Trimaran: A Compiler and Simulator for Research on Embedded and EPIC Architectures

Version 4.0

support@trimaran.org

April 1, 2007

Contents

1 Introduction **3**
1.1 What is new in this version? 3
1.2 Supported Instruction Sets (ISAs) 4

2 Installation **4**
2.1 Software Requirements 5
2.2 Manual Installation 5
2.3 LivePC Installation 6

3 Trimaran Organization **6**
3.1 OpenIMPACT 7
3.2 Elcor 9
3.3 Simu 9
3.3.1 Codegen 10
3.3.2 Emulib 10

4 Running Trimaran **11**
4.1 Using `tcc` 12
4.2 OpenIMPACT 12
4.2.1 Running OpenIMPACT manually 13
4.3 Elcor 13
4.3.1 Running Elcor manually 14
4.4 Simu 14
4.4.1 Running Simu manually 15

5 Automatic Vectorization **15**

6 s2lc: SUIF to Lcode Conversion **16**
6.1 Download and Installation 16
6.1.1 Installing SUIF 16
6.1.2 Installing `s2lc` 17
6.2 Description of `s2lc` Passes 17
6.3 Using `s2lc` with Trimaran 18

7 M5elements Cache Simulator **19**
7.1 Installation 20
7.2 Simulating Benchmarks with M5elements 20

8 ARM Port **21**
8.1 Installation of the ARM Port 21
8.2 Running Benchmarks using the ARM Port 23

9 Adding a Benchmark **23**

10 Modifying Elcor **25**
10.1 Code Examples 25
10.2 Adding Your Own Code 26
10.3 Adding a Command Line Parameter 27
10.4 Templates 28

11 Adding a New Clustering Algorithm **29**

12 MDES: Machine Description **29**
12.1 HMDES vs. LMDES 30
12.2 Macro Registers 30
12.3 Stack Descriptor Section 30
12.4 Literal Formats 31
12.5 Vector Architectures 31

13 Adding a new Opcode **31**
13.1 Adding Opcodes to MDES 32
13.2 Adding Opcodes to Elcor 34
13.3 Adding Opcodes to OpenIMPACT 35
13.4 Adding Opcode to Simu 35

14 Customized Instructions **36**

15 Troubleshooting **38**
15.1 Benchmark Debugging Tips 39
15.1.1 Simulating Each Source File Individually 39
15.1.2 Turning Off Elcor Passes 39

16 Help! My question wasn't answered here. **40**

1 Introduction

Trimaran is an integrated compiler and simulation infrastructure for research in computer architecture and compiler optimizations. Trimaran is highly parameterizable, and can target a wide range of architectures that embody embedded processors, high-end VLIW processors, and multi-clustered architectures. Trimaran also facilitates the exploration of the architecture design space, and is well suited for the automatic synthesis of programmable application specific architectures. It allows for customization of all aspects of an architecture, including the datapath, control path, instruction set, interconnect, and instruction/data memory subsystems.

The modular nature of the compiler and the hierarchical intermediate program representation used throughout makes the construction and insertion of new compilation modules into the compiler especially easy. Trimaran is already populated with a large number of existing compilation modules, providing leverage for new compiler research as well as education in advanced compiler topics. The Trimaran Graphical User Interface (GUI) makes the configuration and use of the system surprisingly easy.

Among the rich suite of compiler analysis and optimizations are:

- Advanced region formation algorithms (e.g., superblocks and hyperblocks) to expose instruction level parallelism with speculation and predication,
- Various backend instruction partitioning and mapping algorithms for automatically distributing parallelism in a multi-clustered architecture,
- A first of its kind back-end vectorizer that extracts and exploits data level parallelism using short vector instructions (SIMD),
- Various register allocation heuristics,
- Instruction scheduling algorithms including software pipelining with modulo scheduling.

Although there are several compiler infrastructures available to the research community, Trimaran is unique in that it is especially geared toward compiler and architecture research. Trimaran is used for designing, implementing, and testing new optimizations, as well as the evaluation of various architectural innovations. Trimaran is also widely used for teaching and education purposes at several universities worldwide.

1.1 What is new in this version?

We are committed to releasing a robust, tested, and documented system. Our website (http://www.trimaran.org) provides the latest information on Trimaran, and includes links to download the system, as well as documentation and other useful resources.

The following is a summary of the newest Trimaran features made available in the current release.

- Support for multi-cluster architectures. The clusters can be organized as either sharing an inter-cluster communication bus or a mesh point-to-point operand network.
- Support for application-specific instruction-set extensions.
- Support for automatic vectorization.

- Support for Fortran applications via a SUIF to Trimaran translator.
- Advanced simulation of the memory system using the M5 simulator.
- Code generation infrastructure to handle ISAs with arbitrary literal bit-width constraints.
- Code generation for the ARM ISA.
- New datatype attribute associated with every operand to describe the data type (integer/float/predicate), whether the operand is signed or unsigned, and the operand bit-width.
- Support for the `long long` datatype
- Modulo variable expansion to support modulo scheduling without rotating registers or other hardware support.
- Significantly improved code quality. Register allocation has been completely rewritten and many optimizations have been added to the default path.
- Lots of bug fixes

1.2 Supported Instruction Sets (ISAs)

Trimaran generates and evaluates code for two instruction set architectures (ISAs). The primary/native ISA is HPL-PD, but the ARM ISA is also supported. A brief description of each ISA follows.

- HPL-PD [4] is a parametric VLIW architecture. It admits processors of different composition and scale, especially with respect to the amount of parallelism offered. The HPL-PD parameter space includes the number of clusters in a multi-cluster processor, the make up of each cluster (e.g., types of functional units, the composition of the register files), and the instruction set including operation latencies and descriptors that specify when operands may be read and written, instruction formats, and resource usage behavior of each operation. The architecture instruction set is akin to a RISC load-store architecture, with standard arithmetic and memory operations. It also supports speculative and predicated execution, compiler exposed memory systems, a decoupled branch mechanism, software pipelining, and most recently, short vector instructions.
- ARM is a architecture that is widely popular in embedded systems. The instruction set is also similar to a RISC load-store architecture. It supports conditional execution of most instructions. More information on the ARM instruction set can be found in [9].

2 Installation

There are two methods of installing Trimaran:

- **Manual installation**: This allows you more control over the installation process, and can be useful if the automated installation does not work on your system. Make sure you have the requirements described in Section 2.1, then see Section 2.2.
- **LivePC installation**: For Windows users. This is a virtual applicance/virtual machine-based version of Trimaran that allows you to easily get Trimaran up and running without installing. See Section 2.3.

2.1 Software Requirements

This Trimaran release has been extensively tested using Fedora Core 3, 4 and 5 on x86 Linux, as well as Redhat Enterprise Linux 4 running on x86-64. Users working with other operating systems may experience some obstacles in the installation process[1].

Trimaran requires the following software packages:

- `gcc` (tested extensively with version 4.0.1, but 3.2 and 3.4.4 should also work)
- `autoconf` version 2.54 (may require this exact version)
- `automake` version 1.7 (may require this exact version)

The optional Trimaran GUI requires the following additional packages:

- TCL 8.0 or later (binary named `tclsh`)
- Tk 8.0 or later (binary named `wish`)

Please note that in order to install the Trimaran compiler on a 64-bit system, the 32-bit development libraries must be installed (e.g. glibc-devel-X.i386.rpm and libstdc++-devel-Y.i386.rpm[2] where X and Y are version numbers corresponding to the x86_64 versions of these libraries).

2.2 Manual Installation

1. Make sure your system has the software requirements described in Section 2.1.
2. Fetch the Trimaran release sources from http://trimaran.org/download.shtml
3. Unpack the sources:

```
tar xzf trimaran_4_0.tar.gz
cd trimaran
```

4. Setup your environment: Several environment variables are required to build or use the Trimaran toolchain. We will refer to the directory housing Trimaran as `<TRDIR>`. The files `envrc` and `envrc.bash` located in `<TRDIR>/trimaran/scripts/` provide a convenient way of loading the variables into your shell environment via the shell `source` command. First modify the file `envrc[.bash]` to replace `TRIMARAN_ROOT` with the path to your Trimaran installation (e.g., `<TRDIR>/trimaran`), then if you are using csh or tcsh:

```
% source <TRDIR>/trimaran/scripts/envrc
```

[1] For example, Ubuntu systems link 'sh' to the 'dash' shell which has strict POSIX compatability. Some trimaran components require non-POSIX extensions that existed in 'sh', so it's necessary to either change the link, or change the offending scripts to use 'bash' instead.

[2] Note that in some Fedora releases, using `yum` to install libstdc++-devel.i386 will uninstall libstdc++-devel.x86_64. Since this is probably undesirable, one should manually download libstdc++-devel.Y.i386.rpm and install using the `rpm -i` command.

or if you are using bash:

```
% source <TRDIR>/trimaran/scripts/envrc.bash
```

It might be helpful to add one of these commands to your shell startup script (e.g., `.cshrc` or `.bashrc`).

5. Build and install:

```
% cd openimpact; ./install_openimpact
% cd ../elcor; make
% cd ../simu; make
```

6. Install optional packages:

```
% wget http://www.graphviz.org/pub/graphviz/ARCHIVE/graphviz-2.8.tar.gz
% tar xzf graphviz-2.8.tar.gz
% cd ../graphviz-2.8/; ./configure; make; make install
```

2.3 LivePC Installation

The easiest way to use Trimaran is the Trimaran LivePC, a virtual appliance that we maintain and distribute to ease the process of installing or upgrading Trimaran. The Trimaran LivePC is based on a lightweight Linux operating system and contains everything needed to run Trimaran. It is precompiled and ready to use. It is also the most convenient way for Windows users to install Trimaran. The LivePC is currently not available for Linux users.

The Trimaran LivePC requires Windows XP SP2 and the LivePC engine from `http://www.moka5.com`. Installation instructions for the engine are available at:

```
http://www.moka5.com/products/getstarted.html
```

Once the engine is installed, download the Trimaran LivePC from:

```
http://www.trimaran.org/livepc
```

3 Trimaran Organization

Trimaran is comprised of three components: the OpenIMPACT compiler, the Elcor compiler, and the Simu simulator (shown in Figure 1). Trimaran uses OpenIMPACT to compile the original source code into an assembly intermediate representation (IR) called Lcode. The Lcode produced is optimized for ILP, but not for a specific machine. This code is then passed to the Elcor compiler, along with a machine description (MDES) that specifies the target machine. Elcor compiles the code for the target machine, producing another IR called REBEL. The Trimaran simulator known as Simu consumes the REBEL code, executes the code, and gathers execution statistics.

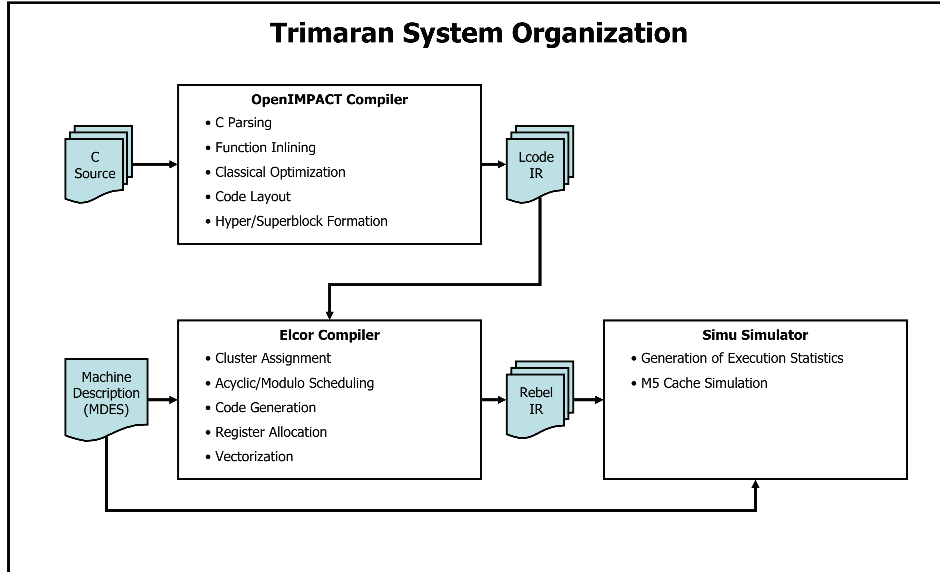

Figure 1: Overview of compilation steps in Trimaran.

3.1 OpenIMPACT

The OpenIMPACT compiler is maintained by the IMPACT group (`http://www.crhc.uiuc.edu/Impact/`) at the University of Illinois. Within Trimaran, a slightly modified version of OpenIMPACT is used to compile the original C source into assembly code. The main passes of OpenIMPACT are detailed below, and illustrated in Figure 2.

- **EDG front end**: The original C source is parsed by a front end licensed from the Edison Design Group (EDG). It is converted to an abstract syntax tree representation called Pcode.
- **Pcode transformation and analysis**: Passes such as flattening and function inlining perform operations on the syntax tree. Optionally, interprocedural analysis (IPA) analyzes the objects pointed to by memory operations (pointer analysis), and annotates the Pcode with this information.
- **PtoL lowering**: The abstract syntax tree (Pcode) is converted to a machine-independent assembly IR called Lcode.
- **Classic optimizations**: The Lopti pass performs classic optimizations.
- **Lcode profiling**: The assembly code is simulated to obtain profiling information.
- **ILP optimizations**: Several passes may perform loop unrolling, superblock formation, and hyperblock formation on the Lcode.
- **Memory profiling**: After optimizations, the Lcode is simulated again, this time to obtain profile information about memory conflicts and cache behavior.
- **Trimaran bridge**: The Lhpl_pd pass performs code generation specific to the HPL-PD architecture (see Section 1.2) used by Trimaran.

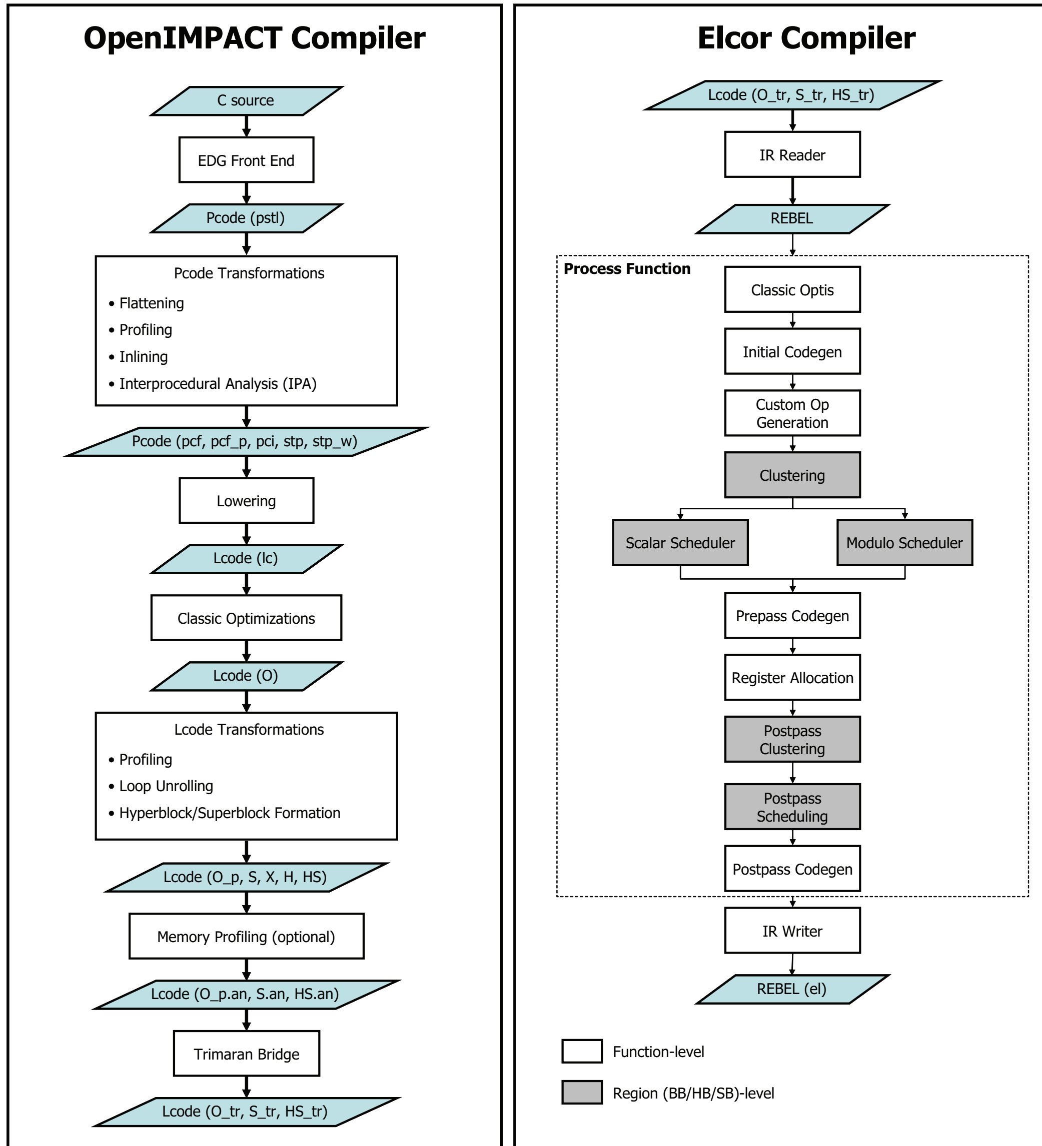

Figure 2: Compilation steps in OpenIMPACT and Elcor. File extensions shown in () where appropriate.

3.2 Elcor

The Elcor compiler is a VLIW compiler that takes largely machine-independent assembly code and compiles/optimizes it for a specific machine described in a given machine description (MDES). The input assembly code is usually generated by OpenIMPACT in Lcode format, and output in the Elcor native REBEL format; however, Elcor is capable of reading or writing either format.

Elcor processes one function at a time (note that OpenIMPACT may have inlined some functions into others). Each function undergoes the following steps:

- **Classic optimizations**: Classic optimizations such as common subexpression elimination, constant propagation, dead code elimination, and others are performed.
- **Initial code generation**: This converts operands to a suitable format for the target architecture.
- **Custom operation generation**: This optional step finds subgraphs within the dataflow graph that are suitable for conversion to custom operations that accelerate execution, as detailed in Section 14.
- **Vectorization**: This optional step identifies and exploits data-level parallelism for efficient execution on architectures with SIMD extensions, as detailed in Section 5.
- **Instruction clustering**: For multi-clustered architectures, instructions are partitioned among the clusters, and move operations are orchestrated to transport operands between clusters.
- **Prepass scheduling**: The code is scheduled region-by-region using the scalar scheduler or the modulo scheduler. The modulo scheduler is used to software pipeline loops. It also performs rotating register allocation.
- **Prepass code generation**: Code generation binds the operations on the clusters to their respective register files.
- **Register allocation**: Physical registers are allocated for scalar operands, and spill code is generated.
- **Postpass clustering**: Newly inserted register spill code is assigned to clusters.
- **Postpass scheduling**: Scheduling binds newly added spill code to resources.
- **Postpass code generation**: A final pass of code generation is performed to appropriately bind register spill code to register files.
- **IR writer**: This outputs the code, usually in REBEL format.

3.3 Simu

The Trimaran simulator (Simu) supports the HPL-PD parametric architecture. (See Section 1.2 for a description of HPL-PD.) It takes REBEL assembly code as input, and creates a C program that runs on the native machine, and emulates how the program would execute on the HPL-PD architecture. The simulator generates multiple statistics to summarize the program execution.

The simulator has a static component and a dynamic component. The static component is called Codegen, and the dynamic part is Emulib. The first generates a low-level C file similar to an assembly file from the REBEL IR. The file is then compiled using the host native C compiler and linked to the emulation library (Emulib) to simulate the application. This is illustrated in Figure 3. C is used as the Codegen output to provide platform independence so that the simulation can run on any platform, without modification.

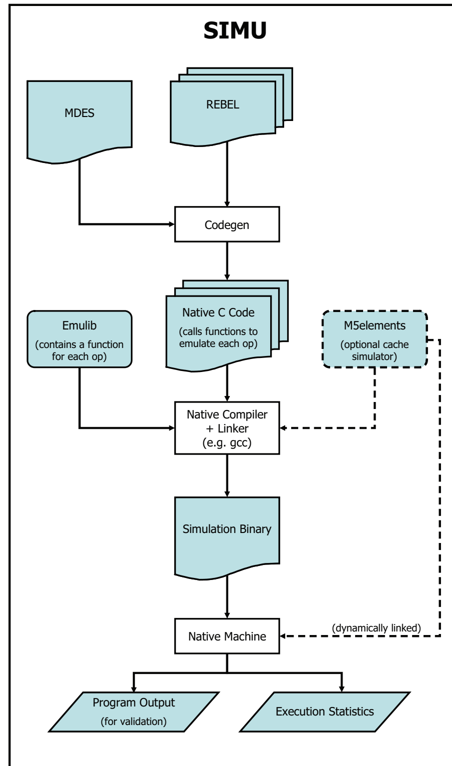

Figure 3: Flowchart for the simulator.

3.3.1 Codegen

For each input REBEL file, Codegen creates four output files:

1. A file with a `.c` extension. This file contains a series of stub routines to inter-operate with native code. There is one stub per original procedure appearing in the original source code file.
2. A file with a `.inc` extension. This file contains external variable declarations, global data, and structure and union layouts that mirror the original C source. This information is required for inter-operation with native code. In addition there are data required by the runtime emulation library (Emulib) for tracking execution statistics and profiling information.
3. A file with a `.tbls` extension. This file is a collection of emulation tables. One emulation table is maintained per procedure of the original source file. Each emulation table lists a sequence of instructions and their operands.
4. A file named `benchmark_data_init.simu.c`. This file initializes the program's global data structures.

3.3.2 Emulib

The emulation library provides an interpreter and a set of emulation routines for the HPL-PD virtual machine. The library supports speculation, predication, software pipelining with modulo scheduling and

rotating registers, clustered HPL-PD architectures, short vector instruction register files, and scalar register files.

The interpreter is invoked on every procedure entry. It emulates the instruction stream in a loop until the procedure returns. There is one emulation function for every HPL-PD operation. These emulation functions are automatically generated from the operation specification. The specification of HPL-PD operations consists of its I/O format and its actions.

The emulation library provides a rich facility for tracing execution. There are built-in plugins to record the dynamic control flow of an application, or the address trace. The plugins are user programmable and allow users to selectively enable and disable the tracing functionality at the level of functions, regions, or operations.

The emulation tracks a multitude of statistics to summarize the program execution. It also provides an interface for simulating the memory hierarchy. The optional M5elements simulator described in Section 7 allows cycle-accurate modeling of the memory system.

4 Running Trimaran

The Trimaran C Compiler script (`tcc`) is the easiest way to use Trimaran. We show how to run Trimaran for an example benchmark called `fir_int` (an integer version of a Finite Impulse Response filter). The following commands compile and simulate `fir_int` using the default settings:

```
% cd /tmp
% tcc -bench fir_int
```

A directory `/tmp/fir_int_O` is created, and will contain intermediate files corresponding to the compilation and simulation of the benchmark `fir_int`. These files are useful for scrutinizing the compiler output, debugging, and performance analysis. If the compilation and simulation complete successfully, `tcc` will print the following at the end of its output stream:

```
tcc: Result Check *SUCCESSFUL* for Benchmark fir_int on
input input1, region type O.
```

The file `/tmp/fir_int_O/elcor_intermediate/ELCOR_STATS` will summarize a few key statistics about the benchmark. The statistics can be totaled for the entire benchmark using the `Sumstat` utility provided with Elcor:

```
% Sumstat -total -i ELCOR_STATS
```

These statistics, which include performance estimations, are based on profiling information.

Actual simulation results are recorded in `/tmp/fir_int_O/simu_intermediate/PD_STATS`. The statistics are aggregated for each function, and then totaled for the entire program. The simulator tracks a number of statistics including:

- `total_cycles`: the total simulated execution time of the benchmark in cycles.
- `compute_cycles`: the total number of computation cycles assuming all branches are perfectly predicted, and all memory operations (e.g., loads and stores) latencies are less than or equal to the MDES specified latencies.
- `stall_cycles`: the number of processor stall cycles due to branch mis-predictions and memory system delays.

4.1 Using `tcc`

The main driver script `tcc` (located in `$TRIMARAN_ROOT/scripts`) will execute the entire Trimaran toolchain. The `tcc` script looks for benchmarks in the `$TRIMARAN_ROOT/benchmarks` directory. The command "`tcc -bench epic -region h`" would compile and run the benchmark `epic` using hyperblocks. A workspace directory with the benchmark name appended with a `_HS`, `_S`, or `_O`, is created depending on the type of region formation that is used during compilation (hyperblock, superblock, or basic block, respectively). The intermediate files generated by OpenIMPACT, Elcor, and Simu are stored in their respective subdirectories within the workspace directory. These files can be later used to run the individual tools separately without having to invoke all of Trimaran.

Running "`tcc -help`" lists the available options. Commonly used options are described below.

`-bench`: specify benchmark to compile.

`-region`: specify regions impact is allowed to form, can be `b|s|h|all`.

`-s{i|e|s|r}`: specify extent of compilation (compile through OpenIMPACT, Elcor, Simu Codegen, run).

`-M`: specify MDES (machine description) file used by Elcor.

`-E`: parameters passed directly to Elcor, e.g., `tcc -E"-Fdo_modulo_scheduling=no"`.

`-S`: parameters passed directly to Simu, e.g., `tcc -S"-Femulate_unscheduled=yes"`.

4.2 OpenIMPACT

The top level script for OpenIMPACT is `compile_bench`, located in `$IMPACT_ROOT/scripts`. Running the script without any parameters will list available options. Additional parameters to OpenIMPACT are found in `$IMPACT_ROOT/parms`. Default parameter settings are found in the `*_DEFAULTS` files. These values can be overwritten by `STD_PARMS.IMPACT`, which can be subsequently overwritten by arguments passed from the command line. Commonly used options include:

- `opti_level`: degree of optimization from 1 to 4.
- `do_loop_unroll`: permit loop unrolling.
- `max_unroll_allowed`: amount of unrolling allowed.
- `regroup_only`: prevent inlining.

- max_expansion_ratio: amount code expansion due to inlining allowed.

4.2.1 Running OpenIMPACT manually

We now show how to run OpenIMPACT manually using the same example benchmark from the beginning of Section 4, fir_int. When we ran tcc, it created the fir_int_O/impact_intermediate directory. We can use this directory, or put the OpenIMPACT files anywhere. (You don't need to run tcc first.) To run OpenIMPACT manually, try:

```
% mkdir -p /tmp/fir_int_O/impact_intermediate   # optional
% cd /tmp/fir_int_O/impact_intermediate         # optional
% compile_bench fir_int -c2O_tr -p $TRIMARAN_ROOT/openimpact/parms/STD_PARMS.IMPACT -project full
```

In the compile_bench command above, -c2O_tr tells OpenIMPACT to compile from C source code to the Trimaran bridge Lcode files, with file extension .O_tr. One of these .O_tr files is created for each original C source file. The final output from OpenIMPACT, which will be the input to Elcor, is a gzipped tar archive of the Trimaran bridge Lcode files (fir_int.O_tr.tgz).

You may notice that additional gzipped tar archives (.tgz) are created; these show the benchmark intermediate representation after various steps in OpenIMPACT compilation. Figure 2 helps explain the meaning of the file extensions.

4.3 Elcor

The top level script for Elcor is gen_elcor.pl (located in $ELCOR_HOME/bin). Since there often are multiple files in a benchmark, gen_elcor.pl calls the elcor binary for each file. Parameters read by Elcor are found in $ELCOR_HOME/parms. The ELCOR_PARMS_FILE environment variable, $ELCOR_HOME/parms/ELCOR_PARMS by default, specifies the list of parameter files used by Elcor. The default values can be overwritten by arguments passed through the command line. The file DRIVER_DEFAULTS contains the main switches affecting Elcor compilation (analysis, optimizations, etc.). Options found in DEBUG_DEFAULTS turn debug output on/off and control the level of detail. Commonly used options include:

- do_classic_opti: perform classical optimizations e.g., common subexpression elimination.
- do_{prepass|postpass}_scalar_scheduling: perform prepass/postpass scheduling on hyperblocks, superblocks, and basic blocks.
- do_modulo_scheduling: perform modulo scheduling.
- do_scalar_regalloc: perform scalar register allocation.
- memvr_profiled: perform memory dependence profiling.
- {input|output}_format: specify input/output file types (rebel, lcode).
- lmdes_file_name: machine description (MDES) file. Same as "-M" when using tcc.

4.3.1 Running Elcor manually

We now describe how to run Elcor manually using the same example benchmark from the beginning of Section 4, `fir_int`. To run Elcor, we need the output from OpenIMPACT, an archive of Trimaran bridge Lcode files (`fir_int.O_tr.tgz`). If we ran OpenIMPACT automatically through `tcc`, this file can be found in `fir_int_O/impact_intermediate`.

Once we have the archive, we uncompress it to get a `.O_tr` file for each original file in the benchmark. For `fir_int`, we have two files:

```
fir_int.O_tr
main.O_tr
```

We can now run Elcor on each of these files separately:

```
% elcor -i fir_int.O_tr -o fir_int.O_el
% elcor -i main.O_tr -o main.O_el
```

The `-i` parameter specifies an input file, and `-o` specifies an output filename. To pass extra parameters to Elcor, we prepend the parameter setting with `-F`. For example, if we wanted to turn off postpass scheduling, we would run:

```
% elcor -i fir_int.O_tr -o fir_int.O_el -Fdo_postpass_scalar_scheduling=no
% elcor -i main.O_tr -o main.O_el -Fdo_postpass_scalar_scheduling=no
```

Note that if we are using superblocks, `O_{tr|el}` becomes `S_{tr|el}`, or if we are using hyperblocks, it becomes `HS_{tr|el}`.

4.4 Simu

The two main scripts for the simulator are found in `$SIMU_HOME/bin`: `genEtoC.pl` runs Codegen on REBEL IR generated by Elcor and produces C files that are compiled with `genCtoO.pl` into object files. Commonly used options include:

- `emulate_unscheduled`: ignore VLIW scheduling and execute code sequentially.
- `emulate_virtual_regs`: ignore register allocation and simulate virtual, instead of physical, registers.
- `interlocking`: enable pipeline interlocking.
- `do_memory_simulation`: enable simulation of the memory hierarchy.
- `control_flow_trace`: trace the dynamic control flow of execution.
- `address_trace`: emit a trace of load and store addresses during execution.
- `generate_assembly`: emit mock assembly code for the input file; used primarily for debugging.

4.4.1 Running Simu manually

The simulator can also be run manually; we continue using the example benchmark from the beginning of Section 4, `fir_int`. We will need the output of Elcor, which is an archive of REBEL files, `fir_int.O_el.tgz`. It may help to refer to Figure 3 during this process. Follow these steps to run Simu manually:

1. Uncompress the archive of REBEL files:

```
% tar xzf fir_int.O_el.tgz
```

2. Run Codegen to generate C files for each REBEL file:

```
% ls *.O_el | gen_EtoC.pl 1 .
```

3. Compile the C files we just generated into object files:

```
% ls *.O_el | gen_CtoO.pl 1 .
```

4. Link the object files with Emulib, creating a simulation binary. Here we call our binary `fir_int_O`:

```
% gcc -o fir_int_O *.O_el.simu.o benchmark_data_init.simu.o -L$SIMU_HOME/lib -lequals -lm
```

5. Finally, run the binary as if it were natively compiled, using whichever command line parameters you desire:

```
% ./fir_int_O
```

5 Automatic Vectorization

Among the new optimizations offered in Trimaran is a back-end vectorizer that can identify and exploit data level parallelism for efficient execution on architectures with SIMD (single-instruction multiple-data) support. The technique implemented in Trimaran is called selective vectorization [6]. It creates highly efficient instruction schedules by distributing computation between scalar and vector functional units to improve resource utilization. For processor models that contain an abundance of scalar *and* vector processing units, selective vectorization creates loops with a balance of both vector and scalar instructions. Since vector and scalar occupy the same loop body, scalar operations are unrolled by a factor of the vector length. As such, the technique is most applicable to the short-vector instruction sets commonly found in today's *multimedia extensions*.

The Elcor parameter `do_vectorize` in `$ELCOR_HOME/DRIVER_DEFAULTS` enables the vectorization of loops during compilation. The vectorizer is most effective, and in many cases only applicable, when it has precise memory dependence information. For this reason, we recommend extracting dependence information using the SUIF front-end as described in Section 6. Selective vectorization is highly coupled with software pipelining and is intended for use in conjunction with the Trimaran modulo scheduler.

The options below, which are found in `$ELCOR_HOME/VECTORIZER_DEFAULTS` along with some others, provide some control in applying the vectorizer:

- `vectorize_model`: The vectorizer will apply different techniques and heuristics according to the model selected. There are four current models implemented. They are:
  - Unroll the loop by a factor of *vector length*.
  - Vectorize all vectorizable operations.
  - Perform full vectorization or no vectorization. Select the option with the highest predicted performance after modulo scheduling.
  - Perform selective vectorization.
- `ignore_comm`: Assume hardware support for communicating operands between vector and scalar instructions. If this option is false, the vectorizer will communicate operands between scalar and vector operations through memory using load and store instructions.
- `vectorize_fp_only`: Only vectorize floating point operations.

The vector length, as well as descriptions of the vector operations are specified in the machine description (MDES) file (Section 12).

6 s2lc: SUIF to Lcode Conversion

`s2lc` is a set of SUIF [10] passes that convert SUIF1 IR to OpenIMPACT Lcode. This functionality is particularly useful if you want to target Fortran sources in Trimaran, or if you want to leverage SUIF's extensive library of optimizations. In particular, the SUIF dependence library is necessary for automatic vectorization, described in Section 5.

6.1 Download and Installation

`s2lc` requires SUIF version 1.3.0.5. It is available from `http://suif.stanford.edu/suif/suif1`. However, we encountered difficulties compiling this distribution on newer systems. Hence, we recommend our own modified version of the SUIF package available from: `http://www.trimaran.org/download.shtml`.

We made minimal changes to the SUIF sources and build system to compile on RedHat Fedora Core 5 and 6 using `gcc` versions 3.2.3, 3.4.6, and 4.0.3. One caveat is that newer Linux systems support `stdargs` exclusively, while SUIF only supports `varargs`. As such, SUIF will not parse many standard header files on newer systems.

6.1.1 Installing SUIF

Download and unpack the SUIF distribution in your desired directory which we will refer to as `<SUIFDIR>`. The following will create a directory `<SUIFDIR>/suifhome` to house SUIF.

```
% cd <SUIFDIR>
% tar xzf suif-1.3.0.5-trimaran.tar.gz
```

Directions to install SUIF are in `<SUIFDIR>/suifhome/src/basesuif/README.basesuif`, although the following recipe was tested on a few different platforms and verified to work:

1. Set the `SUIFHOME` environment variable to point to `<SUIFDIR>/suifhome`.
2. Setup the necessary SUIF environment variables using the follwing in csh or tcsh

   ```
   % eval `$SUIFHOME/setup_suif`
   ```

   or the following equivalent in a bash shell

   ```
   % eval `$SUIFHOME/setup_suif -sh`
   ```

   The command also has the effect of adding the SUIF `scripts` and `bin` directories to your path.
3. Build and install.

   ```
   % cd $SUIFHOME
   % make setup
   % make install
   ```

6.1.2 Installing s2lc

To install `s2lc`, make sure your Trimaran environment variables are properly set (see Section 4), and ensure that `$SUIFHOME` points to your SUIF installation path. Then, fetch, unpack and install `s2lc` as follows:

```
% cd $TRIMARAN_ROOT
% tar xzf s2lc.tar.gz
% cd s2lc/src
% make install
```

The `s2lc` binaries are installed in `$TRIMARAN_ROOT/s2lc/bin`. For additional information, refer to the `README` files in the SUIF and `s2lc` directories.

6.2 Description of s2lc Passes

Converting a C or Fortran program to Lcode is accomplished through a series of SUIF passes with optional optimization passes in between. The `s2lc` package contains a sample Makefile showing the preferred order of the passes. At the very least you must perform the following:

- Parse the source code with `scc`, the SUIF front-end,
- Lower the IR with SUIF's porky pass,
- Convert to three-operand form with `flatten`, and
- Convert to Lcode with `s2lc`.

Most users will also perform several optimization passes along the way. In addition to the core Lcode-conversion functionality, the `s2lc` package contains several optional compiler passes. The passes provided with `s2lc` are as follows:

- **attach_loop_deps** (optional): Utilizes the SUIF dependence library to identify inner loop dependences (loop-independent and loop-carried), and attaches the relevant information as annotations on memory instructions.
- **save_base_syms** (optional): Attaches base address symbols to loads and stores of arrays. This allows later passes to disambiguate memory accesses to different arrays.
- **indvar_opt** (optional): Induction variable identification and address strength reduction.
- **no_memcpys** (usually required): Replaces SUIF memcpy instructions with load/store pairs.
- **pre** (optional): Partial Redundancy Elimination using the Lazy Code Motion algorithm by Knoop, Ruthing, and Steffen [5].
- **flatten**: Converts SUIF IR to three-operand form suitable for `s2lc`.
- **s2lc**: Converts SUIF to Lcode.

6.3 Using `s2lc` with Trimaran

We recommend you add the following to your `$TRIMARAN_ROOT/scripts/envrc[.bash]`. For csh and tcsh:

```
setenv SUIFHOME <SUIFDIR>/suifhome
eval `$SUIFHOME/setup_suif`
set path=($path $TRIMARAN_ROOT/s2lc/bin)
```

or similarly for bash:

```
export SUIFHOME=<SUIFDIR>/suifhome
eval `$SUIFHOME/setup_suif -sh`
export PATH=$PATH:$TRIMARAN_ROOT/s2lc/bin
```

Then use the shell `source` command to properly configure the environment.

For a given input file `<FILE.c>`, the following minimal set of passes generate Lcode that may be further optimized with OpenIMPACT (see Figure 2) or translated directly for input to Elcor.

```
% scc -.spd FILE.c
% porky -Dfcmmas -Darrays -Dmbrs -Darrays -Dblocks -Dfors -Dloops -Difs FILE.spd FILE.low
% no_memcpys FILE.low FILE.nmc
% flatten FILE.nmc FILE.flt
% s2lc FILE.flt FILE.lc
```

The same command sequence will work for Fortran source code. If more optimization is desired, a more extensive set of passes might consist of the following:

```
% scc -.spd FILE.c
% porky -Dblocks FILE.spd FILE.blk
% no_memcpys FILE.blk FILE.nmc
% save_base_syms FILE.nmc FILE.sav
% attach_loop_deps FILE.sav FILE.dep
% porky -Darrays FILE.dep FILE.arr
% indvar_opt FILE.arr FILE.ind
% porky -Dfcmmas -Dmbrs -Darrays -Dfors -Dloops -Difs FILE.ind FILE.low
% porky -fold -const-prop -copy-prop -ucf-opt -iterate FILE.low FILE.opt
% pre FILE.opt FILE.pre
% porky -dead-code FILE.pre FILE.dce
% porky -unused-syms -unused-types FILE.dce FILE.unu
% flatten FILE.unu FILE.flt
% s2lc FILE.flt FILE.lc
```

The passes above include induction variable optimization, constant folding, constant propagation, copy propagation, partial redundancy elimiation, and dead-code elimination. In addtion, `save_base_syms` and `attach_loop_deps` extract memory dependence information that is passed to the Trimaran Elcor compiler. They are necessary if you wish to perform vectorization.

The following will convert the output Lcode to an Elcor compatible format:

```
% Lhpl_pd -Farch=HPL-PD -Fmodel=V1.1-HP -Fphase=1 -i FILE.lc -o FILE.O_tr
```

which can then be vectorized in Elcor by enabling the `do_vectorize` flag either in the Elcor parameter file or via a command line option as shown below:

```
% elcor -i FILE.O_tr -o FILE.el -Fdo_vectorize=yes
```

An example Makefile is available in `$TRIMARAN_ROOT/s2lc/Makefile.sample` for your convenience. Refer to Section 5 to perform automatic vectorization in Elcor.

7 M5elements Cache Simulator

M5elements is a cache simulator that can be optionally used with the Trimaran simulator. It allows Simu to use the memory subsystem of a larger simulator called M5[3].

M5 is a full-system simulator for architecture research. It provides capabilities to simulate a variety of CPU models and memory hierarchies. It features a detailed, event-driven memory system including non-blocking coherent caches and split-transaction buses. A variety of cache configurations and coherence protocols can be modeled. Full details on M5 are available at `http://www.m5sim.org/`.

[3] The name M5elements follows the trend by several popular commercial products that released less feature-rich versions and appended "elements" to the name.

7.1 Installation

M5elements requires the following software packages:

- `Python` (check http://www.python.org/ for the latest version)
- `SCons` (check http://www.scons.org/ for the latest version)

To install M5elements, ensure that the Trimaran environment variables are set according to the instructions in Section 4. Then enter the M5elements directory and run `make`. This step requires an active network connection because it downloads the relevant M5 sources, applies the M5elements patch, and compiles it.

```
% cd $TRIMARAN_ROOT/m5
% make
```

Next, enter the Simu Emulib directory

```
% cd $SIMU_HOME/src/emulib
```

and open the `Makefile` for editing. Change the value of `BUILD_M5E` from `0` to `1` so that it is

```
BUILD_M5E=1
```

then rebuild the emulation library

```
% make
```

At this point the M5elements library is compiled and Simu is properly configured to use it for cache simulation. To disable M5elements at any point, simply toggle `BUILD_M5E` (set to `0`) in the Emulib `Makefile` and rerun `make` to recompile the library.

7.2 Simulating Benchmarks with M5elements

Once M5elements has been installed, just run `tcc` as you would normally. The M5elements library will be linked in automatically.

When the benchmark is simulated, M5elements generates additional statistics to summarize the memory hierarchy performance. The statistics appear in `m5stats.txt` which is located in the `simu_intermediate` directory created for the benchmark.

To configure the memory system used by M5elements, edit the file referred to by the `$M5_CONFIG_FILE` environment variable. You can also change the value of this variable to use a configuration file from another location. The environment variable is read when the benchmark is simulated by Simu. For an explanation of the syntax in this file, please see the M5 website: `http://www.m5sim.org/`.

8 ARM Port

In addition to the traditional HPL-PD output, this version of Trimaran has also been ported to the ARM ISA. In this path, Trimaran produces GNU compatible assembly files that can be assembled/linked using a cross-compiled `gcc`, and then run using a simulator. Development was primarily done targeting SimpleScalar. Quite a bit of effort was spent optimizing this port, and on average it performs within 7% of "`gcc -O3 -fomit-frame-pointer`"[4]. At present, the only known limitation of this port is that the `long long` datatype is not supported.

There are several differences between the main Trimaran path and the ARM port. First, many of the ILP optimizations in OpenIMPACT are either turned off or toned down. ARM only has 15 architecturally visible registers, and ILP optimizations tend to cause a lot of spill code. Secondly, many phases of Elcor are turned off. For example, the machine model expected by the modulo scheduler does not match what the ARM architecture provides, so it is disabled. Lastly, instead of using Codegen to translate from REBEL to C, this path uses `arm-trans` to convert from REBEL to ARM assembly. These files are then compiled and run on a separate simulator.

Below we describe how to set up and run benchmarks using the ARM port. We assume that you are using `gcc` as a cross compiler and SimpleScalar to simulate, though this should be easy to change if desired.

8.1 Installation of the ARM Port

Ensure that your environment is properly set (see Section 4), and that Elcor is already compiled since `arm-trans` requires libraries from Elcor. The first step is to build the REBEL to ARM translator

```
% cd $TRIMARAN_ROOT/arm
% make
```

The second step requires the creation of a cross compiler. We're using an old version of `gcc` because it supports a deprecated floating point standard that is used in SimpleScalar (and consequently by Trimaran). Using a more up to date verion of `gcc` would require upgrading the floating point support in both SimpleScalar and the ARM port.

The following instructions for building a cross-compiler are provided courtesy of the SimpleScalar support team.

1. Make a directory to house the cross compilers and go to that directory. We will refer to the directory as <`CROSSDIR`>.

   ```
   % mkdir <CROSSDIR>
   % cd <CROSSDIR>
   ```

2. Download and unpack the compiler and libraries.

   ```
   % wget http://cccp.eecs.umich.edu/trimaran/binutils-2.10.tar.gz
   % wget http://cccp.eecs.umich.edu/trimaran/gcc-2.95.2.tar.gz
   ```

[4] We know where the inefficiencies are, but most of the low hanging fruit is gone.

```
% wget http://cccp.eecs.umich.edu/trimaran/glibc-2.1.3-armlinux.tar.gz

% tar xzf binutils-2.10.tar.gz
% tar xzf gcc-2.95.2.tar.gz
% tar xzf glibc-2.1.3-armlinux.tar.gz
```

3. Build `binutils`.

```
% cd binutils-2.10
% ./configure --target=arm-linux --prefix=<CROSSDIR>
% make
% make install
% cd ..
```

4. Add <`CROSSDIR`>`/bin` to your executable path. If you are using `csh` or `tcsh` type `rehash` at the command line to rescan the directories in the path for new executables. You do not need to do this if you are using `bash` since it is done automatically.

5. Build GNU `gcc`.

```
% cd ./gcc-2.95.2
% ./configure --target=arm-linux --prefix=<CROSSDIR>
% make LANGUAGES=c
% make LANGUAGES=c install
% cd ..
```

6. Edit `./lib/gcc-lib/arm-linux/2.95.2/specs` as follows: replace all occurrences of "elf32arm" with "armelf_linux". This fixes an incompatibility between `gcc` and GLIBC libraries.

7. Finally, do a `rehash` again so that the shell finds the new executables. You can cross compile a binary that can be run on SimpleScalar/ARM using `arm-linux-gcc -o <binary> <sources...>`. All the other `binutils` should also be available for ARM ELF (e.g., `arm-linux-objdump`, `arm-linux-nm`).

8. Download SimpleScalar ARM. As of this writing, the latest version can be downloaded from http://www.simplescalar.com/v4test.html, or using the following command:

```
% wget http://www.eecs.umich.edu/~taustin/code/arm/simplesim-arm-0.2.tar.gz
```

9. Unpack SimpleScalar and install it.

```
% tar xzf simplesim-arm-0.2.tar.gz
% cd simplesim-4.0
% make config-arm
% make
```

10. Add `simplesim-4.0` to your path so the simulators can be seen by your shell. Everything should be ready to go now.

8.2 Running Benchmarks using the ARM Port

Running benchmarks using the ARM port is nearly identical to the HPL-PD path; simply use `tccarm` instead of `tcc`. This will automatically set the appropriate OpenIMPACT and Elcor switches, run the REBEL-to-ARM translator, assemble/link the assmebly files using `arm-linux-gcc`, and run the benchmark using SimpleScalar. For example, `tccarm -bench wc` will compile and run the `wc` benchmark.

The options of `tccarm` can be seen by running `tccarm -help`. There are two options that are different than the standard `tcc`. First, `-S` is used to pass options directly to the SimpleScalar command line. Second, `-T` is used to select which version of SimpleScalar to use: `sim-safe`, `sim-fast`, or `sim-outorder`. For example: `tccarm -bench wc -Tsim-safe -S‘‘-max:inst 3000’’` will compile `wc` and run it using `sim-safe` for 3000 instructions.

9 Adding a Benchmark

Trimaran provides a number of scripts to make it easier to compile, simulate, verify, and benchmark their applications. The scripts require that benchmarks are packaged in a specific manner. Example benchmark packages are included in `trimaran/benchmarks`, and packages for well known benchmark suites (e.g., SPEC) are availble for download from our website. An example benchmark package includes the following:

- `src/`: a directory containing all the source code
- `input1/`: a directory containing input files needed to run the benchmark
- `output1/`: a corresponding directory containing output files that are necessary to verify the benchmark output when it is compiled with Trimaran
- `compile_info`: native preprocessing and linking options
- `compile_parms`: benchmark-specific Trimaran compiler switches
- `exec_info_input1`: instructions for running and verifying the benchmark output

You can package your own benchmark for use with the Trimaran toolchain. The process is described below although it may be easier to copy one of the existing packages and modifying it for your own purposes.

1. Make sure that you can compile your benchmark with `gcc`. You will need to note all of the related flags necessary to compile the benchmark, as well as input parameters for running it.
2. Create a new directory for the benchmark. As an example, if you are adding an MPEG-4 decoder benchmark called `mpeg4`:

   ```
   % cd $HOME/
   % mkdir mpeg4
   % mkdir mpeg4/src
   ```
3. The Trimaran compiler requires that all source code (including header files) is located in a flat `src` directory. Copy all of the C files and the related header files to `src` directory.

4. The scripts also require information on how to run your benchmark. This is necessary for profiling or simulating your code. If you benchmark requires input files and produces some output that you want to verify against, add them to the benchmark package.

   ```
   % cd mpeg4
   % mkdir input1
   % cp <INPUT-FILES> input1
   % mkdir output1
   % cp <OUTPUT-FILES> output1
   ```

   If you have more than one input-output pair, number the input and output directories as `input1`, `input2`, ..., and similarly for the output (`output1`, `output2`, ...). We recommend at least three input-output pairs: one for testing, one for profiling, and one for benchmarking purposes. You may omit the input directories if your benchmark does not require any input files. Similarly, you can omit the output directories if your benchmark does not produce any meaningful output (i.e., it self verifies).

5. Create a new file called `compile_info` and add it to the new benchmark package directory. This file will contain information required to properly compile your benchmark. For example, it will include preprocessing flags and required libraries, as well as the input workloads to use for profiling (training) and benchmarking (evaluation). The file defines the following six variables, shown here with commonly used settings:

   ```
   LINKING_PREOPTIONS="";
   LINKING_POSTOPTIONS="-lm";
   LIB_REQUIREMENTS="NONE";
   # Specify default training and evaluation inputs
   # Seperate multiple input names by a space (e.g., ‘‘input1 input2’’)
   DEFAULT_TRAIN="input1";
   DEFAULT_EVAL="input2";
   # Optimizing the emulation binary is recommended
   OPTIMIZE_EMUL_EXEC=1
   ```

   For more information on `compile_info` run

   ```
   % read_compile_info --help
   ```

6. Add a new filed called `compile_parms` to the benchmark package directory. This file can customize Trimaran compiler parameters to the particular benchmark. For most benchmarks this file does not override any parameters. For some benchmarks however, the optimizations may need tweaking so that they are more conservative and require less space and time. A standard `compile_parms` is as follows:

   ```
   # No parameters changes are required for this benchmark
   # Use the baseline parameter file for all the parameter settings
   (* $BASELINE_PARMS_FILE$
    end)
   ```

   For more information on the `compile_info` setting run

   ```
   read_compile_info --help
   ```

7. For each input-output pair, you will also need a file that instructs the scripts on how to execute your benchmark. For each pair, create a new file called `exec_info_inputX`, where `X` $= 1, 2, \ldots$, and add it to the benchmark package directory. This file is required for any input workload that you want to use for profiling or benchmarking (e.g., defined in `compile_info`. The file defines the following six variables, shown here with commonly used settings:

```
DESCRIPTION="MPEG-4 decoder"
# Link or copy any required input files to the workspace,
# or execute some required shell commands
SETUP="ln -sf ${BENCH_DIR}/input1/shrek.m2v .";
# Command needed immediately before executable name
PREFIX="";
# Benchmark arguments
ARGS="-b shrek.m2v";
# Commands to check output against reference output
CHECK="diff ${RESULT_FILE} ${BENCH_DIR}/output1/result.out"
# Commands to cleanup the workspace
CLEANUP="rm -f shreak2.m2v";
# Number of instructions to fast forward the simulation
# NOTE: not used by Simu
SKIP="0";
```

For more information on the execution info files, run

```
% read_exec_info --help
```

8. Trimaran looks for user-added benchmarks in four places according to the following environment variables:

```
$USER_BENCH_PATH1
$USER_BENCH_PATH2
$USER_BENCH_PATH3
$USER_BENCH_PATH4
```

You may set any of them to point to your benchmark path. It is recommended that you add the appropriate setting to your `envrc[.bash]` file (see Section 4).

9. You can test your benchmark package with `test_bench_info`. For example:

```
% test_bench_info mpeg4
```

If the test passes, you are ready to use your benchmark with `tcc`:

```
% tcc -bench mpeg4
```

10 Modifying Elcor

The Elcor compiler in Trimaran is highly modular, and is well suited for designing and evaluating new compiler optimizations. The compiler includes a rich suite of analysis passes that you can use to your advantage.

10.1 Code Examples

In order to help you jump into the code, there is a directory in Elcor containing several examples of common things you might want to do in a compiler: `$ELCOR_HOME/src/Examples`. The function `run_examples()`

in `$ELCOR_HOME/src/Examples/example_driver.cpp` contains calls to all of the example functions. To run these examples, set the `do_examples` flag *yes* either by changing it in `$ELCOR_HOME/parms/DRIVER_DEFAULTS` or at the Elcor command line by adding `-Fdo_examples=yes`.

There are six functions called in `run_examples()`, each of which demonstrates a different use. They are outlined below:

- `iterate_edges_example`: demonstrates how to insert dataflow and other styles of dependency edges (e.g., control and memory) into the IR, and how to access that information. Also demonstrates how to walk through all the operations in a function and how to manually adjust edges in the IR.
- `slack_computation_example`: demonstrates how to use the critical path analysis in Elcor, as well as how to recurse down the IR hierarchy.
- `display_cfg_dfg_example`: demonstrates how to create a graphical control flow or dataflow graphs using Dot.
- `add_new_block_example`: demonstrates how to create a new control block, and insert it into the IR.
- `backedge_coalescing_example`: demonstrates how to detect loops, and presents more complicated block creation and control flow restructuring.
- `DU_UD_chains_example`: demonstrates the use def-use analysis.

10.2 Adding Your Own Code

If you plan to add new analsysis techniques or optimizations to Elcor, we recommend you encpasulate your code in its own directory. This helps to preserve the compiler modularity and code organization. Here we outline the process of adding a new Elcor source directory so that it conforms with the compiler and its build system.

1. Create your new directory

   ```
   % cd $ELCOR_HOME/src
   % mkdir MyOpti
   ```

   and move the appropriate files into that directory.

2. Edit `$ELCOR_HOME/include/make_links` to create an entry for `MyOpti` that matches the other directories. For example:

   ```
   if [ -d ../src/MyOpti ] ; then
      echo "Linking files from MyOpti dir"
      ln -s ../src/MyOpti/*.h .;
   fi;
   ```

   This will ensure that all of the header files created in `$ELCOR_HOME/src/MyOpti` are automatically linked into `$ELCOR_HOME/include` when analyzing dependencies with `make depend`. This also enables files in other directories to find the necessary header files during compilation.

3. Create a file `$ELCOR_HOME/src/MyOpti/MyOpti_all.cpp`, and `include` all of the `*.cpp` files found in `$ELCOR_HOME/src/MyOpti`. For example:

```
% cat MyOpti_all.cpp
#include "foo.cpp"
#include "bar.cpp"
#include "baz.cpp"
```

The reason for this file is that it speeds up compilation time. Namely grouping all the source files into a single file avoids repeatedly including the same (large) header files if they are common to many `.cpp` files.

4. Open `$ELCOR_HOME/src/Makefile` for editing. There are a few edits you will need to make. First, add `MyOpti` to the list `ELCOR_SRC_DIRS`. Second, add `MyOpti_all.cpp` to the list `ALL_ADDITIONAL_SOURCES`.

   In some cases you may find it necessary to turn your new code into a library so that it can be reused in other parts of Trimaran, or you may have introduced new data structures that are needed by Simu for Codegen. To create `libMyOpti.a` then we recommend you follow an example from the Makefile; grep for "`scalar`" in the Makefile for a good example to follow.

5. You are all set. Reconstruct the dependences and rebuild Elcor.

```
% cd $ELCOR_HOME/src
% make depend
```

You should see "`Linking files from MyOpti dir`" and "`Processing MyOpti/MyOpti_all.cpp`" scroll by. If that worked, then simply recompile Elcor.

```
% make
```

10.3 Adding a Command Line Parameter

You may find it useful to have a command line argument to enable or disable any code that you add or modify. Here we describe how to add command line parameters to Elcor, although adding parameters to Codegen is very similar. Understand, however, that Elcor (and Codegen) reads all of its parameters from files in the `$ELCOR_HOME/parms` directory. Those parameters may be overridden by command line options. Hence if you add a new command line switch, you should also find the appropriate parameter file and add a corresponding entry there as well.

As an example, to add a boolean parameter `catch_on_fire` as a new *debugging* flag requires the following steps:

1. Declare the variable in Elcor. Since this is a debugging flag, it is best to locate it with similar flags in `$ELCOR_HOME/src/Main/el_debug_init.h`. Edit the file to declare the new variable:

```
extern int El_catch_on_fire;
```

This enables any file that includes `el_init.h` to access this variable. Note that the code that parses the command line arguments is written in C, hence boolean values are generally declared as integers.

2. Next, declare and initialize the variable in `$ELCOR_HOME/src/Main/el_debug_init.cpp`. Then, modify `El_debug_read_parm()` in the same file to read the variable when the Elcor compiler is invoked. In this case, add

```
L_read_parm_b(ppi, "catch_on_fire", &El_catch_on_fire);
```

anywhere in the function. The _b at the end of L_read_parm signifies that this parameter is a boolean variable. Other common read function suffixes are _i for integers, _lf for doubles, and _s for strings (all the parm reading functions can be found in $IMPACT_ROOT/src/library/libparms/l_parms.h.

3. Last, edit $ELCOR_HOME/parms/DEBUG_DEFAULTS and add

   ```
   catch_on_fire = yes;
   ```

   somewhere between (Elcor_Debug declaration and end). The yes setting on this line represents the default value for the parameter if it is not overloaded on the command line.

4. Recompile Elcor and you are all set. To use the parmameter from the command line, add

   ```
   -E"-Fcatch_on_fire={yes|no}"
   ```

   when using tcc or simply -Fcatch_on_fire={yes|no} when using Elcor directly.

The example outlined above assumes the new parameter is added to an existing parameter file. If you wish to create a new parameter file, the process is straightforward: add the file name to $ELCOR_HOME/parms/ELCOR_PARMS, write your own *_init() function in $ELCOR_HOME/src/Main/, and add a hook to your init function in $ELCOR_HOME/src/Main/el_init.cpp.

10.4 Templates

You may notice that Elcor does not use the standard C++ libary (STL). Elcor, which began in November 1993, predates the standardization and widespread availability of the C++ STL. The Elcor STL-like functionality is found in $ELCOR_HOME/src/Tools.

Unlike the STL, Elcor's data structures separate class method declarations and method definitions into different *.h and *.cpp files. Because of this you must explicitly instantiate templated data structures in Elcor, or you will encounter link-time compilation errors of the following sort:

```
Main/main_all.o(.text+0x16a4): In function ‘common_process_function(Procedure*)’:
.../elcor/src/Main/process_function.cpp:131: undefined reference to
    ‘Vector<Codegen>::Vector()’
Main/main_all.o(.text+0x219f):.../elcor/src/Main/process_function.cpp:347:
     undefined reference to ‘Vector<Codegen>::~Vector()’
Main/main_all.o(.text+0x21ee):.../elcor/src/Main/process_function.cpp:347:
     undefined reference to ‘Vector<Codegen>::~Vector()’
collect2: ld returned 1 exit status
make: *** [../bin/elcor] Error 1
```

This error occurs because process_function.cpp has access to the declarations of Vector (from vector.h), but it does not know where the Vector methods are implemented. As a result, gcc cannot properly instantiate the template. In Elcor, all of the templates are instantiated in $ELCOR_HOME/src/Templates. To explicitly instantiate a new template, edit the appropriate file in the template directory as follows:

```
#include "vector.cpp"
#include "codegen.cpp"
template class Vector<Codegen>;
```

This explicitly instantiates the template and resolves the link error.

11 Adding a New Clustering Algorithm

Trimaran implements two instruction clustering algorithms to distribute instructions among the processing clusters in the architecture. The two algorithms are Bottom-Up Greedy (BUG) [2] and Region-Based Hierarchial Operation Partioning (RHOP) [1].

The process to incorporate a new clustering algorithm is straightforward. We faciliate the process by including an example in `elcor/src/Examples/random_cluster.*`. The example implements a random clustering algorithm. It randomly distributes program instructions to clusters in the processor.

There are two main steps. First, create a class that extends `Cluster_algorithm`. In the example, the class `Random_cluster` is created; refer to the following example files for details:

```
elcor/src/Examples/random_cluster.h
elcor/src/Examples/random_cluster.cpp
```

Next, instruct the cluster manager to user the new clustering algorithm. The clustering algorithm is instantiated in `elcor/src/Cluster/cluster.cpp` in function `Cluster_mgr::do_block_clustering()`. For the random clustering algorithm, the following code is added:

```
cluster_alg = new Random_cluster (this);
```

Similar lines of code can be added for other clustering algorithms.

12 MDES: Machine Description

The machine description (MDES) describes the ISA, operand formats, and the microarchitectural resources used by the operations in the target processor. The target processor can be a single/multi-cluster with its ISA derived from the base HPL-PD ISA. Each cluster can be configured to have multiple integer (I), floating (F), memory (M), and branch (B) functional units the corresponding physical register file types. Elcor restricts the number of physical register files per type to one per cluster.

The standard Elcor MDES files are located in `$TRIMARAN_ROOT/elcor/mdes`. The file `hpl_pd_elcor_std.hmdes2` defines the macros that describe the architectural details of the target processor. They include the number of clusters, size of the integer, floating, predicate, control, vector and branch registers (both static and rotating), the number of functional units of each type (I/F/M/B) per cluster, the number of inter-cluster moves allowed per cycle and the functional unit latencies.

The operations in the target processor ISA are defined in the file `hpl_pd_elcor.hmdes2`. This file describes the operations along with the unit specific opcodes associated with each operation and the processor specific flags (under section `Elcor_Operation_Flag`). Notable flags are `Is_comm` which is true for commutative operations, and `Is_unsupported` which flags an Elcor operation that does not have a corresponding version

in the target architecture. Elcoruses the latter substitute unsupported operations with equivalent operations that are supported in the target architecture. For example, the sign extend operation `EXT` is unsupported on ARM, and it is replaced by a logical left shift `SHL` followed by an arithmetic right shift `SHRA`.

The file `hpl_pd_pristine.hmdes2` enumerates the unit specific opcodes corresponding to the target processor opcodes. In addition, it defines macro registers used internally by the compiler.

12.1 HMDES vs. LMDES

The MDES is described using a high level specification language called *Hmdes2* [3]. It has support for macros and control constructs (e.g., if-then, loops), and allows for a compact machine description of the target architecture. The script `$TRIMARAN_ROOT/impact/script/hc` compiles `*.hmdes2` files to a low level MDES specification called LMDES. LMDES is a flat form of the HMDES specification that is used internally by Elcor. Any changes to `*.hmdes2` files require a recompilation to generate the corresponding `*.lmdes2` files. We provide a `Makefile` in `$TRIMARAN_ROOT/elcor/mdes` to automate the process.

12.2 Macro Registers

Macro registers (or simply macros) are compiler assumed registers used for code generation related tasks. Examples macros include the stack pointer, return address registers, parameter passing registers, and loop counters used for modulo scheduling. It is the responsibility of the user to map these registers to appropriate physical registers in the target architecture. As an example, the `Macro` section in `hpl_pd_pristine.hmdes2` lists the standard Elcor macros which are mapped to physical registers in the `Register` section.

The `Macro_Class` defines the following flags that may be attributed to macros:

- `READ_ONLY`: Defines a register to be read only. The register allocator does not register allocate to a read only register.
- `PSEUDO`: Certain macros are used within the compiler to pass information internally and do not have an associated physical register.
- `ALIAS`: This is used to define multiple macros that are all associated with a single physical register. For example, in ARM, the `INT_RETURN` and the `INT_PARAM_1` macros point to the same register.

12.3 Stack Descriptor Section

The stack descriptor describes how local variables and function parameters are laid out in memory. Open-IMPACT performs this layout to a virtual stack, and Elcor (optionally) does the final, physical stack layout. The following MDES parameters tell Elcor how to do the physical layout.

- `Dir`: Whether the stack grows from high addresses down, or from low addresses up.
- `RetAddrThruStack`: Some architectures require the return address (implicitly written on function calls) to be stored on the stack at a specific location, instead of just being caller/callee saved. Setting this

parameter to true saves room on the stack for the return address and inserts the appropriate loads and stores.

- `RetAddrSize`: How many bytes to reserve when RetAddrThruStack is set to true.
- `Alignment`: Many architectures require the top of the stack to be aligned at particular boundaries. This parameter defines what that alignment is; e.g., `Alignment(8)` means the top of the stack must be at an address that's a multiple of 8 bytes.

12.4 Literal Formats

The unit specific opcodes associated with each operation are used to specify the I/O formats and the architecture resources used by each operation. The I/O formats specify the register and literal operands used by each operation. The different literals (or immediates) used by the operations of the target processor are modeled as register files within the `Register` section in the MDES. Each such literal register file has a virtual file type `L`. In addition, the size in bits and the valid set of values allowed by the literals also have to be specified. The valid range of values are specified in the `Constant_Range` and `Constant_Set` sections. Elcor uses the bitwidth and value ranges to generate appropriate moves if the input literal size does not fit within the I/O format of the operation.

12.5 Vector Architectures

Trimaran supports selective vectorization as described in Section 5. The vector capabilities of the target architecture are described in the MDES. The relevant parameters in `hpl_pd_elcor_std.hmdes2` include:

- `vec_length`: This specifies the vector length in number of elements. Currently, all vector registers must have the same length (heterogeneous vector lengths are not supported).
- `vir_static_size` and `vir_rotating_size`: This specifies the number of static and rotating vector integer registers. Each register actually consists of `vec_length` integer elements.
- `vfr_static_size` and `vfr_rotating_size`: This specifies the number of static and rotating vector floating-point registers. Each register actually consists of `vec_length` floating-point elements.
- `vmr_static_size` and `vmr_rotating_size`: These specify vector mask registers. This functionality is currently unimplemented.
- `vec_integer_units`, `vec_integer_perm_units`, `vec_integer_xfr_units`: These specify the number of functional units available for vector computation, vector permutation operations, and vector-scalar transfer operations.

13 Adding a new Opcode

Trimaran is extensible, and it is possible to add new instructions or opcodes to the compiler and simulator. This is useful for a variety of compiler or architecture research (see [7] for an example).

The steps outlined below describe the process of adding a floating point multiply-accumulate (`FMPYADD`) instruction as an example. The instruction has the following format

```
FMPYADD dest, src1, src2, src3 if pred
```

The execution of the operation is guarded by a predicate `pred`. When the predicate is cleared (`pred` = 0), the operation is nullified (i.e., has no effect on the processor state). Otherwise, the instruction has the following semantics:

```
dest = src1 * src2 + src3
```

where the source and destination operands are floating point registers.

There are three steps. First add the instruction to the machine description, then add the instruction to the compiler (Elcor, and optionally OpenIMPACT), and finally add the instruction to the simulator (Simu).

13.1 Adding Opcodes to MDES

Add the new opcode to the machine description files in `$TRIMARAN_ROOT/elcor/mdes`.

1. Add the new opcode to `hpl_pd_ops.hmdes2`:

   ```
   $def OP_floatarith3_floatmpy FMPYADD
   ```

   The description above defines a group of instructions that will share common ISA properties. For example we can also a multiple-substract (`FMPYSUB`) instruction to the group

   ```
   $def OP_floatarith3_floatmpy FMPYADD FMPYSUB
   ```

   since the two instructions have the same I/O format, instruction latency, and scheduling possibilities (scheduling alternatives).

2. Modify `hpl_pd_pristine.hmdes2` to specify the operation format, the resource use patterns and the scheduling alternatives for the new opcode.

   Add the instruction format in section `Operation_Format`:

   ```
   $for (clust in $0..(num_clusters-1)) {
       OF_floatarith3_${clust}(
          pred(FT_p_${clust})
          src(FT_f_${clust} FT_f_${clust} FT_f_${clust})
          dest(FT_f_${clust})
       );
   }
   ```

   In this example, the instruction format is declared to be the same on each of the available clusters, indexed by a loop variable `${clust}`.

   The operation format states that the instruction requires a predicate operand whose file type `FT` is a predicate register (`p`). There are three source operands, each of type `f` for floating point, as well as a single destination operand of the same file type.

   The next step is to add a scheduling alternative for the opcode in `Scheduling_Alternative`:

```
$for (clust in $0.. (num_clusters-1)) {
    $for (idx in $0..(float_units-1)) {
        SA_floatarith3_floatmpy_${clust}_f${idx} (
           format(OF_floatarith3_${clust})
                  latency(OL_floatmpy)
                  resv(RT_${clust}_f${idx})
        );
    }
 }
```

The scheduling alternatives provide information required by the scheduler. There is a scheduling alternative defined for each resource that supports the opcode (i.e., can execute the instruction). In this case, `FMPYADD` instructions can execute on any of the floating point units on any of the clusters.

The scheduling alternative also declares the operation latency `latency(OL_floatmpy)`, and the reservation table entries that are reserved when the instruction is scheduled on the corresponding resources `resv(···)`. The reservation patterns may include functional units, register ports, and operand network wires. The example assumes `FMPYADD` uses the same resources as other floating point operations, so an existing reservation pattern is used. The section `Reservation_Table` defines reservation patterns, and new patterns may be added there if necessary.

Finally the opcode is added to the ISA in the `Operations` section:

```
$for (class in floatarith1_float floatarith1_floatdiv
               floatarith2_float floatarith2_floatdiv floatarith2_floatmpy
               floatarith3_floatmpy) {
    $for (op in ${OP_${class}}) {
        $for (w in ${float_widths}) {
            "${op}_${w}_${clust}.${idx}"(alt(SA_${class}_${clust}_f${idx}));
        }
    }
}
```

Many operations in different classes are added to the ISA by the above code segment. For each operation, the list of scheduling alternatives is bound to the operation. When the HMDES is compiled to LMDES, it will produce the following opcodes:

```
FMPYADD_S_0.0
FMPYADD_S_1.0
......
FMPYADD_D_0.0
FMPYADD_D_1.0
```

These opcodes are known as unit-specific opcodes, and each may appear as a `s_opcode` in the REBEL files. The `S` and `D` attributes in the opcode dictate the width of the operation: single and double precision, respectively. The number preceeding the period specifies the cluster where the operation is scheduled, and the number following the period specifies the functional unit used for the operation.

3. Add the abstract opcode to `hpl_pd_elcor.hmdes2`. These opcodes also appear in the REBEL files, but only describe the functionality of the operation. An Elcor opcode can correspond to many unit-specific opcodes and they are associated with flags that specify the properties of the opcode. The following adds `FMPYADD` as well as a few others to the list of Elcor opcodes.

```
$for (class in floatarith1_float floatarith1_floatdiv floatarith2_float
               floatarith2_floatdiv floatarith2_floatmpy floatarith3_floatmpy) {
    $for (op in ${OP_${class}}) {
        $for (w in ${float_widths}) {
            ${op}_${w}(op($for (clust in $0..(num_clusters-1)) {
                              $for (idx in $0..(float_units-1)) {
                                  "${op}_${w}_${clust}.${idx}"}})
                       flags(ARITHOP FLOAT SPECULATIVE));
        }
    }
}
```

4. Compile the HMDES to LMDES. There is a `Makefile` in the directory for convenience. Type `make` to compile the MDES changes.

13.2 Adding Opcodes to Elcor

Opcodes in Elcor are declared using enumerated types, or in otherwords, the are defined as integer constants. The Elcor opcodes are defined in `$TRIMARAN_ROOT/elcor/src/Graph`. Each opcode has a root opcode (lower 8 bits) and an opcode modifier (upper 24 bits). The modifier is used when there are groups of opcodes with similar functionality (e.g., compare to predicate instruction or `CMPP`).

New opcodes may require new root opcodes. For `FMPYADD` for example, we added a new root opcode `ROOT_FMPYADD` to the `IR_ROOT_OPCODE` enumeration. The single and double precision opcodes are then added to `Opcode` enumeration:

```
FMPYADD_S = ROOT_FMPYADD,
FMPYADD_D = ROOT_FMPYADD | IR_DOUBLE,
```

Next, bind the opcode to a string equivalent (name). This is necessary for generating REBEL files, and for generating error messages. Edit the procedure `el_init_elcor_opcode_maps_arithmetic()` in `$TRIMARAN_ROOT/elcor/src/Graph/opcode.cpp`. For `FMPYADD_S`, we added:

```
el_string_to_opcode_map.bind("FMPYADD_S", FMPYADD_S) ;
el_opcode_to_string_map.bind(FMPYADD_S, "FMPYADD_S") ;
```

to provide a mapping between the opcode integer and string values.

At this point, recompile Elcor. The compiler will recognize the opcode, although the instruction selection cannot use the new opcode unless you instruct the compiler when it should use it. In our example, we will need an analysis phase to find multiply-add instrution chains and replace them with the new opcode. The next section describes an alternate methodology to facilitate the process of retargetting the ISA.

13.3 Adding Opcodes to OpenIMPACT

If you intend to use Elcor to generate Lcode for OpenIMPACT, you will also need to add the opcode to OpenIMPACT. The opcodes in OpenIMPACT are defined in two places: `$TRIMARAN_ROOT/openimpact/src/Lcode/Lcode/l_opc.h` defines the common opcodes that are used by all architectures, and the `$TRIMARAN_ROOT/openimpact/src/machine/Mspec` directory contains opcode files that are specific to particular architectures. For example, the file `m_hpl_pd.h` defines opcodes for the HPL-PD architecture. HPL-PD is the default Trimaran architecture and new opcodes should be added there.

Each new opcode requires an integer and a string value. For `FMPYADDN_S` the following definitions were added

```
#define PLAYDOHop_FMPYADDN_S    1475    /* Lop_MUL_ADD_F */
...
#define PLAYDOHopcode_FMPYADDN_S        "PLAYDOHop_FMPYADDN_S"
```

Lastly, specify the mapping between Elcor and OpenIMPACT opcodes. This is accomplished in `$TRIMARAN_ROOT/elcor/src/Impact/el_opcode_map.cpp`. For `FMPYADD_S`, we added:

```
el_lcode_to_elcor_opcode_map.bind(Lop_MUL_ADD_F,FMPYADD_S);
...
el_elcor_to_lcode_opcode_map.bind(FMPYADD_S,Lop_MUL_ADD_F);
```

to the `el_init_lcode_opcode_arith()` procedure.

At this point, recompile Elcor and OpenIMPACTto complete the compiler edits.

13.4 Adding Opcode to Simu

The final step is to instruct the simulator how to simulate the new instruction. The simulator automatically generates code for each opcode. The opcodes are described in `$TRIMARAN_ROOT/simu/src/emulib/PD_ops.list`. The file consists of several sections. Each section contains a list of opcodes descriptions. The opcodes are grouped according to functionality. For example the section `[float_arith]` groups floating point arithmetic operations. The code generated for the opcodes in that section will appear in a file `PD_float_arith_ops.c`.

There is one line per opcode. The `FMPYADD_S` opcode description appears as follows:

```
FMPYADD_S      ?F,F,F:F;sp      FP Multiply-Accumulate      dest1.S = src1.S * src2.S + src3.S
```

There are four fields in the opcode specification. The fields are seperated by *tabs*. Improper formatting will cause the Emulib parser to fail.

The first field corresponds to the opcode name. The second field indicates if the operation has a predicated equivalent (`?`). It is followed by a comma seperated list of source operand types. In the example, `F,F,F` decribes three source operands of type float. Source operands are seperated from destination operands by a colon. The list of destination operands is terminated by a semicolon. The last part of the field indicates if the operation can be issued speculatively (`sp`).

The third field describes the operation and will appear as a comment in the generated code. The last field describes the operation semantics. The keywords src1, src2, ... refer to the first, second, ... source operands, and similarly dest1, dest2, ..., refer to the destination operands. The .S modifier is used to indicate single precision computation, whereas .D indicates double precision. The code generator automatically generates the appropriate casts.

It is possible to express more complex semantics, including conditionals. If the description becomes unwieldy, you can encapsulate the semantics in a C function and replace the last field by a function call. There are several such examples in PD_ops.list.

Modifications to PD_ops.list take effect when Emulib is compiled. Use the included Makefile to recompile the library. The build process will run gen_functions found in the gen subdirectory. It will parse PD_ops.list and generate the appropriate simulation code. The newly generated code is then compiled and a new emulation library is generated.

The code for FMPYADD_S will appears in two functions:

```
void __PD_FMPY_S_reg_reg(__PD_OP *op);
void __PD_FMPY_S_reg_reg_pred(__PD_OP *op);
```

The first implements the operation semantics, and the second guards the execution of the first according to the predicate operand. The functions will appear in the file PD_float_arith_ops.c. Since the files are automatically generated, you should not modify them. Any changes that are made in the generated files are lost whenever Emulib is compiled.

14 Customized Instructions

The previous section described one way to add an opcode in Trimaran. For non-computation instructions (e.g., loads and branches), this is the only way to add a new opcode. Elcor can however automatically identify acyclic computation graphs and replace them with new opcodes. The benefit of using this method is that the process is entirely MDES driven (meaning Elcor does not have to be recompiled to add instructions), and hence it provides an effective ISA retargetting methodology.

An example acyclic graph used in this example is shown in Figure 4. This graph represents a typical multiply-accumulate instruction. The Elcor pattern matcher uses a description an MDES description of the graph to find all instances that occur in the IR, and replaces them with the new instruction. The particular MDES that describes this graph can be found in $TRIMARAN_ROOT/elcor/mdes/mac_example.hmdes2.

The first few sections in this MDES file simply describe the compiler view of the custom operation to be added (much like in the previous section). Operation_Format describes what operands the new opcode can accept. In this case, there are 3 sources and 1 destination. The Scheduling_Alternative extends that description to include the latency of the new operation, as well as the resources used. Operation defines the name of a particular alternative.

Once the compiler view of the custom instruction is in place, it is necessary to define the graph for the Elcor pattern matcher. The first section, Pattern_Node_Flags, defines some properties of nodes that are declared later. These names are significant and should not be modified. The next section, Pattern_Edge, defines the edges in the graph; notice that each of the 5 edges in Figure 4 is declared in mac_example.hmdes2. The

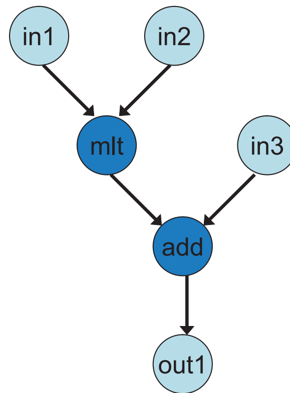

Figure 4: Example acyclic computation graph for multiply-accumulate.

names in this section are not important, and you may choose names that you find appropriate to describe the graph.

Next, the graph nodes are defined in `Pattern_Nodes`. In the example there are 6 nodes to match Figure 4. Inputs to the pattern are marked with the `LIVE_IN` flag, and outputs with `LIVE_OUT`. Nodes declare the opcodes of their corresponding computation. In the example, `add` supports the opcodes `ADD_W` and `ADDL_W`. Each of the opcodes listed in these declarations must appear in the `opcode_to_string_map` defined in `$TRIMARAN_ROOT/elcor/src/Graph/opcode*.cpp`.

Next, the nodes are connected using the previously declared edges. Each node has a `src` and a `dest` that describe where the operands come from or go to. Note that order is important here. The edge for the first source declared for a non-commutative operation (e.g., subtract) is treated differently than the second source in the pattern matcher. Edges can be used in the hyper-edge sense as well (i.e., they can appear in multiple source and destination declarations). This allows the expression of graphs that have a single result consumed by multiple operations.

Once all the nodes are defined, they are collected into a `Customop_Graph` and attached to the `Operation` defined earlier in the `Custom_Operation` section. The name of the custom instruction, in this case `MAC_example` will appear in the IR as the custom opcode.

The new instruction is now fully defined. To instruct the compiler to use it for instruction selection, edit `$TRIMARAN_ROOT/elcor/mdes/hpl_pd_elcor_std.hmdes2`, add

```
$include "mac_example.hmdes2"
```

at the end of the file, and recompile it to generate a new LMDES file:

```
% hc hpl_pd_elcor_std.hmdes2
```

Finally rerun the compiler with the `do_prepass_custom_ops` flag turned on. For example:

```
% tcc -bench fir -E"-Fdo_prepass_custom_ops=yes"
```

Any occurrences of the graph that are found in the application are replaced automatically with the new operation. *Note* however that you must still modify Simu to recognize the new opcode, as was described in the previous section.

15 Troubleshooting

The following are some common error messages you might encounter when using Trimaran, along with suggested remedies.

- Compiler Build Errors
  - **make error: Undefined reference to Vector, List, Hash_set_filterator, etc.**
    `undefined reference to ‘Vector<Codegen>::Vector()’`
    This error usually indicates an issue with your template declarations. Refer to Section 10.4 for details.

  - **make error: Include file not found** (e.g. any `.h` file)
    Your shell environment may not be properly set up. Refer to Section 4 for details.
- MDES Errors
  - **make error: Command not found** (e.g. `/scripts/hc`)
    Your shell environment may not be properly set up. Refer to Section 4 for details.

  - **My MDES changes don't seem to be taking effect.**
    Whenever you update a high-level MDES file (`.hmdes2`), you must recompile it (using `hc`) to generate a new `.lmdes2` file. See Section 12.1 for details.
- Simulation Errors
  - **tcc error: Result Check ***FAILED*****

    This means that the output of the benchmark did not match the expected output. Chances are the benchmark was compiled incorrectly. This is usually one of the trickier errors to debug; see Section 15.1 for helpful tips.

    You can examine the benchmark workspace files in `simu_intermediate/result.err` for more information. This will be useful in the event Simu crashed unexpectedly. If there is no information here, then it's just a result check mismatch.

    Another possibility, which usually only occurs if you packaged your own benchmark, is that the "expected" output is wrong. Refer to Section 9 to ensure the benchmark is setup correctly. The `test_bench_info` script may help.

  - **tcc error: M5_CONFIG_FILE does not exist**
    Simu was compiled with M5elements linked in, but the environment variable `M5_CONFIG_FILE` did not point to an M5 configuration file.

15.1 Benchmark Debugging Tips

On rare occasions, by chances of ill luck, one might encounter the dreaded **Result Check FAILED** message at the end of a benchmark execution. There are two ways to help pinpoint where the problem is occuring:

- Narrow the space by identifying **which source file** in the benchmark led to failure. See Section 15.1.1.
- Narrow the space by identifying **which compiler transformation/pass** led to failure. See Section 15.1.2.

15.1.1 Simulating Each Source File Individually

Since each benchmark is comprised of one or more source C files, we can compile the files individually and determine which one caused failure. A script is provided that automatically compiles the entire benchmark using the host machine's compiler (e.g. `gcc`), and substitutes one Trimaran-compiled file into the benchmark at a time. The combined executable is run partially through Simu, and partially natively.

The script is found at `$TRIMARAN_ROOT/scripts/replace_good.pl`. Instructions are provided in the comments at the beginning of the script.

15.1.2 Turning Off Elcor Passes

Another way to pinpoint what caused benchmark failure is to turn off various passes in Trimaran and rerun `tcc` to see if simulation succeeds.

The Trimaran flags referenced below appear in `elcor/parms/DRIVER_DEFAULTS`, unless they are noted to control Simu, in which case they appear in `simu/parms/SIMULATOR_DEFAULTS`. The following tips are listed in an order that is convenient to follow for debugging, but it is not a strict guideline.

Locate problem: First thing is to locate which part of the compiler is responsible for the incorrect execution. The three most important parts are 1. OpenIMPACT, 2. Elcor, and 3. Simu. One can skim through the output generated by the execution of `tcc` and get hints from it.

Elcor processing: There are many complex optimizations within Elcor that can lead to benchmark failure if something goes wrong. So, as a first measure, one should disable all processing within Elcor by turning on the `do_null_processing_for_simu` flag. For the resulting code to simulate properly, you should also turn on the flags `emulate_unscheduled` and `emulate_virtual_regs` in Simu. If this action makes things work, i.e. the benchmark runs successfully, then the problem is definitely within Elcor. More steps (as discussed by the following techniques) can be taken in such a situation.

Elcor modulo scheduling: Modulo scheduling [8] is an advanced optimization for loops. If incorrect, it can lead to benchmark failure. Try disabling modulo scheduling by turning off the `do_modulo_scheduling` flag in Elcor.

Elcor postpass scheduling: To turn off postpass scheduling, set `do_postpass_scalar_scheduling` in Elcor. To ensure proper simulation, you will also need to turn on the `emulate_unscheduled` flag in Simu.

Elcor register allocation: Another simplification that can be applied is the compilation of code *without* register allocation. Elcor does not perform register allocation when `do_scalar_regalloc` is turned off.

If you suspect a register allocation bug, turn it off in Elcor, and instruct Simu to simulate the benchmark with virtual registers. This is done by turning on the `emulate_virtual_regs` flag in Simu.

Note that when register allocation is performed, you must also perform postpass scheduling to schedule any spill code. This is accomplished by enabling `do_postpass_scalar_scheduling`.

Elcor clustering: Some architecture configurations can cause unexpected failures during compilation. One example is a multi-clustered architecture. This and other architecture attributes can be changed by modifying the appropriate MDES file (e.g., `elcor/mdes/hpl_pd_elcor_std.hmdes2`). In general it is a good idea to test if the single-cluster configuration passes compilation. Setting the number of clusters to 1 disables the instruction clustering algorithms.

Elcor prepass scheduling: Prepass scheduling occurs before register allocation; turning it off is very similar to turning off postpass scheduling. To turn off prepass scheduling, set `do_prepass_scalar_scheduling` in Elcor. To ensure proper simulation, you will also need to turn on the `emulate_unscheduled` flag in Simu.

Elcor optimization: It is possible to disable classic code optimizations from Elcor by toggling the `do_classic_opti` flag. This can be used to target bugs resulting from optimizations steps within Elcor. Finer management of optimization flags can be done through switches defined in `elcor/parms/OPTI_DEFAULTS`.

16 Help! My question wasn't answered here.

If you have other questions or problems regarding Trimaran, please visit our webage http://www.trimaran.org and sign up for our email list. Announcements of bug fixes and periodic updates are posted on this list, so if you plan on using Trimaran, it is a good idea to sign up.

Acknowledgements

Trimaran is the result of many person-years of research and development. It began as a collaborative effort between the Compiler and Architecture Research (CAR) Group at Hewlett Packard Laboratories, the IMPACT Research Group at the University of Illinois, and the Center for Research on Embedded Systems and Technology (CREST) at the Georgia Institute of Technology. CREST was the ReaCT-ILP Laboratory at New York University. We thank the contributors to Trimaran:

- **George Washington University** : Ajay Jayaraj, Yogesh Chobe
- **Georgia Institute of Technology (CREST: Center for Research on Embedded Systems and Technology)** : Lakshmi Chakrapani, Mongkol Ekpanyapong, Krishna Palem, Weng-fai Wong
- **HP Laboratories (CAR: Compiler and Architecture Research Group)** : Santosh Abraham, Sadun Anik, Alex Eichenberger, Shail Aditya Gupta, Matt Jennings, Richard Johnson, Joel Jones, Vinod Kathail, Matthai Philipose, Bob Rau, Sumedh Sathaye, Mike Schlansker, Robert Schreiber, Greg Snider
- **Massachusetts Institute of Technology** : Mark Hampton, Sam Larsen, Rodric Rabbah

- **New York University (ReaCT-ILP Laboratory)** : Robert Dewar, Ben Goldberg, Hansoo Kim, Amit Nene, Igor Petchansky, Suren Talla, Sam Tregar
- **University of Illinois (IMPACT Research Group)** : David August, Roger Bringmann, Pohua Chang, William Chen, Ben-Chung Cheng, Dan Conners, Kevin Crozier, Brian Deitrich, David Gallagher, John Gyllenhaal, Grant Haab, Richard Hank, Hillery Hunter, Sabrina Hwu, Wen-mei Hwu, Teresa Johnson, Robert Kidd, Hong-Seok Kim, Dan Lavery, Eric Nystrom, Shane Ryoo, John Sias, Sain-Zee Ueng, Nancy Warter, Le-Chun Wu
- **University of Michigan (CCCP Research Group)** : Aaron Erlandson, Jason Blome, Mike Chu, Nate Clark, Ganesh Dasika, Kevin Fan, Shuguang Feng, Shantanu Gupta, Amir Hormati, Manjunath Kudlur, Steve Lieberman, Yuan Lin, Scott Mahlke, Robert Mullenix, Pracheeti Nagarkar, Hyunchul Park, Rajiv Ravindran, Mikhail Smelyanskiy, Hongtao Zhong

We are most grateful for the support from ARM, DARPA, NSF, HP Labs, and Yamacraw. Their support over the last several years has made it possible for us to continue to develop and maintain Trimaran, and make it available to the community for research and education.

References

[1] M. Chu, K. Fan, and S. Mahlke. Region-based hierarchical operation partitioning for multicluster processors. pages 300–311, June 2003.

[2] J. Ellis. *Bulldog: A Compiler for VLIW Architectures*. MIT Press, Cambridge, MA, 1985.

[3] J. Gyllenhaal, W. Hwu, and B. R. Rau. HMDES version 2.0 specification. Technical Report IMPACT-96-3, University of Illinois, Urbana-Champaign, 1996.

[4] V. Kathail, M. Schlansker, and B. Rau. HPL-PD architecture specification: Version 1.1. Technical Report HPL-93-80(R.1), Feb. 2000.

[5] J. Knoop, O. Ruething, and B. Steffen. Lazy Code Motion. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, volume 27, pages 224–234, San Francisco, CA, June 1992.

[6] S. Larsen. *Compilation Techniques for Short-Vector Instructions*. PhD thesis, Massachusetts Institute of Technology, April 2006.

[7] R. Rabbah, H. Sandanagobalane, M. Ekpanyapong, and W.-F. Wong. Compiler orchestrated prefetching via speculation and predication. In *Proceedings of the the 11th international conference on Architectural Support for Programming Languages and Operating Systems*, pages 189–198, 2004.

[8] B. R. Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *27th Annual International Symposium on Microarchitecture*, pages 63–74, San Jose, California, Nov. 1994.

[9] D. Seal. *ARM Architecture Reference Manual*. Addison-Wesley, London, UK, 2000.

[10] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy. SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers. *ACM SIGPLAN Notices*, 29(12):31–37, December 1994.