Chronos Version 2.0 User Manual

Xianfeng Li[1] Yun Liang[2] Tulika Mitra[2] Abhik Roychoudhury[2]

[1]Department of Computer Science and Technology, Peking University
lixianfeng@mprc.pku.edu.cn

[2]Department of Computer Science, National University of Singapore
{liangyun,tulika,abhik}@comp.nus.edu.sg

Abstract

This user manual describes the *second release* of Chronos, a prototype tool that performs Worst Case Execution Time (WCET) analysis of embedded software. Compared to most other WCET tools, Chronos models the timing effects of more contemporary micro-architectures on the execution time of a program. Hence it produces tighter WCET estimates. This manual contains an overview of WCET analysis, a brief introduction of the techniques employed, a detailed guide for installation and usage, and a description of the internals of the tool including its structure and major components.

1 Overview

Estimating the Worst Case Execution Time (WCET) of a program is an important problem. WCET analysis computes an upper bound on the program's execution time on a particular processor for all possible inputs. The immediate motivation of this problem lies in the design of real-time embedded systems. Typically an embedded system contains processor(s) running specific application programs and communicating with the external environment in a timely fashion. Many embedded systems are safety critical, e.g., automobiles and power plant applications. The designers of such embedded systems must ensure that all the real-time constraints are satisfied. Real-time constraints impose hard deadlines on the execution time of embedded software. WCET analysis of the program can guarantee that these deadlines are met. Due to its inherent importance in embedded system design, timing analysis of embedded software has been extensively studied. Accurate timing analysis critically depends on modeling the effects of the underlying micro-architecture. Ignoring the micro-architecture can produce extremely pessimistic time bounds. This is particularly so because modern processors employ advanced micro-architectural features such as pipelining, caching, and speculative execution to speed up program execution. Therefore, to obtain safe but tight WCET estimate of a program, we need to model the complex timing effects of pipelining, caching and branch prediction.

Chronos[1] is a WCET analysis tool that incorporates modeling of micro-architectural features present in modern processors. It models both in-order and out-of-order pipelines, instruction caches and local/global branch prediction. Data caches are not modeled. For each of the modeled features, the modeling is parameterizable, e.g., we can set the pipeline re-order buffer size, size and organization of the instruction cache, and the branch prediction scheme (as well as size of associated data structures such as branch history register).

Chronos is built on top of the SimpleScalar simulator. SimpleScalar is a popular cycle-accurate architectural simulator that allows the user to model different processors in software [2]. We target our analyzer to processor models supported by SimpleScalar. Thus the user can compare

[1]The name is taken from ancient Greek mythology where Chronos was the personification of time

the estimated WCET produced by Chronos against an observed result (lower than the actual WCET) produced by running the program on the same processor model with SimpleScalar's cycle-accurate simulator. The comparison will indicate the tightness of the estimated WCET.

The rest of this manual is organized as follows. Section 2 provides a download and installation guide for the tool. Section 3 briefly describes the techniques behind the tool to give the user a better idea about the functionality of the tool. Section 4 gives a detailed guide on using the tool. Section 5 describes the major components of the tool and their mapping to the internals of the analysis engine. Thus, if the user wants to inspect and modify the source code of the tool, this section will help him/her understand its structure and locate the main components of the tool. Finally, Section 6 presents a summary and pointers to possible improvements in the future releases of Chronos.

2 Download and Installation

2.1 Download

Chronos is targeted to Unix-like platforms and currently it has been tested on GNU/Linux. It can be downloaded from

`http://www.comp.nus.edu.sg/~rpembed/chronos/download.html`

where the following packages are provided:

- **chronos-2.0.tgz**: It contains the source code of the WCET analysis engine, a graphical frontend, and a set of benchmark programs. Given a benchmark program and a processor model, the analysis engine generates an Integer Linear Programming (ILP) problem that can be solved to obtain the WCET estimate.
- **lp_solve_5.5.tgz**: It is a free linear programming (LP) solver [1]. We use to obtain the estimated WCET by solving the ILP problem generated by the analysis engine.
- **simplesim-3.0.tgz**: This is a modified source code of SimpleScalar simulator. We provide it here because currently Chronos models a somewhat simplified version of the SimpleScalar architecture. Unfortunately, some of these simplifications cannot be specified through the processor parameters. In addition, the execution time statistics dumped out by SimpleScalar does not meet our requirements as it includes the execution time of library functions, while our estimation considers only the user program. Thus, we made some modifications to SimpleScalar source code and provide the modified package here.
- **gcc-2.7.2.3.tgz**: This is a modified version of the SimpleScalar distribution of GCC compiler, which is used to compile the benchmark programs into binaries of SimpleScalar ISA (as mentioned earlier, our tool is based on SimpleScalar and it takes the same binaries recognized by SimpleScalar simulators). We made minimal changes to the original SimpleScalar GCC distribution to disable the insertion of an unnecessary library function "_main()" into the compiled code because our analyzer does not consider library functions.

In addition, you need to download two utilities for SimpleScalar compiled binaries from

`http://www.simplescalar.com`

- **simpleutils-990811.tar.gz**: It contains GNU binutils source retargeted to SimpleScalar architecture. These utilities help GCC compile benchmark sources into SimpleScalar binaries.
- **simpletools-2v0.tgz**: It contains GCC compiler and library sources needed to build SimpleScalar binaries. However, as we have provided a modified version of the GCC compiler from our website, the compiler contained in this package is not used. Thus after this package is untared, the directory `gcc-2.6.3` can be removed.

2.2 Installation

Assuming all downloaded packages are put in a directory $IDIR (the recommended directory is $HOME/chronos; you can set an environment variable $IDIR by typing
`export IDIR=$HOME/chronos` on the command line), enter $IDIR and proceed to the installation as described below.

The analyzer First untar the package `chronos-2.0.tgz` by invoking

```
tar -xvzf chronos-2.0.tgz
```

This yields the following sub-directories: `est` – the analysis engine; `gui` – the frontend; `benchmarks` – the benchmark programs coming with this release (a subset of the benchmarks from the WCET research group at Mälardelaen University [6]). Next you need to compile the analysis engine. The analysis engine is written in ANSI C. To compile it type:

```
cd est && make
```

This will produce an executable `est` in the directory `est`, which is the WCET analyzer.

The frontend is written in Java. You need to install JRE (Java Runtime Environment) to invoke the command `java`. Please download JRE 5.0 from

```
http://java.sun.com/j2se/1.5.0/download.jsp
```

After downloading `jre-1_5_0_<version>-linux-586.bin`, run the commands:

```
chmod +x jre-1_5_0_<version>-linux-586.bin
./jre-1_5_0_<version>-linux-586.bin
```

After you install JRE, add the bin directory of JRE to the system path. Make sure you can invoke `java` from the command line and the java you invoke is the one that you just installed. You can check the java version by running the command `java -version`.

The frontend also contains some C code to disassemble binary executables for data flow analysis and relate source code with the assemble code. To compile it type:

```
cd gui && make
```

This will produce an executable `dis` in the directory `gui`.

The LP solver Extracting `lp_solve_5.5.tgz` by invoking

```
tar -xvzf lp_solve_5.5.tgz
```

This will produce three packages.

- `lp_solve_5.5_exe.tar.gz`: The pre-compiled binaries for Linux platform. Extracting this package will produce a directory `lp_solve` which contains the executable `lp_solve` and a library `libxli_CPLEX.so`. You will need to tell Chronos via its user interface where the `lp_solve` directory is. As the pre-compiled binaries work on our machine (Intel Pentium 4 platform running Redhat Fedora Core 3/4), it is unlikely that you will need to compile lp_solve yourself with the following two packages.
- `lp_solve_5.5_source.tar.gz`: lp_solve program source. Compiling it will produce the executable `lp_solve`.
- `lp_solve_5.5_xli_CPLEX_source.tar.gz`: CPLEX XLI reader/writer source. Compiling it will produce the library `libxli_CPLEX.so`, which is needed for lp_solve to read and solve files in CPLEX format.

The three packages above are obtained from lp_solve website

```
http://groups.yahoo.com/group/lp_solve/files/Version5.5/
```

For the user's convenience (you need Yahoo! id and have to join the lp_solve group), we distribute them as a single package, `lp_solve_5.5.tgz`, on our website. For more information about lp_solve, please visit its official website.

SimpleScalar simulators To install the SimpleScalar simulators, first untar `simplesim-3.0.tgz`, which yields a sub-directory `simplesim-3.0`. Then invoke the following commands to compile it:

```
cd simplesim-3.0 && make
```

SimpleScalar GCC and related binary utilities. First we need to install the binary utilities which will be used for compiling SimpleScalar cross GCC. After that we can install SimpleScalar GCC. The steps are described as follows[2]

1. Build SimpleScalar binary utilities:

   ```
   tar -xvzf simpleutils-990811.tar.gz
   tar -xvzf simpletools-2v0.tar.gz
   ```

   then go to the sub-directory `simpleutils-990811`

   ```
   ./configure --host=i386-*-linux --target=sslittle-na-sstrix
   --with-gnu-as --with-gnu-ld --prefix=$IDIR
   ```

   Note that the above commands are only applicable to Intel/x86 platform running Linux. You may need to change some of the parameters on a differnt platform. Please refer to SimpleScalar user guide [2] for more details.

   ```
   make all install
   ```

   This will produce utilities installed under `$IDIR/sslittle-na-sstrix/bin`

2. Build SimpleScalar GCC:

   ```
   export PATH=$PATH:$IDIR/sslittle-na-sstrix/bin
   ```

   This is very important! It ensures the following cross compilation will search the right locations for the utility binaries targeted to SimpleScalar, instead of using the native GNU utilities.

   ```
   tar -xvzf gcc-2.7.2.3.tgz
   cd $IDIR/gcc-2.7.2.3

   ./configure --host=i386-*-linux --target=sslittle-na-sstrix
   --with-gnu-as --with-gnu-ld --prefix=$IDIR --enable-languages=c

   make
   make install
   ```

   Again, parameters in above commands need to be changed if the platform is not Intel X86/Linux.

[2]This part is based on the official SimpleScalar user guide and Pan Yu's enhanced SimpleScalar installation guide at `http://www.comp.nus.edu.sg/~panyu/simplesim.htm`

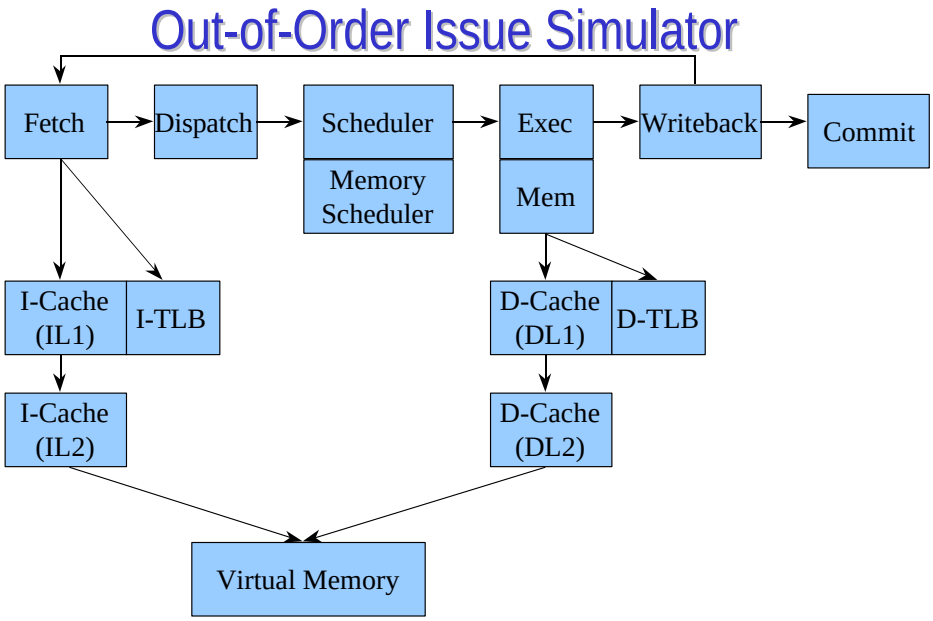

Figure 1: SimpleScalar Out-of-Order Simulator (taken from SimpleScalar hacker's guide)

3 WCET Analysis Technique

3.1 SimpleScalar Processor Model

We first describe the SimpleScalar processor model and point out what features are modeled, what are not modeled and what are modeled with restrictions.

Figure 1 shows the block diagram of SimpleScalar out-of-order simulator containing the major components of the processor model. Note the branch predictor, which enables speculative execution across most of the pipeline stages is not shown in this diagram. The pipeline consists of five stages.

1. **Instruction Fetch (IF)**. In this stage, a sequence of instructions are fetched from the instruction cache or memory into an multi-entry buffer called *dispatch queue*. The number of instructions that are fetched is determined by the following factors: (1) the fetch width of the processor; (2) the available entries in the dispatch queue; (3) cache line size, whether a cache miss arises, and the location. These parameters, as well as most of the parameters mentioned in the description of subsequent stages, can be configured at runtime. Note that the location to fetch the instructions is determined by the program counter and the branch predictor.

2. **Instruction Decode & Dispatch (ID)**. In this stage, instructions in the dispatch queue are decoded and placed into another buffer called the RUU (Register Update Unit), which functions as both a register renaming unit and a re-order buffer. The number of instructions that are decoded is determined by the following factors: (1) the decode width; (2) the available entries in both the dispatch queue and the RUU. In this stage, the scheduler also tracks data dependencies and decides which instructions can be issued for execution and puts the issuable ones into a *ready queue*.

3. **Instruction Execute (EX)**. In this stage, the ready queue is walked through and instructions in the queue are issued for execution if the corresponding function units are available. In addition to the number of instructions in the ready queue and the availability of functional units, the number of instructions that can be issued is also limited by the issue width. For

load and store instructions, this stage is used to compute their memory access addresses. The actual read and write operations take place in the write-back and commit stage respectively. Note that instructions can proceed through this stage out-of-order, e.g., a later instruction whose data operands are ready and functional unit is available can bypass an earlier instruction which is waiting for its operands.

4. **Write Back (WB)**. In this stage, load instructions dispatch the addresses computed in the EX stage to the memory hierarchy, which may involve accesses to the level-one data cache, TLB, level-two data cache and the main memory (as indicated in Figure 1), depending on how the memory hierarchy is configured. The loaded data is forwarded to dependent instructions, if any, in the RUU. If all the operands of a waiting instructions become ready, the instruction will be put into the ready queue by the scheduler. In this stage, the results of ALU operations are written back into the register file. Branch outcome is resolved in this stage and the pipeline is flushed if the prediction turns out to be a misprediction.

5. **Commit (CM)**. This is the last stage where instructions having completed execution are committed in program order. This means no instructions can be committed if the oldest has not completed execution yet. The number of committed instructions, similar to the earlier stages, is also constrained by a parameter called the commit width.

For more details about SimpleScalar architecture, please refer to its user guide [2] and hack guide[3]. In this release, our processor model has the following simplifications to SimpleScalar architecture.

- **Processor core**: We do not model write-back width, i.e., in the write-back stage, every instruction coming out of the EX stage in the previous cycle can write back into the register file.

- **Memory hierarchy**: Data cache is not modeled and each load operation is assumed to complete in a single clock cycle; TLBs are not modeled and we assume the virtual address is identical to the physical memory address; only level-one instruction cache is modeled.

- **Branch prediction**: Not all branch prediction schemes supported by SimpleScalar are modeled in this release. Primarily the following schemes are supported: *perfect prediction*, where each branch is assumed to be predicted correctly; *two-level prediction schemes* including gshare, GAg, and local prediction. These two-level prediction schemes can be uniformly specified by a group of parameters described subsequently. A simplification is that only a single bit, rather than the popular two-bit saturation counter, is used and updated for each prediction table entry. In addition, we assume a perfect Branch Target Buffer (BTB) such that the branch target address is available at the end of the IF stage.

Table 1 gives the default configuration of our processor model (parameters that are supported by SimpleScalar processor model but do not appear here are not modeled in Chronos). Note that the lines starting with '#' are comment lines explaining what these parameters are about. The format is completely compatible with SimpleScalar's parameter format. Thus for further understanding of these parameters, please refer to the SimpleScalar user guide.

3.2 Analysis Technique

Now we give a brief description of the static analysis technique employed in Chronos. Worst-case Execution Time (WCET) analysis of a program usually involves two major steps — **program path analysis** and **micro-architecture modeling**. Micro-architecture modeling captures the timing effects of performance enhancing micro-architectural features. This step is usually used to return the WCET estimate of each basic block in the program's control flow graph. Note that the estimate for a basic block B [4] should be an upper bound on the execution time of B for

[3] http://www.simplescalar.com/docs/hack_guide_v2.pdf

[4] a sequence of instructions such that control flow can only arrive at its head and exit from its tail.

Parameter	Default Configuration
# instruction dispatch queue size -fetch:ifqsize	4
# register update unit (RUU) size -ruu:size	8
# decode width -decode:width	1
# commit width -commit:width	1
# branch predictor type -bpred	2lev
# 2-level predictor config -bpred:2lev	<l1size> <l2size> <hist_size> <xor> 1 128 2 1
# l1 inst cache config -cache:il1	<config>—none il1:16:32:2:l
# memory access latency -mem:lat	<first_chunk> <inter_chunk> 30 2

Table 1: Default Processor Configuration

all possible execution contexts. How do we combine the WCET estimates of basic blocks to get the WCET estimate of the program? This is achieved by program path analysis. Usually most state-of-the-art WCET tools use Integer Linear Programming (ILP) formulation for this step. Formally, let $\mathcal{B}$ be the set of basic blocks of a program. Then, the program's WCET is given by the following objective function

$$maximize \sum_{B \in \mathcal{B}} N_B * c_B$$

where N_B is an ILP variable denoting the execution count of basic block B and c_B is a constant denoting the WCET estimate of basic block B. The linear constraints on N_B are developed from the flow equations based on the control flow graph (CFG). Thus for basic block B

$$\sum_{B' \to B} E_{B' \to B} = N_B = \sum_{B \to B''} E_{B \to B''}$$

where $E_{B' \to B}$ ($E_{B \to B''}$) is an ILP variable denoting the number of times control flows through the control flow graph edge $B' \to B$ ($B \to B''$).

Additional linear constraints are also provided to capture loop-bounds and any known infeasible path information. Such constraints can either be derived automatically via data flow analysis of the program or be provided by the user. In most cases, the user cannot completely rely on data flow analysis. Therefore we specify the format of the ILP constraints the user has to conform with. We first use a concrete example for illustration.

Figure 2(a) gives the C source code of a simple program – `insertsort`, which sorts an array $a[\,]$ of ten elements ($a[0] = 0$ is a dummy one not being sorted). The control flow graph extracted from the binary executable of `insertsort` is shown in Figure 2(b) (this file can be generated by the analyzer `est` with an option "-run CFG"). Note that the first column corresponds to the basic block numbers and the second column gives the starting address (in hexadecimal) of the corresponding basic block. The numbers within brackets indicate the successor basic blocks.

The program has two nested loops. To get an estimation, the loop bounds must be provided to the analyzer. It is obvious that the outer loop iterates nine times ($i = 2$ to $i = 10$). In fact, Chronos is able to derive such simple loop bounds with a light-weight data flow analysis. The inner loop iterates variable number of times for each invocation. But we know that for insertion sort of an array of ten elements, the overall number of iterations of the inner loop should not exceed 45. This loop bound is provide by the user via Chronos interface. These constraints are stored in a file named "`insertsort.cons`" (note that the analyzer always tries to locate the user constraint file with the same name of the benchmark and a suffix ".cons"), which is shown in Figure 2(c).

The linear constraints in above example are in the ILOG/CPLEX format, which can be described more formally as follows.

$$\begin{aligned}
\langle constraint \rangle &::= \langle terms \rangle \sqcup \langle rel_op \rangle \sqcup \langle digits \rangle \\
\langle terms \rangle &::= \langle coeff \rangle \sqcup c\langle digits \rangle.\langle digits \rangle \;\mid\; \langle terms \rangle \sqcup \langle arith_op \rangle \sqcup \langle terms \rangle \\
\langle rel_op \rangle &::= = \;\mid\; < \;\mid\; <= \\
\langle digits \rangle &::= [0-9] \;\mid\; [1-9]\langle digits \rangle \\
\langle coeff \rangle &::= \langle digits \rangle \\
\langle arith_op \rangle &::= + \;\mid\; -
\end{aligned}$$

The symbol "$\sqcup$" denotes a white space. Note that the constraints will be fed into the linear programming solver. So you cannot expect it to precisely point out the error source if you give incorrect constraints. Here we remind the user of a few points that are likely to cause problems:

- Each term denoting a basic block's execution count always starts with the character 'c' followed by the procedure number (the order in which the procedure appears in the program),

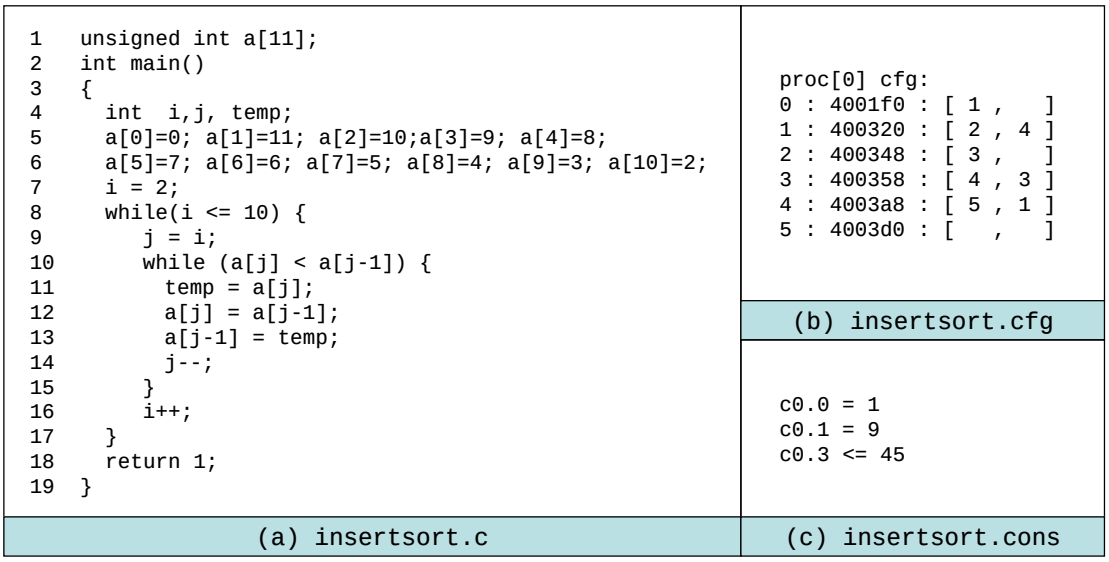

Figure 2: An Example Program, its Control Flow Graph, and User Constraints

a dot symbol, and a basic block number (the order in which the basic block appears in the procedure). Even if there is only a single procedure in the program, the procedure number should not be omitted.

- Terms denoting basic blocks can only appear on the left-hand side of the relation operator (=, <, <=) whereas a constant (a string of digits) can only appear on the right-hand side of the relation operator. For example, "$c0.2 - c0.3 < 10$" is a valid constraint, but "$c0.2 < c0.3 + 10$" is not, neither is "$c0.2 - c0.3 - 10 < 0$".
- A space must be placed between the coefficient and the basic block id in a term. But the coefficient can be omitted if it is "1". Similarly, there should be spaces preceding and succeeding each arithmetic operator (+, −) and each relation operator (=, <, <=). For example, "$c1.5 - 10\ c1.6 = 0$" is a valid constraint, but "$c1.5 - 10c1.6 = 0$" is invalid, as there is no space between 10 and $c1.6$.
- All numbers should be integers.

A good news is that the analyzer comes with a frontend that allows the user to specify constraints on the source code through a dialog box (illustrated in Section 4). Thus in most cases the user does not need to look into the control flow graph and specify constraints in terms of basic blocks. Take "$c0.3 <= 45$" in Figure 2 for example. The user can specify an equivalent constraint "$line10 <= 45$" at the source code level, where $line10$ denotes the execution count of line 10 of the source code. The analyzer is responsible for converting the source-level constraints into basic block level constraints.

4 Using the Analyzer

This section provides the detailed guidance on using the analyzer. As described earlier, the WCET analysis engine interacts with the user through a graphical frontend.

Launch Chronos

Go to the directory `$IDIR/chronos/gui` containing the file `gui.jar` and launch the tool by executing runnable `gui.sh` in the terminal.

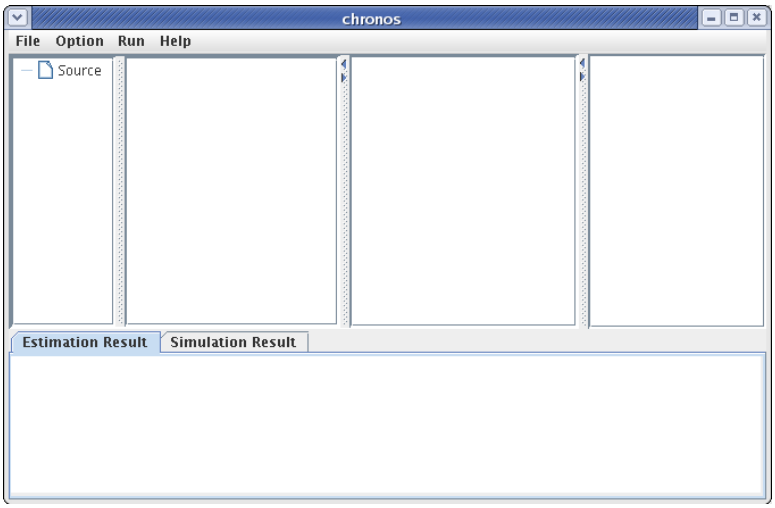

Figure 3: Launching Chronos

The main window of Chronos will appear as in Figure 3. Next, we will use an example benchmark program **insertsort** for discussion.

Perform Analysis

There are a few steps in analyzing the WCET of `insertsort`:

- Set SimpleScalar GCC bin, lp_solve, simplesim-3.0 directory.
- Open a benchmark directory.
- Set loop bounds(if necessary).
- Set other constraints (if necessary).
- Set indirect jump targets (if necessary).
- Set recursion bound (if necessary).
- Set processor configuration (optional).
- Perform Estimation.
- Perform Simulation (optional).

Set directories for SimpleScalar GCC bin, lp_solve, and simplesim-3.0

As mentioned before, Chronos needs SimpleScalar GCC (sslittle-na-sstrix-gcc). The installation of these tools were described in Section 2. After installation, we should tell Chronos where these tools are located. This is done as follows: Click "Simplescalar GCC bin directory" from the "Option" menu, then locate GCC bin directory from the dialog box, as shown in 4. For lp_solve and simplesim-3.0, click "ILP-solver directory" and "Simplesim-3.0 directory" respectively from the "Option" menu.

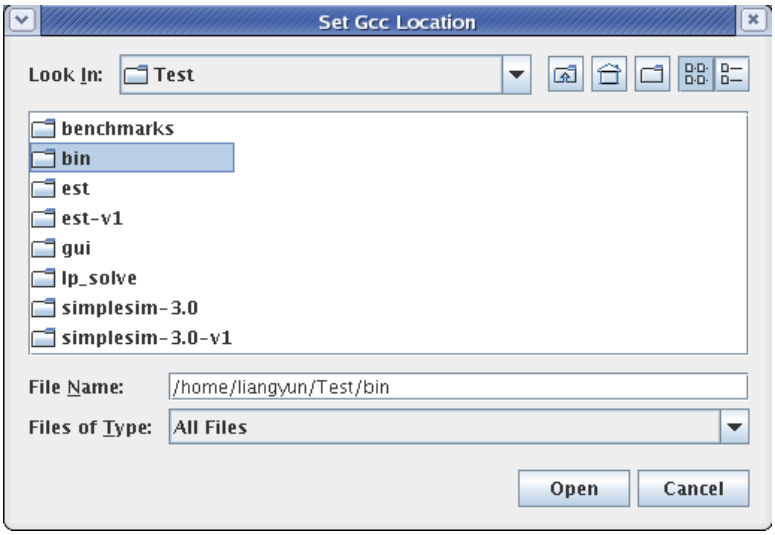

Figure 4: Set SimpleScalar Gcc bin tools directory

Open a benchmark directory

Choose the command "Open File..." from the "File" menu. A standard file dialog box will appear on the screen. After selecting the benchmark directory from the list box, Chronos will load the source code, and it will take the following actions. First it invokes SimpleScalar GCC to compile the benchmark. Next it disassembles the binary executable and reconstructs the control flow graph (CFG), which is shown in the third pane. In the CFG, each rectangle represents a basic block. For example `P:x B:y` represents the basic block y of procedure x. The outgoing edges of a basic block represent the possible outgoing control flows from this basic block. Chronos also dumps out the assembly code on the fourth pane with basic block annotations. The result is shown in Figure 5. Chronos can not model any library calls directly. If there are some library calls such as"sqrt" in the benchmark, the user has to put the source code of these library functions inside the benchmark directory.

Set user constraints

Chronos provides an user interface for giving additional flow constraints. Some of the user constraints are compulsory, like loop bounds that cannot be automatically derived by Chronos; or constraints that can further limit the possible program paths, like bounds on `if-then-else` branches. Loop bounds can be given in two forms. If an inner loop executes fixed number of times when entered from its parent loop or procedure, the user can select "Loop bound constraints" from the option menu and simply give the loop bound per invocation. Otherwise, a bound on the overall iterations of the inner loop should be given for better accuracy. In this case, select "Other constraints" from the "Option" menu and give a constraint like what is shown in Figure 6. Constraints for "if-then-else" statements can be given in a similar way. These source-level constraints will be converted to an internal constraint format in terms of basic blocks, as described in Section 3.

The user can also provide other annotations such as possible targets of indirect jumps and bounds for self-recursive functions. In order to do this, the user can select "Indirect jump targets" and "Recursion bound" from the "Option" menu and simply provide the possible target addresses and recursion depth. The target addresses can be found from the disassembly pane shown in Figure 5.

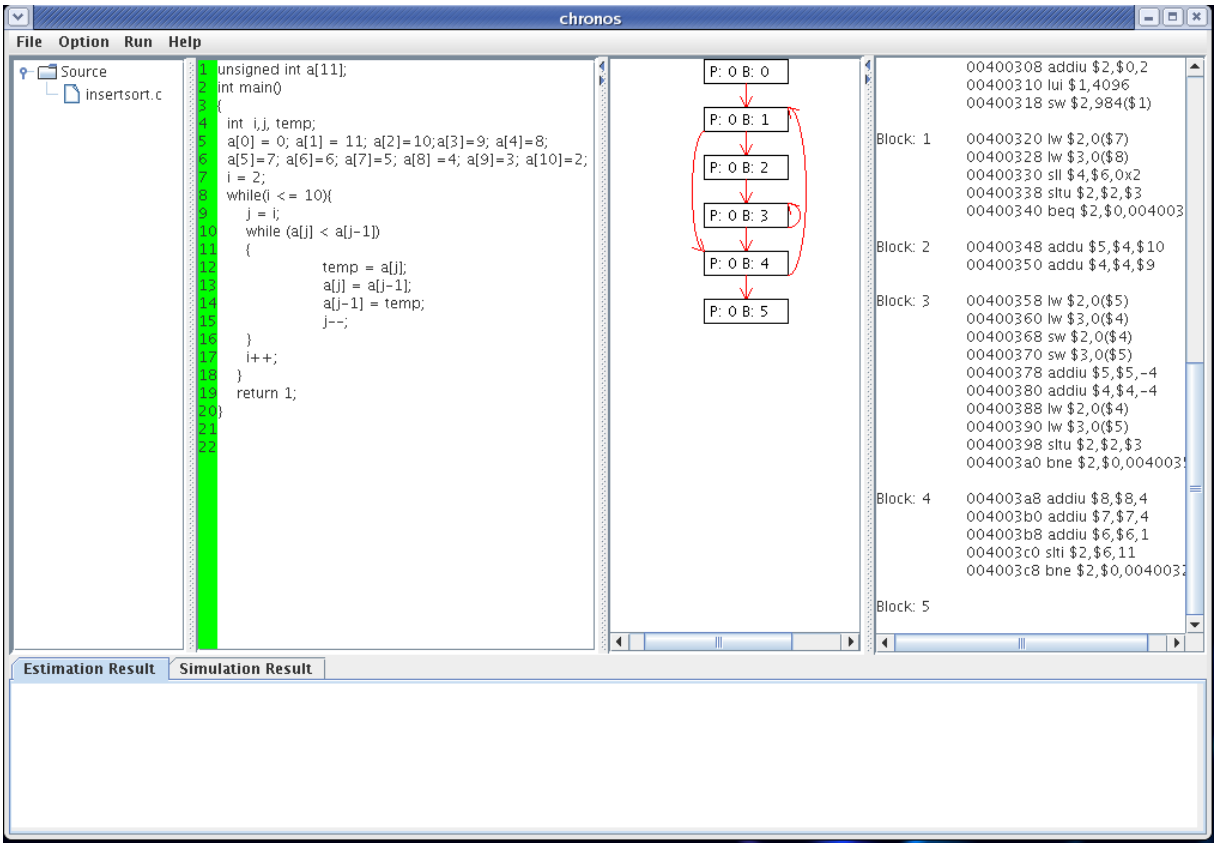

Figure 5: Source, CFG and Assembly Code of insertsort

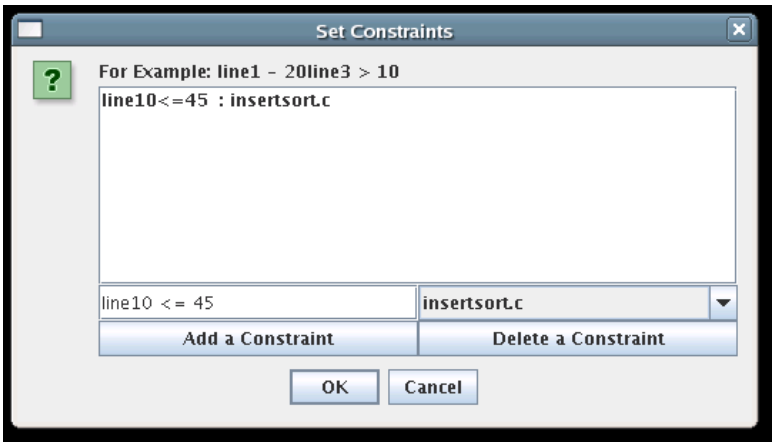

Figure 6: Other Constraints Dialog

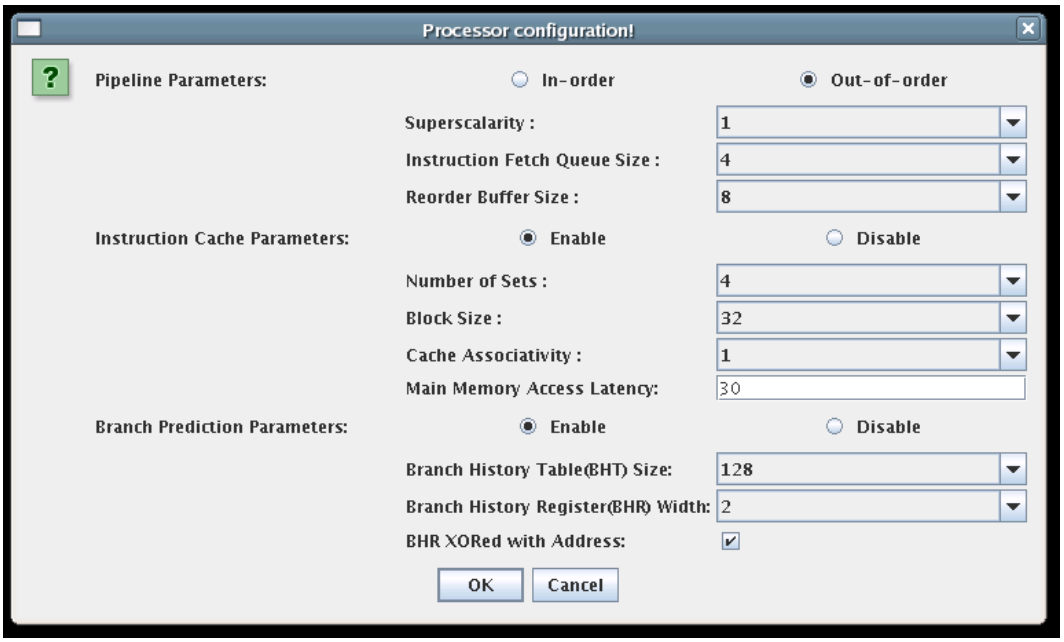

Figure 7: Processor Configuration Dialog

Change processor configuration

To change the processor configuration, click the command "Processor configuration" from the "Run" menu. and a dialog pops up for the user to set the processor parameters, as shown in Figure 7. The dialog only shows the features of SimpleScalar architecture modeled in Chronos.

Perform estimation

After giving user constraints or changing some processor parameters, click the command "Estimate" from the "Run" menu for WCET estimation. Chronos then invokes the analysis engine, formulates the program's WCET estimation as an ILP problem, and invokes lp_solve to solve the ILP problem. The estimated WCET, together with the number of branch mispredictions and the number of cache misses are returned by lp_solve, as shown in the pane "Estimation Result" in Figure 8.

Perform simulation

To see how accurate the estimated result is, you may want to perform a simulation of the benchmark program with the same processor configuration. Note that the simulation often does not capture the actual worst case; rather it yields a value that is lower bound of the *unknown* actual WCET. Our frontend provides an interface to SimpleScalar simulator. But the user can also choose to run the simulator independently. In the frontend, click the command "Simulate" from the "Run" menu. Chronos then invokes the `sim-outorder` simulator to run the program using the same processor configuration as estimation. The simulation cycles, number of branch mispredictions, as well as number of cache misses are dumped out to the pane "Simulation Result", as shown in Figure 9. Now the user can compare the simulated result to the estimated one.

5 Workflow and Analyzer Internals

This section provides a guidance for the user to understand the implementation of Chronos and to inspect its source code. The workflow of the analysis engine (plus some actions taken by the

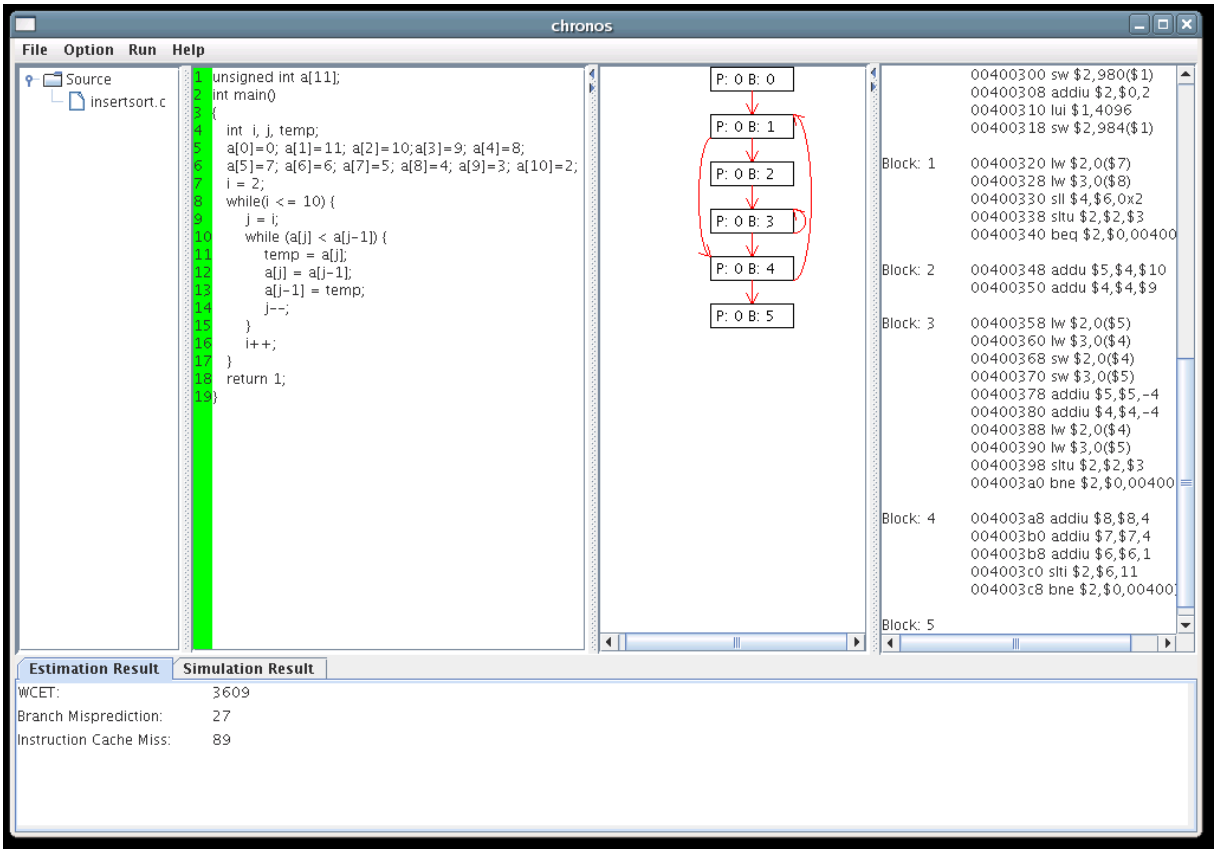

Figure 8: Estimation Result

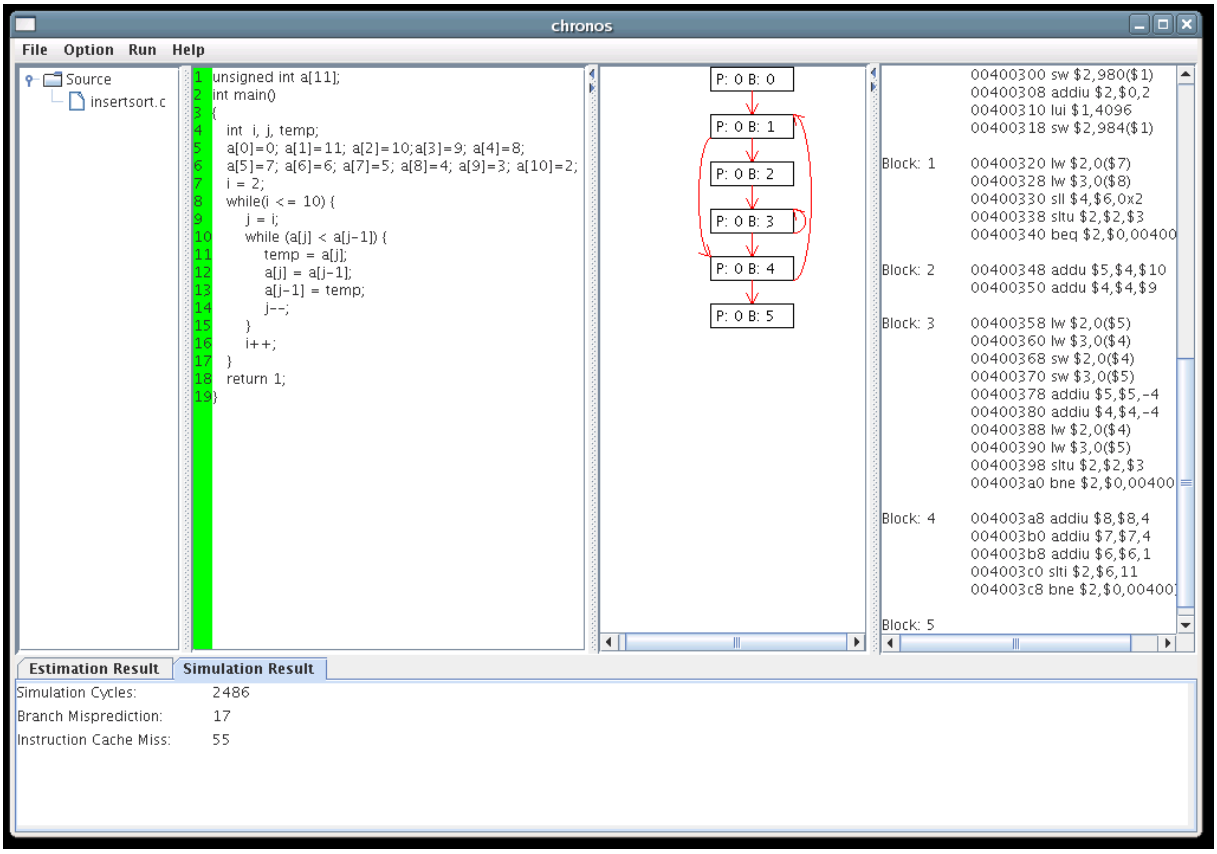

Figure 9: Simulation Result

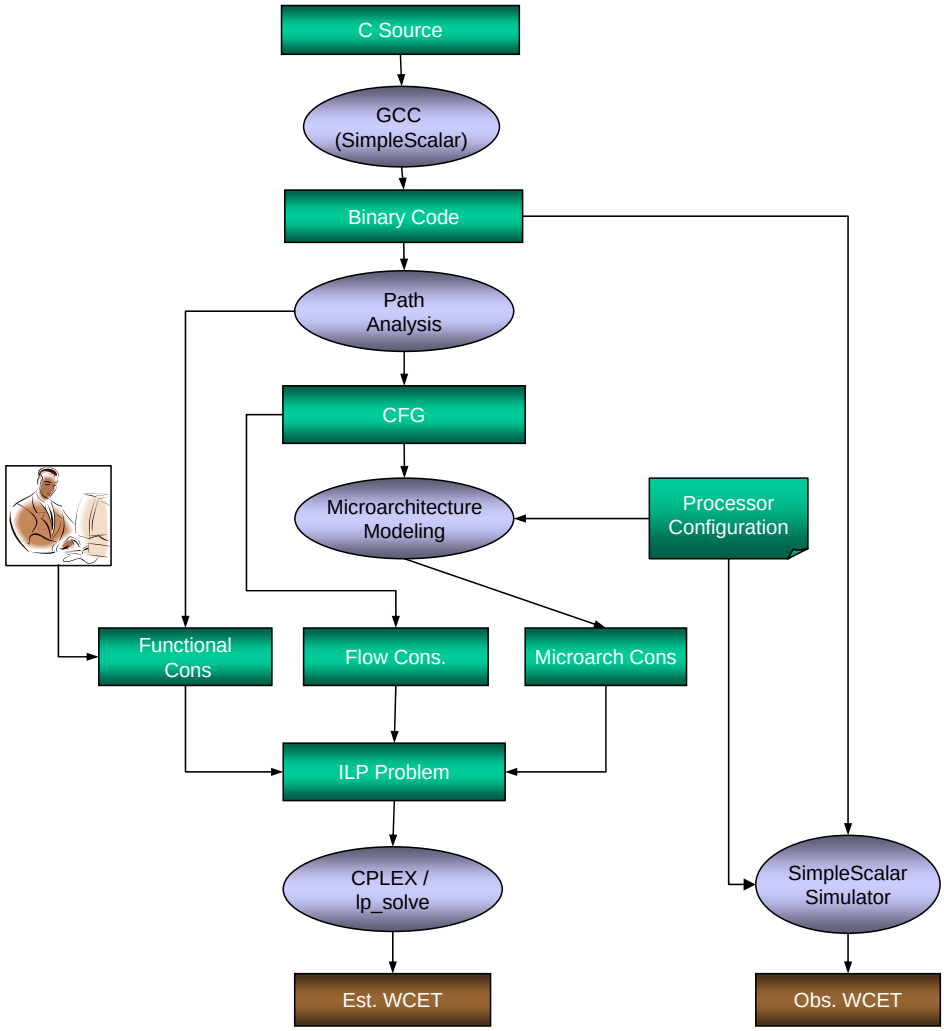

Figure 10: Workflow of Chronos

frontend) is illustrated in Figure 10. Now we describe the workflow of the analyzer in more detail and reveal the internals of the analyzer along with the description of the workflow.

Pre-analysis workflow

1. First, the frontend loads `benchmark` chosen by the user and invokes SimpleScalar GCC to compile it into binary executable. The compiled binary cannot run on the host machine, but can be simulated by SimpleScalar.

2. Next, the user may interact with the frontend further to determine the actions and processor configuration. After that, the frontend assembles arguments including (1) benchmark path, (2) action to be taken by the analyzer, and (3) processor configuration if non-default parameters are used, and invokes the analyzer **est**. We give a few examples of the equivalent command lines.

   - `est ../benchmarks/insertsort/insertsort`
     which takes the `insertsort` program and uses default processor configuration for WCET estimation.
   - `est ../benchmarks/insertsort/insertsort -run CFG`
     which takes the `insertsort` program but does not perform WCET estimation. Instead, it dumps out the CFG of `insertsort` to a file `insersort.cfg` (the option -run determines what action to be taken, and by default, it is -run EST, which can be omitted).
   - `est ../benchmarks/insertsort/insertsort -config processor.opt`
     which takes the `insertsort` program and uses a processor configuration stored in a disk file "processor.opt" for WCET analysis.

Analysis workflow

From now on, the analysis engines takes over and we will point out the corresponding places in the source code responsible for the functionalities described below.

1. **Path analysis:** The analyzer reads in the binary code, reconstructs the control flow graphs (CFG) for the procedures in `benchmark`. The CFGs can be dumped into a file named `benchmark.cfg` in the same directory of the `benchmark`. After the individual CFGs are constructed, The tool performs inter-procedural analysis and constructs a *global* flow graph called *transformed* CFG (refer to `struct tcfg_t` in `tcfg.h`) from the CFGs of the individual procedures and their calling relations. All subsequent analysis will be conducted on this transformed CFG instead of the CFGs of the individual procedures. This step corresponds to **path analysis** in the workflow in Figure 10, and its implementation is like this.

   ```
   path_analysis() {
       read_code();
       build_cfgs();
       prog_tran();
       loop_process();
   }
   ```

   These major functions are explained below:

   - `read_code()` in `readfile.c` reads in the program text from the object code.

- `build_cfgs()` in `cfg.c` identifies procedures in the benchmark (data structures `prog_t` and `proc_t` in `cfg.h`) and builds a control flow graph for each of the procedure (data structure `cfg_node_t` in `cfg.h`).
- `prog_tran()` in `tcfg.c` performs inter-procedural analysis and transforms the individual control flow graphs into a global control flow graph following the procedure call graph (data structure `tcfg_node_t` in `tcfg.h`) for each procedure. Subsequent analysis will use this transformed graph instead of working on the individual graphs.
- `loop_process()` in `loops.c` identifies loop levels and associate each basic block with the inner-most loop level containing it (data structure `loop_t` in `loops.h`).

Note that **path analysis** also performs data flow analysis to discover flow facts such as loop bounds, infeasible paths etc. The analysis results are remembered as a set of functional constraints (the block "functional cons" in the workflow diagram). This work is conducted by the frontend of Chronos when `benchmark` is loaded.

2. **Branch prediction analysis:** It performs branch prediction analysis to capture branch mispredictions. This is part of the component "Microarchitecture Modeling" in the workflow (refer to [5, 3] for technical details). Its implementation is as follows.

```
bpred_analysis() {
    collect_mp_insts();
    build_bfg();
    build_btg();
}
```

- `collect_mp_insts()` in `bpred.c` collects instructions along the wrong path for a branch instruction when it is mispredicted. These wrong path instructions will be used for evaluating the effect of misprediction on cache.
- `build_bfg()` in `bpred.c` captures the control flow transfer between adjacent branches under specific branch histories, i.e., the outcomes of recent branch instructions. The resulting data structure is a graph of nodes of `bfg_node_t` (in `bpred.h`).
- `build_btg()` in `bpred.c` further captures branch transfer information by building connections among (possibly) non-adjacent branches under the same branch history. This transition reflects how they interact via the branch predictor, and such information will directly lead to the bounding of branch mispredictions. The resulting data structure is a set of graphs of nodes `btg_node_t` (in `bpred.h`).

3. **Instruction cache analysis:** It performs instruction cache analysis to capture the occurrences of instruction cache misses. It is part of the component "Microarchitecture Modeling" in the workflow. Its implementation is as follows.

```
cache_analysis() {
    get_mblks();
    find_hitloop();
    categorize();
}
```

- `get_mblks()` in `cache.c` identifies memory blocks (a subsequence of instructions in a basic block that map to the same cache line). They correspond to the data structure `mem_blk_t` in `cache.h`.

- `find_hitloop()` in `cache.c` finds for each memory block a loop level where the memory block is guaranteed to hit in the cache as long as the execution is repeated within the loop level. But beyond that loop level, cache hit cannot be guaranteed.
- `categorize()` in `cache.c` creates a set of loop contexts for each basic block according to the hit loop levels found in the previous step. For example, if a basic block B has N memory blocks, whose hit loops are a set of $L(L \leq N)$ loops, then $L + 1$ loop contexts will be assigned to the execution of B, with the remaining memory blocks being categorized as not-hit in context of outer loop levels. See [4] for more technical details.

4. **Pipeline analysis:** It performs pipeline analysis to obtain the upper bound of execution times for each analysis unit (basic block) under the contexts of instruction cache information and branch prediction information. This means a basic block may have multiple upper bounds, each of which is applicable to a specific context (refer to [4] for technical details). Its implementations is like this.

```
pipe_analysis() {
    ...;
    est_units();
}
```

The function `est_units()` (in `pipeline.c`) invokes the function `ctx_unit_time()` (in `pipeline.c`) with (1) identifier of the unit under estimation; (2) a loop context where memory block hit/miss category can be decided; (3) prediction information for the branch at the entry to the estimated unit. The function `ctx_unit_time()` in turn invokes a set of functions iteratively over different path contexts (instructions preceding the estimated unit, which we call prologue; and instructions succeeding it, which we call epilogue). After all path contexts have been considered, the maximum estimation will be taken as the unit's worst case (under the given cache and branch context). Therefore, the implementation of `ctx_unit_time()` is like this.

```
ctx_unit_time(unit u, cache_context c, bpred_context b) {
    ...
    worst_case[u, c, b] = 0;
    for each prologue p {
        for each epilogue e {
            create_egraph();
            t = est_egraph();
            worst_case[u, c, b] = max(t, worst_case[u, c, b])
        }
    }
}
```

The two functions `create_egraph()` and `est_egraph()` implement the core algorithms for pipeline analysis.

- `create_egraph()` in `exegraph.c` constructs an execution graph given the estimation unit, its prologue/epilogue, as well as cache and branch prediction contexts. This execution graph is a static representation for all possible dynamic executions. Specifically, it captures dependences and contentions among instructions in the pipeline.
- `est_egraph()` in `estimate.c` works on the constructed execution graph, and yields an estimate. For details of the algorithm, please refer to [4].

5. **ILP problem formulation:** It formulates an ILP problem for the WCET analysis of the `benchmark` program based on: (1) path analysis (step 1); (2) microarchitecture modeling (steps 2, 3, 4); and (3) user provided functional constraints. The resulting ILP problem is in ILOG/CPLEX format and is written into a file named `benchmark.lp` within the same directory as the source and binary of `benchmark`. The major functions performing this task are:

   - `cost_func()` in `ilp.c`, which generates the objective function of the ILP problem.
   - `tcfg_cons()` in `ilp.c`, which generates the flow constraints from the transformed CFG created earlier.
   - `bfg_cons()`, `tcfg_bfg_cons()`, and `btg_cons()` in `ilp.c`, which generate branch prediction related constraints and additional constraints connecting the branch history transfer with pure control flow transfer.
   - `cache_cons()` and `mp_cache_cons()` in `ilp.c`, which generates cache miss constraints under normal execution and extra misses due to wrong path execution under branch mispredictions, respectively.
   - `tcfg_estunit_cons()` in `ilp.c`, which generates constraints connecting estimation units to transformed cfg nodes.
   - `user_cons()` in `ilp.c`, which reads in the user provided constraints from the file `benchmark.cons` in the directory of the `benchmark` program, and incorporates them into the ILP file.

Post-analysis workflow

The major functionality of the analysis engine is to perform path analysis and microarchitecture modeling, and transform the WCET analysis into an ILP problem. The solving of the ILP problem relies on third party LP/ILP solvers. Since the format of the ILP files is in ILOG/CPLEX format, both the commercial CPLEX ILP solver and the free lp_solve LP solver can be used for this purpose. For the user's convenience, we provide lp_solve along with the distribution, and briefly describe its usage.

By default, we assume the user works with the frontend of Chronos. Thus there is a command "estimate" in the graphical user interface, and the user does not need to be concerned with the usage of lp_solve. In case the user wants to invoke lp_solve to solve the ILP file, here is an example showing how to do that:

```
lp_solve -rxli ./xli_CPLEX ../benchmarks/matsum/matsum.lp
```

Note that the option "`-rxli ./xli_CPLEX`" is used to inform lp_solve that the input LP file is in ILOG/CPLEX format.

Another optional task is to perform simulation of the `benchmark` with SimpleScalar (should use the modified one provided with Chronos release) to get an idea of the accuracy of the estimation. Note that it is very important that the same processor configuration is used for estimation and simulation.

6 Summary

This is the *second release* of Chronos. This release focuses on modeling several advanced microarchitectural features commonly found in modern processors for Worst Case Execution Time (WCET) analysis. The modeled features include superscalar, out-of-order pipelines, instruction caches, and dynamic branch predictions. The framework developed in Chronos captures the interaction of these three features. We believe it can be easily extended to incorporate additional

microarchitectural features. The major improvements of Chronos-2.0 over Chronos-1.0 are the following.

- Chronos-1.0 processor model had very limited support for superscalarity. But Chronos-2.0 extends the processor model to the full SimpleScalar processor architecture with complete support for superscalarity.
- Chronos-2.0 extends instruction cache modeling to set-associative caches. Chronos-1.0 employed ILP-based cache modeling that can be applied to set-associative caches, but at the cost of long ILP solving time. Chronos-2.0 employs a fast categorization based instruction cache analysis.
- Considerable effort has been spent in the second release for a better implementation. Improvements over the first release include better and modular structure of the source code, a more efficient and scalable analysis engine, and a front-end supporting navigation among multiple source files.

The wish list for future releases of Chronos include the following.

- Model data cache. The distinct feature of a data cache is that its behavior is not only affected by the control flow, but the data values as well.
- Model real processors, especially the popular embedded processors such as ARM, PowerPC, MIPS, etc.
- Perform more aggressive path analysis to discover useful feasible paths/infeasible paths. The path information can substantially improve the analysis accuracy.

References

[1] Michel Berkelaar, Kjell Eikland, and Peter Notebaert. lp_solve version 5.5, 2005. free software, http://groups.yahoo.com/group/lp_solve.

[2] D. Burger and T. Austin. The SimpleScalar Tool Set, Version 2.0. Technical Report CS-TR-1997-1342, University of Wisconsin, Madison, June 1997.

[3] Xianfeng Li, Tulika Mitra, and Abhik Roychoudhury. Modeling control speculation for timing analysis. *Real-Time Systems Journal*, 29(1), January 2005.

[4] Xianfeng Li, Abhik Roychoudhury, and Tulika Mitra. Modeling out-of-order processors for wcet analysis. *Real-Time Systems Journal*, 34(3), November 2006.

[5] T. Mitra, A. Roychoudhury, and X. Li. Timing analysis of embedded software for speculative processors. In *ACM SIGDA International Symposium on System Synthesis (ISSS)*, 2002.

[6] Mälardelaen WCET research group. WCET benchmark programs. `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`.