
   Typical Usage .................................................. 1
   System Software Requirements ........................... 2

2 Installing R-Stream .................................................. 3
   Downloading the R-StreamBinary Installer .......... 3
   Downloading the R-StreamTarball ..................... 3
   Setting up the User Environment ...................... 4
   Activating the R-StreamLicense Key File ............ 5
   Configuring RCC ............................................. 5
   Validating the RCC Installation ....................... 5
   Updating the Machine Model Description .......... 5
      Updating OpenMP Machine Model Files ............ 6
      Updating Non-OpenMP Machine Model Files ...... 7

3 Compiling for Parallel Execution ................................. 8

4 Programming Guidelines ........................................... 10
   Affine Parametric Loop Bounds ....................... 10
   Constant Loop Counter Increments .................. 11
   Affine Array Accesses ................................... 12
   Conditionals ............................................... 12
   Non-Analyzable Constructs ............................ 12
5 R-Stream Coding Best Practices
  Memory Allocation ........................................ 14
  Multidimensional Arrays ................................. 15
  Pointer Swapping ......................................... 16
  Short-circuited conditionals ............................ 17
  Library Calls ............................................. 18
  Unmappable Code ......................................... 19

6 Auto-Tuning Using ARCC ................................. 21

7 Code Examples ............................................ 22
  Where to Find the Examples .............................. 22
  Directory Structure ...................................... 22
  Running the Examples .................................... 23
  Further Documentation ................................... 23

8 For More Reading ......................................... 24
1 Introducing R-Stream

The R-Stream Parallelizing C Compiler (hereafter referred to as RCC) is designed to address the needs of high-performance scientific, signal and image processing applications. RCC is a high-level, source-to-source C compiler that performs automatic parallelization using its internal mapper\(^1\) component. The mapper is a powerful loop analysis and transformation tool that is based on the polyhedral model of static control programs.

The mapper accepts C89 functions that comply to certain coding guidelines, defined in Sections 4 and 5 of this guide. Basically, these guidelines help you present the function you want to map in a form that makes advanced parallelizing compiler analysis feasible. The typical programs that benefit from the mapper’s optimizations are loop codes (for loops) that perform computations over dense arrays. A class of programs called “static control programs” can usually be mapped directly without changes to the code. This class of programs consists of:

- **for** loops, with loop bounds that are affine functions of outer loops and parameters (loop-invariant values) and whose loop counters are incremented by a constant.
- Array indexing functions that are affine functions of loop indices and parameters.
- Conditional (if/else) expressions, where the array indexing functions involved are affine functions of loop indices and parameters.

Other programs that do not only present such “regular” features need to be written in such a way that the “irregular” parts are abstracted away, as presented in Sections 4 and 5 of this guide, and in the “Programming guidelines” section of the *R-Stream Parallelizing C Compiler Power User Guide*.

**Typical Usage**

Users typically run RCC to map their C source code and then automatically compile the newly mapped source code using a standard low-level C compiler, such as gcc, icc, xlc, tile-cc (tile-gcc support is in development), or nvcc. Only those portions of C code that have been tagged for mapping and are mappable are transformed into parallelized code. The low-level, backend compiler optimizes the resulting individual tasks that run on single cores or accelerators.

Optionally, you can compile only the *compute kernel\(^2\)* portion of your application using RCC, then compile and link the resulting output files with your application’s other source files using a low-level compiler. This requires you to put the compute kernel parts of your application into separate files. This approach enables you to run RCC on only those parts of the program that you want to parallelize.

\(^1\)We refer to *mapping* as the process of transforming a sequential C program into efficient parallel code.

\(^2\)It refers to the portion of code in your High-Performance Technical Computing (HPTC) application that you want RCC to parallelize. This code must conform to the structure of an extended static control program.
System Software Requirements

RCC has been fully tested on these Linux versions: Red Hat Enterprise Linux 5.5 (RHEL5.5), Fedora 9, and Ubuntu 10.04. RCC requires the Java JDK version 1.6.x or 1.7.x. If it isn’t already installed on your system, download and install it before you download and install RCC.

- RHEL5.5 installs Java in /usr/java/latest
- Fedora installs Java in /usr/java/latest
- Ubuntu installs Java in /usr/lib/jvm/java-6-sun

For OpenJDK on Ubuntu:

- /usr/lib/jvm/java-6-openjdk
- /usr/lib/jvm/java-7-openjdk

Please note that by default, RCC is distributed as a 64-bit version. To make it run, you will need a 64-bit OS and Java.

Additionally, RCC has been fully tested for proper installation and partially tested for all functionality on OS X 10.8.x / Darwin 12.x.x. RCC on these systems are not yet fully supported, but have been tested with OpenJDK 6 and 7.
2 Installing R-Stream

R-Stream is distributed with a binary installer for ease of installation. The binary installer will guide the user through all steps necessary to properly install and configure R-Stream. Reservoir also provides a tarball image for manual installation. Instructions for both methods of installation are provided below.

Downloading the R-Stream Binary Installer

We’re providing a binary installer available on CD, or as a download from Reservoir Labs Product Support site. Download and extraction steps are given below.

1. Using the username and password provided to you by Reservoir Labs, login to the Reservoir Labs Product Support site: https://www.reservoir.com/support/r-stream-support/.

2. Select the R-Stream link.

3. Click the option that matches the licensing agreement you made with Reservoir Labs (e.g., R-StreamCommercial, R-StreamResearch).

4. Select the item you want to download (e.g., rstream_install.bin binary installer).

5. In the download dialog box, click Save File.

6. Change the permissions of the installer:

   $ chmod +x rstream_install.bin

7. Run the installer:

   $ bash ./rstream_install.bin

8. If the installer is run as a local user it will ask to install in the working directory. If the installer is run as root, it will ask to install in /opt/reservoir.

9. After completing, the installer will provide user instructions for exactly which environment variables need to be set, and it will make a suggestion on how to set the path.

   After completing the installation you can skip to Activating the R-Stream License Key File, but note that Configuring RCC is no longer necessary.

Downloading the R-Stream Tarball

RCC is packaged as a tarball available on CD, or as a download from Reservoir Labs Product Support site. Download and extraction steps are given below.

1. Using the username and password provided to you by Reservoir Labs, login to the Reservoir Labs Product Support site: https://www.reservoir.com/support/r-stream-support/.
2. Select the **R-Stream** link.

3. Click the option that matches the licensing agreement you made with Reservoir Labs (e.g., R-StreamCommercial, R-StreamResearch).

4. Select the item you want to download (e.g., R-Stream\text{x.x.x} Binary).

5. In the download dialog box, click **Save File**.

6. Extract the R-Stream\text{tarball}:

   \begin{verbatim}
   $ tar -xzvf rstream-bin-x.x.x-y-linux.tar.gz
   \end{verbatim}

   where \text{y} identifies the platform and \text{x.x.x} version of the released software. When extracted, the installation creates the \text{rstream-x.x.x} subdirectory and installs the \text{RCC} there.

### Setting up the User Environment

For the example’s sake, let us assume that you unpacked the tarball in a directory named \text{my\_dir} (this could be anywhere from $\text{HOME/work}$ to $\text{/tmp}$). Edit your \text{.bashrc} file to declare these environment variables:

- \begin{verbatim}
   $ export RSTREAM_HOME=my_dir/rstream-x.x.x
   \end{verbatim}

   Note that you can have multiple versions of \text{RCC} installed at the same time, but to use a particular version you must set the \text{RSTREAM_HOME} environment variable to point to it.

- \begin{verbatim}
   $ export JAVA_HOME=/usr/java/latest
   \end{verbatim}

   Set \text{JAVA_HOME} to the location where the Sun/Oracle Java JDK is installed on your system. For example, on RHEL5.5 systems, it is normally installed in $\text{/usr/java/latest}$. A list of tested java versions is provided in Section 1. Please note that Oracle/Sun JDK java is required. The GNU java tools are not sufficient.

   Note that if you have the 64-bit release of \text{RCC}, make sure to use 64-bit Java. Ditto for 32-bit. Otherwise, a linkage error will be encountered when running \text{RCC}.

- Include the following to make \text{RCC} (and Java) accessible from everywhere.

  \begin{verbatim}
  $ export PATH=$JAVA_HOME/bin:$RSTREAM_HOME/rstream/bin:$PATH
  \end{verbatim}

  An important point here is that $\text{JAVA_HOME/bin}$ appears before $\text{/usr/bin}$ in the path, so that Oracle/Sun’s java is used.
Activating the R-Stream License Key File

If you have purchased a user-specific or node-specific license for RCC, you need to run the licensing information tool:

$ my_dir/rstream-x.x.x/rstream/bin/license_tool

This will create a file rstream_proto_license in the current directory. You should email this file to rstream-support@reservoir.com in order to obtain your R-Stream license file.

If you have already obtained a license file from Reservoir, you do not need to perform the above step. You can place the license file in my_dir/rstream-x.x.x/rstream/lib/rstream.rlm, or you can set the environment variable $RSTREAM_LIC to point to the the license file. Once that is done, RCC is ready to be configured and used.

Configuring RCC

If you installed R-Stream with the binary installer, please skip this step.

$ cd my_dir/rstream-x.x.x/rstream
$ ./configure

Validating the RCC Installation

To ensure that RCC was installed properly, you can run a matrix-multiply validation test:

$ cd my_dir/rstream-x.x.x/rstream/examples/openmp
$ make PROGRAMS=matmult_c
$ ./matmult_c

The make command should first invoke RCC in order to compile a serial matrix-multiply code into OpenMP code (matmult_c.gen.c). Then, it will call the system’s low-level compiler to create an executable (matmult_c). The final command will run the executable, and report the performance from multiplying two 1024x1024 single-precision matrices together. If any of these steps do not occur, there may be issues with the RCC installation.

Updating the Machine Model Description

It is important to provide RCC with an updated machine model description to match your target hardware architecture. R-Stream includes machine model files for various target architectures. The contents of a machine model file (<target>-mapper.xml) describe the main components of the corresponding target architecture to the RCC mapper. The mapper produces a mapping according to this description of the target architecture.

1The _c suffix denotes a constant version of the matrix-multiply code where the input problem sizes are compile-time constants. The opposite of a constant version is a parametric version that has parametric input problem sizes.
If the specified machine model file fails to describe the target architecture accurately, the resulting executable can be compromised in one of two ways:

- In most cases, the executable fails to perform as expected.
- In some cases, depending on the target architecture, the executable fails to execute correctly.

For example, the NVIDIA GPU processing elements have a limited amount of memory that is explicitly managed. If the machine model file describes a larger amount of local memory than the target architecture has, the parallelized program may overflow the target machine’s local memory, leading to incorrect results or causing the executable to crash.

The supplied machine model files are located in directory: 
```
my_dir/rstream-x.x.x/rstream/mm
```
Machine model files are supplied to support the following target architectures:

- **OpenMP Intel**: core2duo-mapper.xml.
- **Intel Vector Architecture**: xeon5520-mapper.xml.
- **NVIDIA GPU (run on x86 machine)**:
  - core2duo-cudal-gtx285-mapper.xml,
  - core2duo-cudal-gtx480-mapper.xml.
- **Tilera**: tile64-2D-mapper.xml,
  - tile64-2D-scratchpad-mapper.xml\(^2\).

AMD x86 machines are supported as well (by adapting one of the Intel machine models).

If the version of your target architecture is out-of-sync with the description in its supplied machine model file, you need to ensure that RCC uses a correct description of your machine architecture. How to update a machine model description is described next.

If your target is an OpenMP-based machine (x86 or PowerPC), you can use the following tool to extract information about the main components of your target machine architecture and use this information to modify the target’s machine model description. For example, to run the tool:

- **Locally on the target machine**:

  ```
  $ my_dir/rstream-x.x.x/rstream/bin/smp-config
  ```

- **Remotely on the target machine**
  
  First copy `smp-config` to the remote target machine, then:

\(^2\)Despite the `tile-64` name, these machine model files are not limited only to TILE64, but also work for TILEPro. Also, "scratchpad" here indicates the enabling of virtual scratchpad optimization.
Updating the Machine Model Description

$ ssh <username@remote_target>
my_dir/rstream-x.x.x/rstream/bin/smp-config

The output of smp-config looks like this:

'--setmm=proc.cpu.geometry:[8]'
'--setmm=mem.global.size:[12326584K]'
'--setmm=mem.L2.size:[8M]'
'--setmm=mem.L2.cache_line_size:64'
...

There are two ways to update an OpenMP machine model description:

1. Make a copy of the core2duo-mapper.xml file (name it to opteron2212-mapper.xml, for example) and save it in my_dir/rstream-x.x.x/rstream/mm. Update the new file with the xml properties that correspond to the smp-config output. Then select the new machine model using the --mm command line option of RCC:

   $ rcc --mm=opteron2212 ...

2. Supply the lines of smp-config output as arguments to RCC at compile time to dynamically override any out-of-sync entities in the supplied machine model description.

   $ rcc --mm=core2duo 'smp-config' ...

Note that, by using this command line technique, the used machine model file (core2duo-mapper.xml) remains unchanged.

If you are targeting a non-OpenMP machine model, you must gather the relevant information about the machine, and then update the existing machine model file that corresponds to the target architecture. The vendor’s datasheet for the target architecture typically includes this information.

More information about machine models for RCC is available in R-StreamParallelizing C Compiler Power User Guide. Additionally, The machine model description instructions and information have been expanded into a separate document, rstream-machine-model-x.x.x.pdf, available at https://www.reservoir.com/support/r-stream-support/.

---

3Creating a new file and then editing it are preferred as opposed to modifying core2duo-mapper.xml, since other machine model files may include core2duo-mapper.xml so side effects may be introduced in other existing machine models.
3 Compiling for Parallel Execution

RCC can be invoked in batch mode and interactive mode. Batch mode is the simplest way to invoke since it automatically proceeds from the original C source code to the executable in one step. This includes mapping the input C code, generating the new target code, and finally invoking the low-level compiler on the target code. For simplicity, this guide only covers how to run RCC in batch mode. Details on running RCC in interactive mode are discussed in R-Stream Parallelizing C Compiler Power User Guide.

Note that, before compiling to automate parallelization using RCC, make sure the functions in your source code that you want RCC to parallelize conform to the static control program structure (for details, see Section 4).

Follow the steps given below in order for RCC to map and parallelize the chosen functions in your C code (in batch mode):

1. In your source code, specify which functions are valid regions for mapping by tagging them with a #pragma rstream map directive. For example:

```c
#pragma rstream map
void f(float (*A), float (*B), int T, int N) {
    int i, t;
    for (t=0; t<T; t++) {
        for (i=1; i<N-1; i++) {
        }
        for (i=1; i<N-1; i++) {
        }
    }
}
```

In this example, T and N are loop-invariant values, which we call parameters. These parameters must be formal parameters of the function you want to map.

2. To enable mapping:

   - Use the `-polymap` flag to activate the RCC’s mapper component. If not used, RCC will only perform scalar optimizations on the given input code.
   - Use the `-mm=<mapper_file>` flag to specify to the mapper which machine model file to use for the target architecture. You can see the list of available mapper files in
   ```
   ~/work/rstream-x.x.x/rstream/mm
   ```
   - Use the `-march=<architecture>` flag to specify other platform properties that are not captured by the machine model description. This flag invokes the file:
   ```
   ~/work/rstream-x.x.x/rstream/mm/<arch>.pl
   ```
   Examples of valid values of `-march` include `x86_64-linux`, `ia64-linux`, `i686-linux`, `i686-cygwin`, and `tile64`. The listed directory contains a more complete and up-to-date list.
• Use `-backend=<string>` and `-backendoptions=<string>` flags to control the low-level compiler and its command line options (respectively).

• The `-S` flag may be added to instruct RCC to generate the transformed target code, but not to further compile the generated code using the low-level compiler. This flag is usually used so that multiple files can be linked together before creating the final executable.

For example, to compile for an x86 OpenMP target machine using `icc` and OpenMP parallelization, run RCC this way:

```
$ rcc --polymap --mm=core2duo-mapper.xml --march=x86_64-linux --backend="icc" --backendoptions="-O3 -openmp" -o foo foo.c
```

In the example above, the final generated executable is `foo`.

Note that because RCC recognizes abbreviated versions of `-mapper.xml` file names, you can drop the `-mapper.xml` part of the name.

Another run example given below is to generate an OpenMP parallel code for an Intel OpenMP target machine. In the example, the generated code is stored by default in `foo.gen.c`.

```
$ rcc -S --polymap --mm=core2duo --march=x86_64-linux foo.c
```

The filename format of the generated output is 
`<filename>.gen.<target_ext>`, where the value of `<target_ext>` depends on the supplied machine model. For instance, `<target_ext>` will be `c` or `cu` for OpenMP and CUDA targets, respectively.
4 Programming Guidelines

This section presents guidelines for developing C programs that can take advantage of the mapper component of RCC. The mapper can parallelize mappable regions of code that conform to the extended static control program structure. An extended static control program is a program that operates on dense matrices and arrays and that consists of for loops whose bounds are integer affine functions of outer loops and parameters, and array indexing functions that are integer affine functions of loop indices and parameters. Conditional expressions are also within the scope of extended static control programs.

The following pseudo C syntax summarizes the extended static control program structure:

```
parameters ::= M, N, ...
arrays ::= A, B, C, ...
loop indices ::= i, j, k, ...
e ::= integer affine expression of indices and parameters
S ::= A[e] = f(A_1[e_1], ..., A_n[e_n])
    | for (i = e; i < e; i += N) S
    | if (e) S else S
    | S; S; ...
```

Although the mapper currently parallelizes only loops, it parallelizes the entire loop, allowing parameterization of some values and parallelization of imperfectly nested loops. This behavior enables the mapper to parallelize complete linear algebra kernels and even fused or nested kernels.

We suggest that code be kept as simple as possible. Write a compute kernel that conforms to the extended static control program structure and that looks like a "textbook-style" serial C code. This includes eliminating any optimizations and parallelization techniques, including OpenMP pragmas, pthread calls, or the reuse of variables for other purposes. Including these optimizations interferes with parallelization. Leave the details of the implementation and performance to RCC.

Abiding by the following basic rules will align your programming methodology more closely with the algorithmic and mathematical conception of your application.

Affine Parametric Loop Bounds

The mapper of RCC can analyze and transform loops with known bounds that are integer affine functions of parameters and outer loop index variables. In general, an integer affine function is a linear polynomial of parameters or outer loop index variables taking the general form:

\[ f(p_1, \ldots, p_n) = A_1 * p_n + A_n * p + b \]

\(^1\)It refers to a generalized structure of loop nests in which each loop may contain one of more loops and statements
Constant Loop Counter Increments

where each \( p \) is a parameter or outer loop index variable, and \( b \) and each \( A_i \) is a constant.

However, loop bounds need not be compile-time constants. The following affine function is supported: \( f(i, j) = i - 7j + 9 + 2N \), where \( N \) is an unknown constant parameter. In contrast, the following examples of affine functions are not supported: \( f(i, j) = i/2 + j \) and \( f(i) = i*i \). However, these non-affine codes can "approximated" by affine code so that the rest of the codes can still be mapped. This technique is called blackboxing and explained in more details in Section 5.

Affine functions are an important class of functions for multiple reasons:

- They are statically analyzable, and represent most of the "known cases."
- They allow for simple conservative approximations of dynamic cases.
- They are closely related to program transformations.

An an example, the mapper of RCC can deal with the following form of code:

```c
double f(int n) {
    int i, j;
    for (i=0; i<n; i++) {
        for (j=0; j<i; j++) {
            C[i][j] = A[i][j] * B[j];
        }
    }
}
```

The loop nest shown above describes a (non-rectangle) triangular iteration space. The mapper can work with these loop bounds that are known only at runtime. The upper bound of the outer loop is not a compile-time constant, but is determined by an input parameter to the function when the function is called. Furthermore, in the inner loop, the loop bound changes for each iteration of the outer loop. However, because the bound is an integer affine function of the outer loop index variable, the region is still perfectly legal.

### Constant Loop Counter Increments

To handle a loop nest, the loop counters must be incremented by a constant. Here is a safe example of such a loop:

```c
for (i=10; i>0; i--) {
    for (j=0; j<20; j+=7) {
        A[i][j] = 2 * B[i][j];
    }
}
```

The following similar code, however, is not mappable (but can be blackboxed), due to irregular “holes” in its 2D iteration space that cause it to be non-affine:

```c
for (i=10; i>0; i--) {
    for (j=0; j<20; j=j+i) {
        A[i][j] = 2 * B[i][j];
    }
}
```
Affine Array Accesses

Similar to loop bounds, array index expressions must be integer affine functions of parameters and loop index variables of the enclosing loops. The following code shows such an example:

```c
double f(int n, int c) {
    int i, j;
    for (i=0; i<n; i++) {
        for (j=0; j<(2*i); j++) {
        }
    }
}
```

The next example is similar to the previous one, but it is not analyzable since the left-most index of array A is non-affine. This prevents RCC from employing its powerful mathematical optimizations:

```c
double f(int n, int c) {
    int i, j;
    for (i=0; i<n; i++) {
        for (j=0; j<(2*i); j++) {
            C[i+c][j] = A[i*j][3*j+c] * B[j];
        }
    }
}
```

While this example is mappable by RCC, it can be blackboxed, thereby allowing the rest of the code to be mapped by RCC.

Conditionals

RCC can also handle loops that contain conditional statements in the body. The supported types of conditionals are the ones that contain fixed values or affine functions of loop indices and parameters. Example of mappable conditionals are:

```c
if (A[i] < B[i]) and if (i-j < 0) and if (A[i] < N), where A and B are input/output arrays, i and j are loop indices, and N is a constant parameter.
```

Non-Analyzable Constructs

The following non-analyzable constructs should be avoided inside a mappable region:

- the use of `malloc()` or `malloc()`-like calls; `malloc()` should only be called outside the mappable region
- pointer codes and linearized array accesses; RCC currently does not support advanced pointer analysis
- references to `struct` or `union` members
Non-Analyzable Constructs

- indirect array accesses (e.g., \texttt{A[B[i]]})
- side effects

If your code has non-analyzable code, first try to rewrite it. If that is not possible, place the non-analyzable code snippets in a separate function call and then use blackboxing. Both these steps will be illustrated in the following section.
5 R-Stream Coding Best Practices

This section shows multiple examples of coding best practices for RCC. These examples will elaborate on several of the points made in the previous section.

Memory Allocation

Our first example deals with memory allocation. `malloc()` or `malloc()`-like calls are not supported within a mappable region, but they may be safely moved to an unmapped part of the program, as we see in the following example.

```c
double pA[256], pB[256];
#pragma rstream map
void stencil_1D_3pt(double *A, double *B) {
    double *temp;
    int nx=15, i, j, k;
    temp = (double *) malloc(nx*sizeof(double));
    for (i=1; i<(nx-1); i++) {
        temp[i] = 2 * A[i];
    }
    for (i=1; i<(nx-1); i++) {
    }
}
int main() {
    stencil_1D_3pt(pA, pB);
    // ...
}
```

In the previous example, the `temp` array is used to store temporary values while computing the 1D 3-point stencil. However, memory for the array is allocated within the mapped `stencil_1D_3pt()` function (the line marked in red), which is not permissible in RCC. Fortunately, this function can be easily modified to be RCC analyzable by moving the `malloc()` call outside of the mapped function, as follows:

```c
double pA[256], pB[256];
#pragma rstream map
void stencil_1D_3pt(double *A, double *B, double *temp, int nx) {
    int i, j, k;
    for (i=1; i<(nx-1); i++) {
        temp[i] = 2 * A[i];
    }
    for (i=1; i<(nx-1); i++) {
    }
}
int main() {
    int nx = 15;
    double *temp = (double *) malloc(nx*sizeof(double));
    stencil_1D_3pt(pA, pB, temp, nx);
    // ...
}
```
Now, the main() function contains the malloc() call, and the temp array is passed as a parameter to the stencil_1D_3pt() function. This results in analyzable RCC code.

There are two reasons why someone would want to allocate arrays with mallocs. Most usually, malloc is used because the size of the array depends on runtime values. In this case the only solution is the one presented above.

Another reason why one would allocate an array with malloc is to avoid stack overflow problems. Risks of stack overflows are actually taken care of automatically by RCC for arrays that are declared inside a mappable function (like temp in the original program). In the example above, if nx is a constant, you can allocate temp on the stack even if it would normally overflow the stack. R-Stream will use this to analyze accesses to temp and ensure that it is not aliased, and it will automatically allocate temp on the heap in the output code to avoid a stack overflow.

Multidimensional Arrays

The next example shows how to handle multi-dimensional arrays in RCC. For various reasons, programmers will often linearize (or “flatten”) multi-dimensional arrays before working with them, as seen here:

```c
int NX, NY, NZ;
double pA[256*256*256], double pB[256*256*256];
#pragma rstream map
void double_3D(double *A, double *B, int nx, int ny, int nz) {
    int i, j, k;
    for (i=0; i<nx; i++) {
        for (j=0; j<ny; j++) {
            for (k=0; k<nz; k++) {
                B[i*ny*nz + j*nz + k] = 2 * A[i*ny*nz + j*nz + k];
            }
        }
    }
}
int main() {
    double_3D(pA, pB, NX, NY, NZ);
    // ...
}
```

The problem with making the arrays one-dimensional is that the accesses are no longer affine functions, as seen in the red texts above. For RCC to analyze this code, the arrays must be kept multi-dimensional. As an added benefit, this is also a more natural and readable way to write the code, as seen here:

```c
int NX, NY, NZ;
double (*pA)[256][256], double (*pB)[256][256];
#pragma rstream map
void double_3D(double (*A)[256][256], double B[256][256][256],
               int nx, int ny, int nz) {
    int i, j, k;
    for (i=0; i<nx; i++) {
        for (j=0; j<ny; j++) {
            for (k=0; k<nz; k++) {
```
To map an multi-dimensional array, the array’s dimensions must be specified explicitly (only the leftmost dimension need not be specified. This code now has affine array indices and is mappable by RCC. Declarations of arrays A and B are equivalent, although declaration of B provides more information about the size of B to RCC.

Another technique that programmers often use is pointer swapping. For instance, in stencil codes, pointers are sometimes swapped in order to alternate the read and write arrays after each stencil sweep. The following code performs exactly that:

double pA[256], pB[256];
#pragma rstream map
void stencil_1D_3pt(double *C, double *D, int iter, int N) {
    double *temp_ptr;
    for (t=0; t<iter; t++) {
        for (i=1; i<N-1; i++) {
            D[i] = 2.0 * C[i] + 3.0 * (C[i-1] + C[i+1]);
        }
        // swap pointers to C and D
        temp_ptr = C;
        C = D;
        D = temp_ptr;
    }
}
int main() {
    stencil_1D_3pt(pA, pB, iter1, N1);
    // ...
}

However, this type of stencil swapping (highlighted in red text) destroys the static analysis performed by RCC. In general, it is best to avoid aliasing issues by steering clear of pointer code. In the case of this specific code, we can avoid pointer code by performing manual loop unrolling, as follows:

double pA[256], pB[256];
#pragma rstream map
void stencil_1D_3pt(double *C, double *D, int iter, int N) {
    int i, t;
    for (t=0; t<(iter-1); t=t+2) {
        for (i=1; i<N-1; i++) {
            D[i] = 2.0 * C[i] + 3.0 * (C[i-1] + C[i+1]);
        }
    }
}
Short-circuited conditionals

A common idiom used in C to combine conditionals are the short-circuited conditionals `&&`, `||`. When the first operand of `&&` is evaluated to false, the second operand is not evaluated, leading to a rather complex control flow, in particular when there is an `else` counterpart to the conditional. In the presence of short-circuit operators, the current conditional analysis in RCC tends to represent the conditionals in an approximate fashion, weakening their analysis.

One easy way of avoiding `&&` operators is to just nest the conditionals, as shown on the following example. The following code:

```c
for (i=0; i< N; i++)
    for (b=0; b<K; b++)
        if ((i+b < N) && b-i >=0)
            for (k=0; k<F && k<=i; k++)
                Y[i][b] += C[k] * X[i-k][b];
```

should be written as:

```c
for (i=0; i< N; i++)
    for (b=0; b<K; b++)
        if (i+b < N)
            if (b-i >=0)
                for (k=0; k<F; k++)
                    if (k <= i)
                        Y[i][b] += C[k] * X[i-k][b];
```

When the conditional is an affine function of loop indices and parameters, the conditional becomes part of the loop bounds. Hence no “empty iterations” are executed in the resulting mapped code.
Library Calls

External library calls within mapped functions, without any other changes, also result in code that is not analyzable by RCC. This is because the memory access pattern of the library calls is opaque to RCC. The following code presents such a scenario:

```c
extern void fft_1D(double v_in[N], double v_out[N], int fft_dir);
double px1[256], px2[256], px3[256], px4[256];
#pragma rstream map
void kernel(double x1[N], double x2[N], double x3[N], double x4[N]) {
    fft_1D(x1, x2, +1);
    scale(x2, x3);
    fft_1D(x3, x4, -1);
}
int main() {
    int i, titer = sscanf();
    for (i=0; i<titer; i++) {
        kernel(px1, px2, px3, px4);
    }
    // ...
}
```

As we can see, the `kernel()` function is called multiple times in `main()`, so it is important to map this function, despite the library calls to `fft_1D()`. We can do this by creating an image function for `fft_1D()`. The purpose of an image function is to describe the accesses made by the original function in a conservative manner. During RCC’s mapping phase, this description allows the compiler to "pretend" to map the image function, even though the output program will still execute the original code.

There are a few points to be aware of regarding image functions:

- The image function must be RCC mappable.
- The arguments of the library function and the image function must match.
- The image function must perform a superset of all the possible reads and writes of the original function.
- RCC may simplify the code in the image function before it gets to the mapper. As a result, avoid writing dead code (i.e., code that can be easily removed without changing the semantics of the image function).

By following these points, here is one safe image function for the `fft_1D()` library call:

```c
void image_fft_1D(double v_in[N], double v2_out_r[N], int fft_dir) {
    int i;
    for (i=0; i<N; i++) {
        v2_r[i] = v1_r[i];
    }
}
```

We then need to verify that the appropriate pragmas are used:
extern void fft_1D(double v_in[N], double v_out[N], int fft_dir);
#pragma rstream map image_of:fft_1D
void image_fft_1D(double v_in[N], double v2_out_r[N], int fft_dir) {
  int i;
  for (i=0; i<N; i++) {
    v2_r[i] = v1_r[i];
  }
}
#pragma rstream map
void kernel(double x1_r[N], double x2_r[N], double x3_r[N], double x4_r[N]) {
  fft_1D(x1_r, x2_r, +1);  
  scale(x2_r, x3_r);
  fft_1D(x3_r, x4_r, -1);
}
int main() {
  int i, titer = sscanf();
  for (i=0; i<titer; i++) {
    kernel(px1, px2, px3, px4);
  }
  // ...
}

The image_of pragma shown in red text above tells the compiler that the image_fft_1D() function is an image function for fft_1D(). This technique is called blackboxing, and it allows RCC to map a function even when parts of it are not mappable. In this example, the library calls were unmappable, but blackboxing allowed us to safely map the remaining code in the kernel() function. In the next example, we will use blackboxing to deal with indirect array accesses.

Unmappable Code

In this example, we deal with code that has indirect accesses (i.e. A[B[i]]):

#pragma rstream map
void kernel(double (*A)[N], double *B, int n) {
  for (i=0; i<n; i++) {
    for (j=0; j<n; j++) {
      A[B[i]][j] = i+j;
    }
  }
}

Indirect accesses are often found in sparse codes. However, they create non-affine array indices, resulting in unmappable code. The first step to making this mappable code is to outline (isolate in a separate function) the non-analyzable code, as follows:

void ind(double (*A)[N], double *B, int i, int j) {
  A[B[i]][j] = i+j;
}
#pragma rstream map
void kernel(double (*A)[N], double *B, N) {
  for (i=0; i<n; i++) {
    for (j=0; j<n; j++) {
      // ...
    }
  }
}
So far, we have extracted the unmappable code into the \texttt{ind()} function, which is then called from original \texttt{kernel()} function. The next step is to create an image function for \texttt{ind()}, following the same image function rules described in the previous "Library Calls" example:

\begin{verbatim}
#pragma rstream map image_of:ind
void ind_image(double (*A)[N], double *B, int i, int j) {
    int k;
    for (k=0; k<N; k++) {
        A[k][j] = B[i] + 3;
    }
}
\end{verbatim}

This blackboxing allows RCC to map the rest of the \texttt{kernel()} function and produce correct code, even though the indirect accesses are not analyzed or optimized.
A high-level compiler like RCC relies on back-end compilers whose behavior may be unpredictable. In addition, analytical models employed by compilers often do not model the entire problem. Hence, a tool for searching the space of mapping solutions is desirable. For RCC, users can utilize the ARCC tool (the auto-tuner for RCC) to automatically and empirically find a good combination of RCC’s advanced compiler option values that enable generation of a code with close-to-optimal performance.

The use of ARCC is straightforward. Users only need to know:

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

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

For instance, a user can use ARCC to auto-tune a matrix-multiply code on an SMP machine by executing the following commands:

```sh
$ cd ~/work/rstream-x.x.x/rstream/examples/openmp
$ arcc --build="make PROGRAMS=matmult_c" --run="./matmult_c"
--clean="make clean"
```

Once the ARCC run is complete, the matmult_c.gen.c file is the final auto-tuned code, and the matmult_c binary is its corresponding executable.

Note that the build system will not rebuild the program if it is up-to-date. In this case, the --clean option is required. However, if the build system overwrites the existing program, the clean option is not needed, and it can be assigned the empty string:

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

Essentially, ARCC auto-tuning is done in two steps. In the first step (called production mode), meta-data is produced by RCC tactics (i.e., the optimization components of the mapper) during the first build. The produced meta-data defines the syntax of the tactic’s options and the space of values that ARCC can set the options to.

In the second step (called consumption mode), ARCC iteratively instantiates solutions in the search space defined by the meta-data and produces option instances that will be used by RCC. 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.

To see what ARCC is doing when auto-tuning a given code, use the --verbose flag. More details on ARCC including its advanced usage are available in the R-Stream Parallelizing C Compiler Power User Guide.

---

1Note: As per the release notes, the precompiled python bytecode files supporting ARCC require Python 2.7.3 to execute properly
7 Code Examples

Concrete code examples are provided that demonstrate the various features of RCC. Since the goal here is to illustrate RCC’s functionality, not to show performance, the RCC’s compilation flags (defined in the makefiles) are the minimal flags to generate parallel code for a selected target architecture. Note that these flags can be further optimized for performance.

Where to Find the Examples

The examples are listed in the ~/work/rstream-x.x.x/rstream/examples directory.

Directory Structure

Currently, the examples directory is structured as follows:

- **src**: common source codes for all the examples.
- **openmp**: directory containing makefile for OpenMP target.
- **cuda**: directory containing makefile for CUDA target.
- **tilera**: directory containing makefile for Tilera target.
- **ensign**: directory containing makefile for Ensign OpenMP Tensor target examples.
- **ocr**: directory containing the Open Community Runtime (OCR) target examples.
- **swarm**: directory containing ET International’s SWARM library target examples.
- **runtime-agnostic edts**: Self-contained directory with Makefile examples, source code and directories for building and running C++-based EDT (Event-Driven Tasks) codes for different runtimes. Please see runtime-agnostic-edt/README for more information.
- **make.header**: generic independent part of makefile (header).
- **make.footer**: generic independent part of makefile (footer).
- **make.openmp**: OpenMP-specific part of makefile.
- **make.cuda**: CUDA-specific part of makefile.
- **make.tilera**: Tilera-specific part of makefile.
- **make.ocr**: OCR-specific part of the makefile.
- **make.swarm**: SWARM-specific part of the makefile.
Running the Examples

- `make.tune_edt`: EDT-specific part of the makefile.
- `apps`: directory containing independent examples of application programs.
- `auto-tuning`: directory containing makefile for running ARCC (the auto-tuner for RCC).

A few notes:
- All of the source codes containing the kernels that are to be mapped by RCC are kept in the `src` subdirectory. Currently, the examples directory is designed to support OpenMP, CUDA, and Tilera target architectures. Every target has its own subdirectory, and a makefile is contained in each target subdirectory.
- Examples of application programs mappable by RCC that are given in the `apps` subdirectory. In that `apps` subdirectory are several subdirectories, each representing an independent application program.
- The `auto-tuning` subdirectory is given to demonstrate how ARCC can be used for automatically tuning the kernel codes and the application programs.

Running the Examples

To compile a kernel program using RCC for a specific target, follow the commands below:

```
$ cd ~/work/rstream-x.x.x/rstream/examples/<target>
$ make PROGRAMS=<kernel-program>
```

then the `<kernel-program>` file is the executable generated by RCC.

To compile and an application program for a specific target, use the following commands:

```
$ cd ~/work/rstream-x.x.x/rstream/examples/apps/<app-program>
$ make TARGET=<target>
```

then the `<app-program>` file is the executable generated by RCC.

The following commands can be used to auto-tune a kernel program and an application program using ARCC:

```
$ cd ~/work/rstream-x.x.x/rstream/examples/auto-tuning
$ make TARGET=<target> PROGRAMS=<kernel-program> APPS=<app-program>
```

then the results of the auto-tuning can be seen in ARCC-generated log files (i.e., `<program>-arcc-result.log`). The generated executables are the auto-tuned programs.

Further Documentation

More detail about the code examples is documented in each source file and also in the README files stored in the

```
~/work/rstream-x.x.x/rstream/examples directory and its subdirectories. Please consult these files before running the examples.
```
8 For More Reading

This document is a concise introduction to the R-Stream Compiler. The compiler is a powerful tool with many advanced options that were not covered here. More information about the compiler can be found in *R-Stream Parallelizing C Compiler Power User Guide* and more information about the machine model can be found in *Mapper Machine Model for the R-Stream Compiler*.