XMOS Timing Analyzer Manual

REV B

Publication Date: 2013/5/30

SYNOPSIS

The XMOS Timing Analyzer (XTA) lets you determine the time taken to execute code on your target platform. Due to the deterministic nature of the XMOS architecture, the tools can measure the shortest and longest time required to execute a section of code. This document explains how to use the tool in xTIMEcomposer Studio and what to look out for in the results. It contains examples to help you get started and lists all the commands and options supported by the XTA.

Table of Contents

1 Introduction

The XMOS Timing Analyzer (XTA) lets you determine the time taken to execute code on your target platform. Due to the deterministic nature of the XMOS architecture, the tools can measure the shortest and longest time required to execute a section of code. When combined with user-specified requirements, the tools can determine at compile-time whether all timing-critical sections of code are guaranteed to execute within their deadlines.

The typical flow for using XTA is shown in Figure 1.

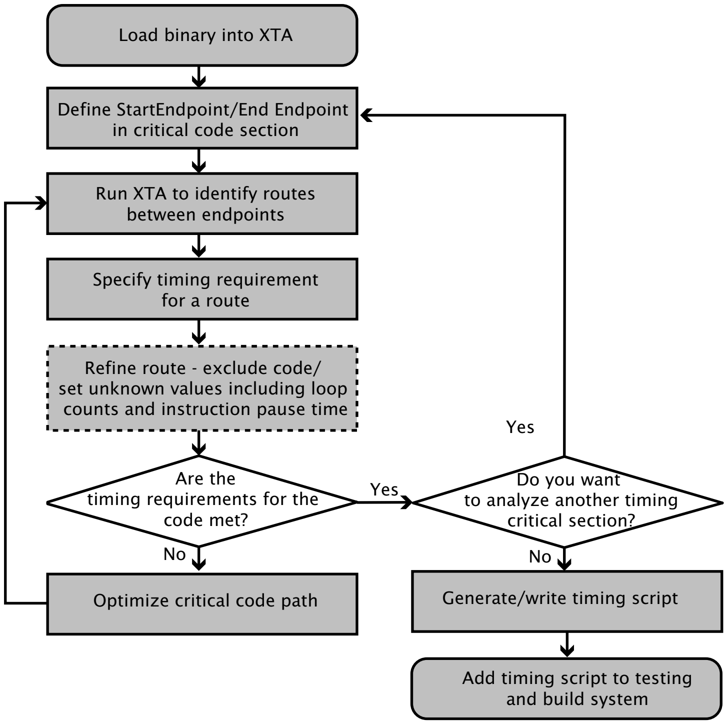

Load binary into XTA
Define StartEndpoint/End Endpoint in critical code section
Run XTA to identify routes between endpoints
Specify timing requirement for a route
Refine route - exclude code/ set unknown values including loop counts and instruction pause time
Are the timing requirements for the code met?
Yes
No
Optimize critical code path
Do you want to analyze another timing critical section?
Yes
No
Generate/write timing script
Add timing script to testing and build system

Figure 1: Typical XTA development flow

1. **Define the timing-critical code sections**: It is possible to define the execution time of functions and paths between two points in a program as being timing-critical. For each of the routes identified, the timing requirements need to be specified.

2. **Create a timing script**: Timing scripts are sequences of XTA console commands. In order to write portable scripts the source code needs to be annotated with XTA pragmas. The commands in the script can then refer to these pragmas.

xTIMEcomposer Studio can automatically generate a script and annotate source code with XTA pragmas once the user has identified the timing-critical routes.

3. **Use the script**: The timing script can be passed to the compiler or run in batch mode to verify that the program continues to meet its timing requirements each time it is compiled.

2 Defining Timing-Critical Code

IN THIS CHAPTER

2.1 Using The Tool

The user must load a compiled binary into the XTA, then define the timing-critical sections of code within the application and specify the timing constraints for each section of code. The tool supports timing functions, timing loops and the execution between two *endpoints*. The tool verifies that all possible paths of execution meet the specified timing constraints. If all paths meet their timing requirements then the minimum xCORE tile frequency the xCORE device can be run at to meet timing requirements is also output.

The tool can be used interactively on the command-line or through a graphical user interface (GUI) in xTIMEcomposer Studio.

2.2 Loading a Binary

When working within xTIMEcomposer Studio, loading a binary into the XTA is done by creating a new *Time* configuration. These work in the same way as the *Run* and *Debug* configurations. Once the configuration exists, the binary can be loaded into the XTA using the **Time** button in the toolbar. To load a binary that does not have an associated xTIMEcomposer project, see §10.1.7.

To load a binary from the console, type:

```
load <FILE NAME>
```

2.3 Routes

A *route* is a timing-critical section of code. It consists of the set of all paths through which control can flow between two points in a program (*endpoints*). A route can be created by timing a function, timing a loop or by timing between endpoints.

2.4 Endpoints

An *endpoint* is any source line that, during the compilation process, must be preserved, and its order with respect to other endpoints must be maintained. In the GUI the locations of valid endpoints are marked in the editor.

To show a list of all endpoints type the following command on the console:

```
list allendpoints
```

If specifying a route with respect to assembly code then any valid label/program counter (PC) can be used as an endpoint. However, program counters are classed as non-portable endpoints as they are likely to change between compilations and their use in scripts is therefore discouraged.

2.5 Adding Endpoints To Source

Source lines can be labeled with endpoint pragmas in order to ensure that the endpoints are portable. For example, Figure 2 shows a function that has been annotated with endpoint pragmas called `start` and `stop`.

```
int g(in port p) {
    int x, y;

    # pragma xta endpoint " start "
    p :> x;

    # pragma xta endpoint " stop "
    p :> y;

    return (y - x);
}
```

Figure 2: Putting an endpoint pragma into XC.

The endpoints defined in source are listed in the *Info View*.

On the console, type:

```
list endpoints
```

2.6 Timing Between Endpoints

In xTIMEcomposer Studio set the *from* and *to* endpoint and then click the **Analyze endpoints** button. Endpoints are set by right-clicking on the marker showing where valid endpoints are in the source.

On the console, type:

```
analyze endpoints <from ENDPOINT> <to ENDPOINT>
```

The tool does not time code across multiple xCORE tiles so both endpoints must be on the same tile.

One analysis can result in multiple routes being generated (see §6.1 for further details).

2.7 Timing Functions

In xTIMEcomposer Studio select the function name from the list of available functions using the **Analyze function** button.

On the console, type:

```
analyze function <FUNCTION>
```

This will create a route which describes the set of all possible paths from the function entry point to all the function return points.

2.8 Timing Loops

In xTIMEcomposer Studio set the *looppoint* and then click the **Analyze loop** button. Looppoints are set by right-clicking in the margin and selecting the **Set loop point** option.

On the console, type:

```
analyze loop <ANY>
```

This creates a route that describes all possible paths through the loop. It it effectively a shortcut for timing between endpoints where the start and stop endpoint is the same, the point is within a loop and an exclusion has been placed such that everything outside the loop is excluded.

One analysis can result in multiple routes being generated (see §6.1 for further details).

2.9 Setting Timing Requirements

For each route created it is necessary to define its timing requirements. In xTIMEcomposer Studio right-click on the route in the upper panel of the *Routes View* and select **Set timing requirements** in the context menu.

On the console, type:

```
set required <route id> <value> <MODE>
```

The supported timing modes are defined in §12.21.

The route IDs can be found in the console by typing:

```
print summary
```

Alternatively, the - character can be used on the command-line or in a script to refer to the last route analyzed.

3 Viewing Results

IN THIS CHAPTER

3.1 Route IDs

All analyzed routes are given a unique route ID. However, when referring to routes in a script, using the route ID may not always result in portable or robust scripts. In many cases, the only route that needs to be referenced is the one that was last analyzed. This can be achieved by using the '-' character as the route ID. If the last command created multiple routes then the '-' character refers to all of the routes created.

3.2 Node IDs

Within a single route, all nodes are assigned a unique ID number. This is required as input for some of the console commands. The '-' character can be used in this context to refer to the top level node of the route.

3.3 Summary

In the GUI all routes created are shown in the upper pane of the *Routes View* in the xTIMEcomposer. More detailed information on each route can be found by hovering over the route in this list.

On the console, type:

```
print summary
```

Details for a specific route are shown using the command:

```
print routeinfo <route id>
```

3.3.1 Violation

When a timing requirement has been set for a route and the route takes more time to execute than required, the time difference is called a *violation*. This value specifies how much faster the route needs to be executed in order to meet the timing requirement.

3.3.2 Slack

When a timing requirement has been set for a route and the route takes less time to execute than required, the time difference is called *slack*. This value specifies how much slower the route could be executed and still meet the timing requirement.

3.4 Structure

The structure of a route can be examined in xTIMEcomposer Studio in two ways. The lower pane of the *Routes View* shows the structure of the selected route as a tree and the *Structure Panel* of the *Visualizations View* renders it graphically.

In the console, to display the the structure of a route type:

```
print structure <route id>
```

The structure used by the tool is described in §5.2.

3.5 Source Code Annotation

In xTIMEcomposer Studio the editors highlight code which is executed by the section of a route that has been selected. The selection is done using the lower pane of the *Routes View*.

In the console, to display the source code which is executed by a route type:

```
print src <route id>
```

If only a part of a route should be used then the node ID can be specified:

```
print src <route id> <node id>
```

3.6 Instruction Traces

To help the user understand the execution flow of a route, the tool supports creating representative instruction traces. In xTIMEcomposer Studio right-click on the route and select **Trace in console**.

In the console type:

```
print trace <route id>
```

As a result of loops being unrolled when tracing, it is possible for the traces to get very large. The trace operation can be cancelled at any time by pressing the **Escape** key in xTIMEcomposer Studio or *CTRL+C* in the command-line tool.

A trace can be redirected to a file by using the console command:

```
print trace <route id> > <file>
```

Worst/Best Case

By default, the trace for worst-case path is printed. This can be changed to print the best-case path instead.

In xTIMEcomposer Studio, to print the best case path instead, open the **Timing Properties** and uncheck **Print worst case**.

On the console, type:

```
config case best
```

3.7 Fetch no-ops

The xCORE device may need to pause at certain times while more instructions are fetched from memory. This results in the issue of fetch no-op instructions. These are shown in the traces as `FNOP` at the points they will happen on the hardware.

In xTIMEcomposer Studio they are inserted into the disassembly at the points they occur.

3.8 Scaling Results

By default, the tool scales all timing results. This means that the appropriate unit (ms, us, ns) will be used to print time values. This can be changed so that all times are printed in ns.

In xTIMEcomposer Studio open the **Timing Properties** and uncheck the **Scale results**.

On the console, type:

```
config scale false
```

3.9 Unknowns

The tool may not always be able to determine the exact timing of a section of code. This happens when the tool is unable to determine loop iteration counts or the execution time of instructions. These conditions are reported as warnings in the tool. The unknowns for the currently selected route are shown in the **Unknowns** section of the *Info View*.

To display unknowns on the console, type:

```
list unknowns <route id>
```

§4 describes how to address these warnings.

4 Refining Timing Results

IN THIS CHAPTER

There are cases where the tool is unable to fully determine the timing of a section of code, due to, for example, not being able to determine a loop count. This can be addressed by adding *defines*. Defines can be added in two ways, to a *global list*, or to a *route-specific list*. Those added to the *global list* get applied to every route when upon creation.

⚠ The use of the global list can result in more concise scripts. However, It is important to be careful with defines added to the *global list* since they are ignored if they fail to get applied to a route. This allows a full set of defines to be created before any routes, but does mean that errors in these defines might be missed. Route specific defines (added post route creation) will always flag an error if there is one.

4.1 Exclusions

Not all paths of execution in a route may be timing-critical. The route may contain cases to handle errors where the timing of the code is not important. These paths can be ignored in the timing script by adding exclusions. *Exclusions* tell the tool to ignore all paths which pass through that code point. Exclusions can be added to the global list or applied to a specific route.

In xTIMEcomposer Studio, to set an exclusion on an existing route, right click within the relevant path and select **Exclude**. This can be done in the editor, the lower panel of the *Routes View* or the *Structure* tab of the *Visualizations View*.

On the console, type:

```
set exclusion <route id> <ANY>
```

To add an exclusion to the list of exclusions to be taken into account during route creation, right click within the relevant path and select **Add to exclusion list**. This can be done on the vertical ruler of the text editor, in the *Disassembly View*, the lower panel of the *Routes View* or the *Structure* tab of the *Visualizations View*.

On the console, type:

```
add exclusion <ANY>
```

The current global list of exclusions can be found in the **Exclusions** section of the *Info View*.

On the console, type:

```
list exclusions
```

To remove an exclusion from the global list, find the relevant exclusion in the *Info View*, right click and choose **Remove** or press **Delete**.

On the console, type:

```
remove exclusion <ANY|*>
```

For example, consider the code in Figure 3.

```
int calculate ( int a, int b) {
    if ( willOverflow (a, b) {
        # pragma xta label " overflow "
        return processOverflow ();
    }
    return a + b;
}
```

Figure 3: Excluding an invalid path

To time the `calculate` function ignoring the error case:

- Using route-specific defines:
  - `analyze function calculate`
  - `set exclusion - overflow`
- Using global defines:
  - `add exclusion overflow`
  - `analyze function calculate`

Although functionally equivalent, exclusion via the global defines mechanism can result in faster, and more memory efficient, route creation. This is because the global exclusions can be taken into account during route creation, so the search space can be reduced. For post route creation exclusions, the complete route is created before any pruning occurs.

4.2 Loop Iterations

Loop iteration counts can be unknown. Whenever possible, the compiler tells the tool about loop iteration counts. However, some loop counts are not known statically. In these cases the user is required to specify worst-case values.

The compiler does not emit any loop iteration counts unless optimizations have been enabled (-O1 or greater).

Some loops are self loops (loops whose body is the same as the header) and therefore have a minimum iteration count of 1.

In xTIMEcomposer Studio, to set loop iterations on an existing route, right click within the body of the loop, but not within an inner loop, and select **Set loop iterations**. This can be done on the vertical ruler of the text editor, in the *Disassembly View*, the lower pane of the *Routes View* or the *Structure* tab of the *Visualizations View*.

On the console, type:

```
set loop <route id> <ANY> <iterations>
```

To add an iteration count to the list of iteration counts to be used during route creation, click the **Add Define** button in the toolbar, and select the *Loop* tab.

On the console, type:

```
add loop <ANY> <iterations>
```

The current list of global loop iteration counts can be found in the defines section of the *Info View*.

On the console, type:

```
list loops
```

To remove a loop iteration count from the global list, find the relevant entry in the *Info View*, right click and choose **Remove** or press **Delete**.

On the console, type:

```
remove loop <ANY|*>
```

For example, consider the code in Figure 4.

To time the `test` function:

```
void delay ( int j) {
    for ( unsigned int i = 0; i < j; ++i) {
        # pragma xta label " delay_loop "
        delay_us (1);
    }
}

int test () {
    delay (10);
}
```

Figure 4: Setting loop iterations.

- Using route-specific defines:
  - `analyze function test`
  - `set loop - delay_loop 10`
- Using global defines:
  - `add loop delay_loop 10`
  - `analyze function test`

4.3 Loop Path Iterations

A loop may contain multiple paths through it. When a loop iteration count has been set the tools assumes that all iterations will take the worst-case path of execution through the loop. This is not always the case, and a more realistic worst-case can be established by specifying the number of iterations on individual paths through the loop.

To set loop path iterations on an existing route, right click on a path within the body of the loop, and select **Set loop path iterations**. This can be done on the vertical ruler of the text editor, in the *Disassembly View*, the lower pane of the *Routes View* or the *Structure* tab of the *Visualizations View*.

On the console, type:

```
set looppath <route id> <ANY> <iterations>
```

To add a loop path count to the list of loop path counts to be used during route creation, click the **Add Define** button in the toolbar, and select the *Loop path* tab.

On the console, type:

```
add looppath <ANY> <iterations>
```

The current list of global loop path counts can be found in the defines section of the *Info View*.

On the console, type:

```
list looppaths
```

To remove a loop path count from the global list, find the relevant entry in the *Info View*, right click and choose **Remove** or press **Delete**.

On the console, type:

```
remove looppath <ANY|*>
```

There are some rules that need to be followed when setting loop path iterations:

- In a nested loop, the outer loop iterations need to be set first.
- The loop path iterations set must be less than or equal to the loop iterations set on the enclosing loop.
- If the loop path iterations set are less than that of the enclosing loop, then there must exist another path within the loop without its iterations set to which the remaining iterations can be allocated.

For example, consider the code in Figure 5:

```
void f( int j) {
    for ( unsigned int i = 0; i < j; ++i) {
        # pragma xta label " f_loop "
        if ((i & 1) == 0) {
            # pragma xta label " f_if "
            g ();
        }
    }
}

int test () {
    f (10);
}
```

Figure 5: Setting loop path iterations.

To time the `test` function:

- Using route-specific defines:
  - `analyze function test`
  - `set loop - f_loop 10`
  - `set looppath - f_if 5`
- Using global defines:
  - `add loop f_loop 10`

  - `add looppath f_if 5`
  - `analyze function test`

4.4 Loop Scope

By default, the tool assumes that the iterations for loops are *relative*—the iterations for an inner loop will be multiplied by the iterations of enclosing loops. However this is not sufficient to describe all loop structures. If this assumption is not correct a loop count can be set to *absolute*. The iteration count set on an absolute loop is not multiplied up by the iterations set on enclosing loops.

To set loop scope on an existing route, right click within the body of the loop, and select **Set loop scope**. This can be done on the vertical ruler of the text editor, in the *Disassembly View*, the lower pane of the *Routes View* or the *Structure* tab of the *Visualizations View*.

On the console, type:

```
set loopscope <route id> <ANY> <absolute|relative>
```

To add a loop scope to the list of loop scopes to be used during route creation, click the **Add Define** button in the toolbar, and select the *Loop scope* tab.

On the console, type:

```
add loopscope <ANY> <absolute|relative>
```

The current list of global loop scopes can be found in the defines section of the *Info View*.

On the console, type:

```
list loopscopes
```

To remove a loop scope from the global list, find the relevant entry in the *Info View*, right click and choose **Remove**.

On the console, type:

```
remove loopscope <ANY|*>
```

For example, consider the code in Figure 6

To time the `test` function:

- Using route-specific defines:
  - `analyze function test`
  - `set loop - outer_loop 10`

```
void f( int l) {
    for ( unsigned int i = 0; i < l; ++i) {
        # pragma xta label " outer_loop "
        for ( unsigned int j = 0; j < i; ++j) {
            # pragma xta label " inner_loop "
            g ();
        }
    }
}

void test () {
    f (10);
}
```

Figure 6: Setting loop scope.

  - `set loop - inner_loop 45`
  - `set loopscope - inner_loop absolute`
- Using global defines:
  - `add loop outer_loop 10`
  - `add loop inner_loop 45`
  - `add loopscope inner_loop absolute`
  - `analyze function test`

4.5 Instruction Times

Some instructions can pause the processor. By default, the tool reports timing assuming that no instructions pause, but flags them as warnings. The user must specify what the worst-case execution time of instructions are.

To set an instruction time in an existing route, right click on the instruction and select **Set instruction time**. This can be done on the vertical ruler of the text editor, in the *Disassembly View*, the lower pane of the *Routes View* or the *Structure* tab of the *Visualizations View*.

On the console, type:

```
set instructiontime <route id> <ENDPOINT> <value> <MODE>
```

To add an instruction time to the list of instruction times to be used during route creation, click the **Add Define** button in the toolbar, and select the *Instruction time* tab.

On the console, type:

```
add instructiontime <ENDPOINT> <value> <MODE>
```

The current list of global instruction times can be found in the defines section of the *Info View*.

On the console, type:

```
list instructiontimes
```

To remove an instruction time from the global list, find the relevant entry in the *Info View*, right click and choose **Remove** or press **Delete**.

On the console, type:

```
remove instructiontime <ANY|*>
```

For example, consider the code in Figure 7.

Figure 7: Setting an instruction time.

```
void f( port p) {
    # pragma endpoint " instr "
    p :> value ;
}
```

To time the `f` function:

- Using route-specific defines:
  - `analyze function f`
  - `set instructiontime - instr 100.0 ns`
- Using global defines:
  - `add instructiontime instr 100.0 ns`
  - `analyze function f`

4.6 Function Times

In some cases it is necessary to define the time it takes to execute an entire function. The tool supports defining a function time. Once a function time is defined, all the unknowns within it are ignored and any routes which span this function will use the defined time instead of calculating it.

To set a function time on an existing route, right click on a function and select **Set function time**. This can be done in the the lower pane of the *Routes View* or the *Structure* tab of the *Visualizations View*.

On the console, type:

```
set functiontime <route id> <FUNCTION> <value> <MODE>
```

To add a function time to the list of function times to be used during route creation, click the **Add Define** button in the toolbar, and select the *Function time* tab.

On the console, type:

```
add functiontime <FUNCTION> <value> <MODE>
```

The current list of global function times can be found in the defines section of the *Info View*.

On the console, type:

```
list functiontimes
```

To remove an function time from the global list, find the relevant entry in the *Info View*, right click and choose **Remove**.

On the console, type:

```
remove functiontime <FUNCTION|*>
```

For example, consider the code in Figure 8.

```
void delayOneSecond () {
    g ();
    }

void test () {
    delayOneSecond ();
}
```

Figure 8: Setting a function time.

To time the `test` function:

- Using route-specific defines:
  - `analyze function test`
  - `set functiontime - delayOneSecond 1000.0 ms`
- Using global defines:
  - `add functiontime delayOneSecond 1000.0 ms`
  - `analyze function test`

4.7 Path Times

In some cases it is necessary to define the time it takes to execute a particular section of code. The tool supports defining a path time for this case. Once a path time is defined all the unknowns within it are ignored, and any routes which span this section of code will use the defined time instead of calculating it.

To set a path time on an existing route, highlight the start and end nodes in the lower pane of the *Routes View*, right click and and select **Set path time**.

On the console, type:

```
set pathtime <route id> <from ENDPOINT> <to ENDPOINT> <value> <MODE>
```

To add a path time to the list of path times to be used during route creation, click the **Add Define** button in the toolbar, and select the *Path time* tab.

On the console, type:

```
add pathtime <from ENDPOINT> <to ENDPOINT> <value> <MODE>
```

The current list of global path times can be found in the defines section of the *Info View*.

On the console, type:

```
list pathtimes
```

To remove an path time from the global list, find the relevant entry in the *Info View*, right click and choose **Remove**.

On the console, type:

```
remove pathtime <from ENDPOINT|*> <to ENDPOINT|*>
```

For example, consider the code in Figure 9.

Figure 9: Setting a path time.

```
int f() {
    int time ;
    timer t;
    # pragma xta endpoint " start "
    t :> time ;
    # pragma xta endpoint " stop "
    t when timerafter ( time + 100) :> time ;
}

void test () {
    f ();
}
```

To time the `test` function:

- Using route-specific defines:
  - `analyze function test`
  - `set pathtime - start stop 1000.0 ns`
- Using global defines:
  - `add pathtime start stop 1000.0 ns`
  - `analyze function test`

4.8 Active Tiles

By default the tool finds routes on all tiles within a program. However, it is possible to restrict the tool to work only on a subset of the tiles in the program. The set of tiles all commands apply to is called the *active* tiles.

In xTIMEcomposer Studio open the **Timing Properties** and select which tiles are active.

On the console, type:

- `add tile <tile id>`
- `remove tile <tile id|*>`
- `list tiles`

4.9 Node Frequency

An XMOS device consists of a number of nodes, each one composed of a number of xCORE tiles. The frequency at which a node runs is defined in the binary and the tool reads this and configures the node frequencies when it loads the binary. It is possible to experiment to determine what will happen at different frequencies if desired.

In xTIMEcomposer Studio open the **Timing Properties** and change the frequency for the node to be changed.

On the console, type:

```
config freq <node id> <tile frequency>
```

4.10 Number Of Logical Cores

The maximum number of logical cores run on a tile is known at compile time and the tool extracts this information from the binary for each tile. It is possible to experiment to determine what will happen if running with a different number of cores if desired.

In xTIMEcomposer Studio open the **Timing Properties** and change the number of cores for the node/tile to be changed.

On the console, type:

```
config cores <tile id> <num cores>
```

5 Program Structure

IN THIS CHAPTER

It is essential to understand how program structure is defined in order to use the tool correctly.

Programs are written in multiple source files, each containing functions. Each function will contain sequences of statements, loops (e.g. `for` / `while` / `do`), conditionals (e.g. `if` / `switch`) and function calls.

5.1 Compiling For XTA

The compiler outputs information which allows the XTA tool to make associations between source and instructions. This information is on by default but can be disabled by adding the following flag to the compiler options:

```
-fno-xta-info
```

The compiler also supports adding debug information without affecting optimizations. Debug information is not required in order for the XTA tool to analyze code, but the mapping between instructions and source code is not available without the debug information. In order to add debug information compile with:

```
-g
```

5.2 Structural Nodes

The compiler tools create a binary file with one program per xCORE tile. The XTA tool uses the binary file in order to produce accurate timing results.

When a route is created, the tool analyzes the binary to create a structure which closely represents the high-level program structure. It decomposes the program into structural nodes which can be displayed as a tree.

The worst and best case time is then calculated for each of the structural nodes. The way this is calculated depends on the type of structural node. The worst and best case times for the overall route is built up from the worst and best case times of the sub nodes.

The structural nodes can be of the following types:

- Instruction: the most basic building block of the program is the instruction.
- Block: a list of instruction nodes with no conditional branching which is therefore executed in sequence. The worst/best case time for a block is the sum of its component instructions.
- Sequence: a list of structural nodes which are executed in order. The worst/best case time for a sequence is the sum of the worst/best case times of its sub nodes.
- Conditional: a set of structural nodes out of which at most one node is executed. If this is within a loop then on each iteration a different node might be chosen. In some cases the entire conditional is optional. In those cases the best case time is for none of the options to be taken. The worst/best case time for a conditional is determined by the worst/best case time of each of its sub nodes.
- Loop: consists of a header and a body (both of which are structural nodes). The header corresponds to the conditional test part of the loop, and the body corresponds to the code that is executed if the loop is taken. This roughly corresponds to high level code structures such as `while` or `for` loops.

  The body is executed once per iteration. The header always executes once more than the number of iterations. The worst/best case times for a loop is the worst/best case time of its header multiplied by (number of iterations + 1) plus the worst/best case time of the body multiplied by the number of iterations.
- Self-loop: a loop where the header and body are the same. It is therefore considered to have a minimum loop count of 1. This roughly corresponds to high level code structures such as `do` loops. The worst/best case time for a self-loop is determined by the worst/best case time of its body multiplied up buy the number of iterations.
- Function: is the high-level construct of the function and consists of a list of other structural nodes. The worst/best case time for a function is calculated in the same way as that of a sequence.

5.3 Identifying Nodes: Code References

A `code reference` is the way to specify a particular location in an application. A code reference is made up of a base and an optional backtrail. The base consists of a `reference type` and the backtrail consists of a comma separated list of `reference types`.

There are a number of different reference types, all of which map to one or more instruction program counters (PCs). This will usually be one PC, but can be more than one due to compiler optimizations or because the user has explicitly named multiple instructions with the same reference. Compiler optimizations such as inlining or unrolling will result in the same reference mapping to multiple PCs. For the grammar of specifying a code reference see §14.

The different reference types are detailed below. The commands to list the instances of them for the currently loaded executable in the console are detailed with each type. In xTIMEcomopser Studio the available references are shown in the *Info View*.

5.3.1 Source File-Line

Source file-line references are valid for source lines which the compiler has defined as belonging to a source-level basic block. The valid lines can be listed in the console with:

```
list allsrclabels
```

5.3.2 Source Label

Source labels are added to source code using the `#pragma xta label`. To list the source labels in the console type:

```
list srclabels
```

5.3.3 Call File-Line

Call file-line references are valid for source lines which map to function calls. To list the valid source lines in the console type:

```
list allcalls
```

5.3.4 Call Label

Call labels are added to source code using the `#pragma xta call`. To list the source labels in the console type:

```
list calls
```

5.3.5 Endpoint File-Line

Endpoint file-line references are available for source lines which map to a valid *endpoint*. To list the endpoints in the console type:

```
list allendpoints
```

5.3.6 Endpoint Label

Endpoint labels are added to the source using `#pragma xta endpoint`. They must be on the line before an input/output operation. To list the labeled endpoints in the console type:

```
list endpoints
```

5.3.7 Label

Labels are arbitrary text strings referring to any source or assembly label. To list the labels in the console type:

```
list labels
```

Labels in assembly must be within an executable section.

5.3.8 Function

Functions are the functions contained within the binary. To list the labels in the console type:

```
list functions
```

Functions in assembler must be labeled as functions with the `.type` directive in order to be correctly detected by the tool (see xTIMEcomposer Studio User Guide). They must also be within an executable section.

5.3.9 Program Counter (PC)

Program counters are the lowest-level reference, giving a hexadecimal program counter value starting with `0x`. They must map to the PC of an instruction within the executable section of the program.

5.4 Reference Classes

Particular console commands and GUI actions only work on particular types of references. The sets of reference types that are defined for a particular command is know as a reference classes.

5.4.1 ENDPOINT

These are references which can be used for timing. This means any reference in assembler (PC/label) and only source references which map to lines which can be reliably used for timing. Compiler optimizations cannot remove them or re-order them with respect to each other. In XC code these correspond to source lines with

I/O operations. The following console command lists the types available in the class:

```
help ENDPOINT
```

5.4.2 CALL

These references map to function calls. These are used in *back trails* to identify unique instances of a code reference. The following console command lists the types available in the class:

```
help CALL
```

5.4.3 FUNCTION

These references map to functions. The following console command lists the types available in the class:

```
help FUNCTION
```

5.4.4 LABEL

The following console command lists the types available in the class:

```
help LABEL
```

5.4.5 PC

The following console command lists the types available in the class:

```
help PC
```

5.4.6 Forcing a Specific Type

It is possible to have a code reference which could map to multiple types. For example there could be an endpoint which has been given the same name as a function in the program. The way a reference in a backtrail is matched can depend upon the type of the reference. In order to resolve this potential ambiguity, it is possible to force the code reference to a certain type by prefixing with its type. See §14 for details.

5.5 Back Trails

A code reference's base may occur multiple times within a program. For example, a function can be called from multiple places. The *back trail* for a reference is a way of restricting a reference to specific instances. Consider the example file shown in Figure 10.

```
1  void delay_n_seconds ( int j) {
2      for ( unsigned int i = 0; i < j; ++i) {
3          # pragma xta label " delay_loop "
4          delay_1_second ();
5      }
6  }
7
8  int test () {
9      # pragma xta call " delay_1 "
10     delay_n_seconds (10);
11     # pragma xta call " delay_2 "
12     delay_n_seconds (20);
13     return 0;
14 }
```

Figure 10: Using backtrails.

The following commands could be used to time the `test` function:

- `analyze function test`
- `set loop - delay_loop 10`

That would have the effect of setting the number of loop iterations for the loop in both instances of the `delay_n_seconds` to 10. However, as the number of iterations are passed as a parameter to delay_n_seconds, the value is different for each call.

To time test correctly the loop iterations for each instance needs to be specified differently. This can be achieved by the use of the call references and backtrails. For example:

- `analyze function test`
- `set loop - delay_1,delay_loop 10`
- `set loop - delay_2,delay_loop 20`

This tells the tool to set `delay_loop` to 10 iterations when called from `delay_1`, and to 20 iterations when called from `delay_2`. The references used in the above case are composed of a base reference of type source label, and a backtrial or size one, of type call label. The above can also be achieved using the file-line equivalents. For example:

- `analyze function test`
- `set loop - source.xc:10,source.xc:3 10`
- `set loop - source.xc:12,source.xc:3 20`

However, this would not have resulted in a portable and robust script implementation, so using file-line references in this way from a script is not encouraged.

5.5.1 Inlining

When the compiler inlines some code (for example the `delay_n_seconds` function above) then some references will no longer be valid. In this case the following reference would not exist because the call no longer exists:

```
source.xc:10,source.xc:3
```

However, if the call has been labeled with a call label, the compiler ensures that the reference is still valid even if the code is inlined. So, in the above case, the following reference will still be valid;

```
delay_1,delay_loop
```

5.6 Scope of References

References can have either *global* or *local* scope. Globally scoped references are those which apply to (or get resolved on) the global tree. The global tree is the notional structural representation of the whole program, prior to any route analysis taking place. Locally scoped references are those which apply to (or get resolved on) a user created route tree. Whether a particular reference is globally of locally scoped depends on the command being executed. The following commands used globally scoped references:

- `analyze path`
- `analyze function`
- `analyze loop`
- `add exclusion`
- `add branch`

The following commands used locally scoped references:

- `set/add loop`
- `set/add looppath`
- `set/add loopscope`
- `set/add instructiontime`
- `set/add pathtime`
- `set/add functiontime`

In general, globally scoped references can lead to multiple route creation.

6 When Code Analysis Succeeds

IN THIS CHAPTER

▶ Multiple Route Creation

When analysis completes successfully, one or more routes are created and can be seen in the GUI in the top pane of the *Routes View* or by typing:

```
print summary
```

6.1 Multiple Route Creation

In many cases, for a single analysis, only a single route will be created. However, in the general case, for a single analysis, multiple routes can be created. This can occur in a number of different situations.

6.1.1 Endpoints Exist On Multiple xCORE Tiles

Consider the code in Figure 11. Analyzing from a to b produces two routes, one for each tile it is found on.

```
port p = XS1_PORT_1A ;
void f() {
    int value ;

    # pragma endpoint "a"
    p :> value ;

    # pragma endpoint "b"
    p :> value ;
}

int main () {
    par {
        on tile [0] : f();
        on tile [1] : f();
    }
    return 0;
}
```

Figure 11: Routes on multiple tiles.

6.1.2 From Endpoint Can Be Reached In Multiple Ways

Consider the code in Figure 12. Analyzing from a to b in the following code will produce two routes because there are two instances of the function `f`.

```
port p = XS1_PORT_1A ;
void f() {
    # pragma endpoint "a"
    p :> value ;
    }

int main () {
    f ();
    f ();

    # pragma endpoint "b"
    p :> value ;
    return 0;
}
```

Figure 12: Multiple instances of a function.

6.1.3 Route Specific Exclusions Exist

Consider the code in Figure 13. Setting an exclusion on `a1,excl` and analyzing function `a` causes two routes to be created. This is because there are two calls to function `a`, but the exclusion is only relevant for one of the calls so the two routes are different.

```
int a( int i) {
    if (i == 6) {
        # pragma xta label " excl "
        ++i;
    }
    return i;
}

int main () {
    # pragma xta call "a1"
    a (4);
    # pragma xta call "a2"
    a (5);
    return 0;
}
```

Figure 13: Route specific exclusions.

6.1.4 Route Specific Branches Exist

Consider the code in Figure 14. Setting a branch from `CALL:calltest1,source1` to `target1` and analyzing function `test` will cause two routes to be created. This is because there are two calls to function `test`, but the branch is only relevant for one of the calls thus the two routes will be different.

```
. type main , @function
. globl main
main :
    entsp 1
calltest1 :
    bl test
calltest2 :
    bl test
    retsp 1

. type test , @function
. globl test
test :
source1 :
    bau r1
target1 :
    retsp 0
```

Figure 14: Route specific branches

6.1.5 Looppoint Within Multiple Loops

Consider the code in Figure 15. Analyzing a loop using the `looppoint` as a reference will cause two routes to be created. This is because the reference exists within two different loops.

```
int g() {
    # pragma xta label " looppoint "
    return 1;
}

void a( int loopCount ) {
    int i = 0;
    while (i < loopCount ) {
        i += g ();
    }
}

void b( int loopCount ) {
    int i = 0;
    while (i < loopCount ) {
        i += g ();
    }
}
```

Figure 15: Looppoint within multiple loops.

7 When Code Analysis Fails

IN THIS CHAPTER

There are cases where analysis fails and no routes are created.

7.1 Endpoint Not Found

Endpoints are invalid if they fail to map to an instruction address. This can be due to a number of reasons.

7.1.1 Invalid Reference

The reference used in the analysis cannot be resolved to a valid PC. This can be due to having made a mistake in the spelling of a label or endpoint, or because of a compiler optimization has eliminated the code being timed.

The *Info View* of xTIMEcomposer Studio shows which endpoints, labels and calls are available in the binary.

On the console the available endpoints, labels and calls can be listed with:

- `list endpoints`
- `list srclabels`
- `list calls`

7.1.2 Not Found On Active Tiles

The user can configure which of the xCORE tiles in the binary are taken into account when performing analysis. The endpoint reference may not be found on the current subset of tiles being analyzed.

To fix this ensure that the relevant tiles are active for the endpoint (see §4.8).

7.1.3 Endpoint In Data Section

The tool only decodes instructions from the executable sections. Therefore, if an endpoint maps to a data section it will not be found. This is usually due to an error in an assembler file so that code is not placed in a `.text` section.

To fix this place a `.text` directive before the code.

7.2 Out Of Memory

There are cases where the tool can run out of memory when analyzing the requested route or function. This is due to the complexity of the route being analyzed. This can be resolved by either restricting the route (using exclusions) or increasing the JVM size.

7.2.1 Restrict The Route

Generally, when the tool runs out of memory it is because the user is timing code which is not restricted to the code they intended to time. In these cases add *exclusions* to reduce the complexity of the routes and therefore the memory usage of the tool.

7.2.2 Increase The JVM Size

To increase the amount of memory available to the Java Virtual Machine, change the JVM_ARGS environment variable. The default value for xTIMEcomposer Studio is 1024MB. This is configured through the `<tools>/xtimecomposer_bin/xtimecomposer.ini` file.

For the command-line tools the `<tools>/bin/xtimecomposericon.bat` should be edited to configure the JVM size. Adding the following line would give the same memory size as xTIMEcomposer Studio uses.

```
JVM_ARGS=-Xmx1024m
```

8 When Code Fails To Time

IN THIS CHAPTER

There are cases when the analysis completes, i.e. routes are created, but the worstcase time for the routes cannot be determined. The following sections describe the main causes of these failures and how to diagnose and resolve them.

8.1 Unresolved

When the XTA tool finds a branch instruction whose branch target is unknown it has to report that the time is **Unresolved**.

In order to determine which instruction is causing the issue the tool can be instructed to print an instruction trace. This will show the worst-case path ending with the unresolved instruction.

In xTIMEcomposer Studio, a trace can be printed by right clicking on the route in the top panel of the *Routes View* and selecting **Trace in console**.

On the console, type:

```
print trace -
```

To resolve this issue the XTA tool needs to be told the target of the branch instruction. This is done through either the `.xtabranch` assembler directive or the console command:

```
add branch <from BRANCH> [<to INSTRUCTION>]+
```

8.2 Recursion

The tool does not support timing of recursive functions. The tool reports that the path contains recursion and the timing will therefore fail.

The location at which the recursion occurs can be determined by printing the instruction trace as for the case of unresolved branches.

Since this is a current limitation of the tool it is not possible to time this path. If the code has to be timed then it should be re-factored into an iterative function. If the recursive path is not of interest then it can be excluded so that the rest of the route can be timed.

8.3 Infinite

If the tool encounters an infinite loop (for example `while(1)`) it reports that the timing is infinite.

To determine where the infinite loop is use the console command:

```
print structure <route id>
```

or examine the route structure in the lower panel of the *Routes View* or the *Structure* tab of the *Visualizations View*.

Since the infinite loop never terminates it cannot be timed. The tool can time how long it takes to execute each iteration of the loop. To time the loop, analyze a path with the *to* and *from* endpoints set to the same point in the loop, or use the analyse loop functionality.

8.4 Illegal

If an illegal instruction is found in a path then the time cannot be determined. The location of the illegal instruction can be found by printing a worst-case trace of the route.

It is impossible for the tool to time illegal instructions. Therefore, in order to get a valid timing result the path with the illegal instruction must be excluded.

9 Automating The Process

IN THIS CHAPTER

Once the XTA has been used interactively to create the timing critical routes, the tool can be automated to ensure that new versions of an application still meet the timing requirements.

9.1 Creating a Script

The first step to automating the process is creating a timing script. The script can either be generated by the tool or written by hand.

9.1.1 Generating A Script

Once the timing critical sections of code have been timed, refined and their timing requirements set, a script can be generated. The script will re-create the current set of routes.

Tool Creating A Script

In the GUI press the **Generate Script** button on the toolbar. This brings up a dialog showing the endpoints, labels and calls which will be used to create the script. The tool attempts to make all references portable. In order to do this it inserts the necessary *pragmas* into the source.

The names of the pragmas and whether or not they are used can be configured through this dialog. The name of the script to be created is also configured through this dialog.

If creating a script within an existing project then the script is automatically run on future compilations.

If the script creation process has modified the source (e.g. by inserting pragmas), then the relevant binary must be rebuilt before the script can be successfully executed.

9.1.2 Writing A Script

It is also possible to write a script by hand. In this case the user must insert `pragmas` into the source code where required to make the script portable.

The script file is a sequence of XTA console commands. Each one on a separate line. Any line starting with the # symbol is considered a comment.

It is recommended not to put a `load` or `exit` command in the script. These commands should be done at the time of calling the script.

⚠ XTA scripts must use the `.xta` extension in order to be used by the compiler and understood correctly by xTIMEcomposer Studio.

9.2 Running a Script

Scripts can be run in a number of different ways, either in xTIMEcomposer Studio or on the command-line.

9.2.1 During Compilation

In xTIMEcomposer Studio `.xta` scripts are automatically added to the compiler flags for compilation. On the command line the `.xta` scripts must be passed to the compiler manually. By default, timing failures are treated as warnings and syntax errors in the script as errors.

To treat timing failures as errors, add the following to the compiler arguments:

```
-Werror=timing
```

In order to treat script syntax errors as warnings, add the following to the compiler arguments:

```
-Wno-error=timing-syntax
```

9.2.2 Running From Within xTIMEcomposer Studio

There are two ways in which an `.xta` script can be executed from within xTIMEcomposer Studio.

If the binary was loaded into the XTA via a *Time Configuration*, then an XTA script can be specified in the configuration. This has the effect of running the specified script on the loaded binary whenever a new timing session is started. Note: This will also rebuild the binary if required, thus ensuring that the script is run on the most up to date version of the binary.

Alternatively scripts can be sourced from within the *xTIMEcomposer Timing Perspective* by clicking on the **Run Script** icon in the toolbar. This will run the script command on the binary that is currently loaded in the XTA.

9.2.3 Batch Mode

The XTA tool can be run in batch mode. It takes command-line arguments and interprets them as XTA commands. For example, to run an XTA script (`script.xta`) on a binary (`test.xe`) use:

```
xta -load test.xe -source script.xta -exit
```

Note: the '-' character is used as a separator between commands.

9.3 Embedding Commands Into Source

The tool supports the ability to embed commands into source code. A command is embedded into the source using a command pragma. For example,

```
#pragma xta command "print summary"
```

All commands embedded into the source are run every time the binary is loaded into the XTA. Commands are executed in the order they occur in the file, but the order between commands in different source files is not defined.

Pragmas are only supported in XC code. See §12.20 for further information.

9.4 Advanced Scripting Via the Jython Interface

The XTA supports the writing of scripts using the Jython language (an implementation of Python running on the Java virtual machine). XTA Jython scripts must have the extension `.py`. They can be executed in the same way as command based XTA scripts. From within Jython, XTA features are made available though the globally accessible *xta* object. See Figure 16 for an example script. This scripts loads the binary `test.xe` into the XTA and analyzes the function `functionName`. It then sets a loop count on each of the resulting routes and finally, prints the best and worst case times for each.

The interface to the global xta object is as follows:

9.4.1 Load Methods

```
void load(String fileName) throws Exception
```

9.4.2 Route Creation/Deletion Methods

```
List<Integer> analyzeFunction(String functionName) throws Exception
List<Integer> analyzeEndpoints(String fromRef, String toRef) throws Exception
List<Integer> analyzeLoop(String loopRef) throws Exception
void removeRoute(int routeId) throws Exception
```

```
import sys
import java

try :
    xta . load (" test .xe");

except java . lang . Exception , e:
    print e. getMessage ()

try :
    ids = xta . analyzeFunction (" functionName ");

    for id in ids :
        xta . setLoop (id , " loopReference ", 10)

    for id in ids :
        print xta . getRouteDescription (id),
        print xta . getWorstCase (id , "ns"),
        print xta . getBestCase (id , "ns")

except java . lang . Exception , e:
    print e. getMessage ()
```

Figure 16: Example of an XTA Jython script.

9.4.3 Add/Remove Methods

```
void addTile(String tileReference) throws Exception
void removeTile(String tileReference) throws Exception
Collection<String> getTiles() throws Exception

void addExclusion(String ref) throws Exception
void removeExclusion(String ref) throws Exception
Collection<String> getExclusions() throws Exception

void addBranch(String fromRefString, Collection<String> toRefStrings)
throws Exception
void removeBranch(String fromRefString, Collection<String> toRefStrings)
throws Exception
Collection<String> getBranches() throws Exception
Collection<String> getBranchTargets(String branch) throws Exception

void addLoop(String ref, long iterations) throws Exception
void removeLoop(String ref) throws Exception
Collection<String> getLoops() throws Exception

void addLoopPath(String ref, long iterations) throws Exception
void removeLoopPath(String ref) throws Exception
Collection<String> getLoopPaths() throws Exception

void addLoopScope(String ref, boolean absolute) throws Exception
void removeLoopScope(String ref) throws Exception
Collection<String> getLoopScopes() throws Exception

void addInstructionTime(String ref, double value, String units)
throws Exception
void removeInstructionTime(String ref) throws Exception
Collection<String> getInstructionTimes() throws Exception

void addFunctionTime(String ref, double value, String units)
throws Exception
void removeFunctionTime(String ref) throws Exception
Collection<String> getFunctionTimes() throws Exception

void addPathTime(String fromRef, String toRef, double value, String units)
throws Exception
void removePathTime(String fromRef, String toRef) throws Exception
Collection<String> getPathTimes() throws Exception
```

9.4.4 Set Methods

```
void setRequired(int routeId, double value, String units) throws Exception
void setFunctionTime(int routeId, String refString, double value,
String units) throws Exception
void setPathTime(int routeId, String fromRef, String toRef, double value,
String units) throws Exception
void setInstructionTime(int routeId, String refString, double value,
String units) throws Exception
void setLoop(int routeId, String refString, long iterations)
throws Exception
void setLoopPath(int routeId, String refString, long iterations)
throws Exception
void setLoopScope(int routeId, String refString, boolean absolute)
throws Exception
void setExclusion(int routeId, String refString) throws Exception
```

9.4.5 Get Methods

```
double getRequired(int routeId, String units) throws Exception
double getWorstCase(int routeId, String units) throws Exception
double getBestCase(int routeId, String units) throws Exception
List<String> getWarnings(int routeId) throws Exception
List<String> getErrors(int routeId) throws Exception
List<Integer> getRouteIds() String getRouteDescription(int routeId)
throws Exception
```

9.4.6 Config Methods

```
void configCores(String tileReference, int numCores) throws Exception
void configFreq(String nodeId, double tileFrequency) throws Exception
```

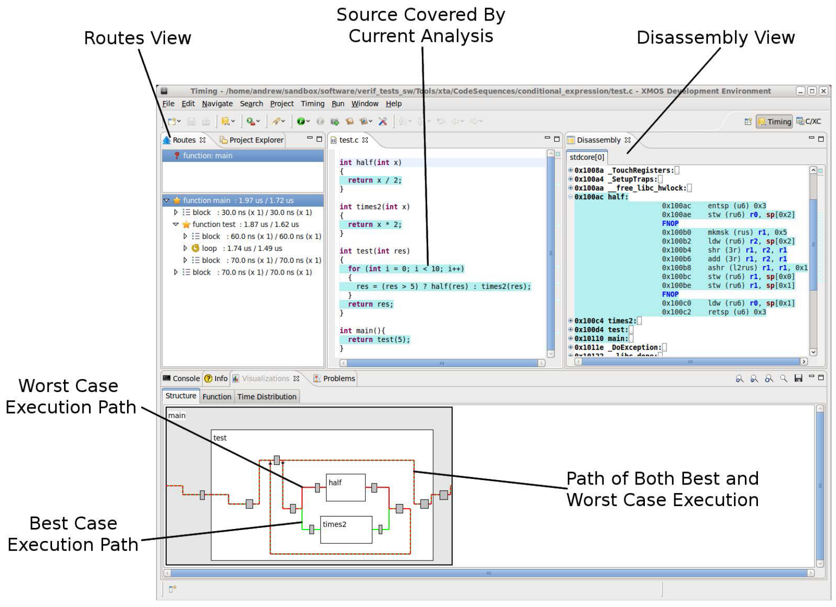
10 GUI Reference

IN THIS CHAPTER

The XTA graphical user interface is provided as part of the XMOS Development Environment. The *xTIMEcomposer Timing Perspective* can be opened by selecting a binary in the *Project Explorer* and clicking the **Time** button on the toolbar.

10.1 Views

The *xTIMEcomposer Timing Perspective* consists of a set of editors, which display the code and a set of supporting views, which highlight timing related aspects of the code.



Routes View
Source Covered By Current Analysis
Disassembly View
Worst Case Execution Path
Best Case Execution Path
Path of Both Best and Worst Case Execution
Timing - /home/andrew/sandbox/software/verif_tests_sw/Tools/xta/CodeSequences/conditional_expression/test.c - XMOS Development Environment
File Edit Navigate Search Project Timing Run Window Help
Timing
C/XC
Routes
Project Explorer
test.c
Disassembly
stdcore[0]
Console
Info
Visualizations
Problems
Structure
Function
Time Distribution
main
test
half
times2

```
function: main

function main  : 1.97 us / 1.72 us
  block   : 30.0 ns (x 1) / 30.0 ns (x 1)
  function test  : 1.87 us / 1.62 us
    block   : 60.0 ns (x 1) / 60.0 ns (x 1)
    loop   : 1.74 us / 1.49 us
    block   : 70.0 ns (x 1) / 70.0 ns (x 1)
  block   : 70.0 ns (x 1) / 70.0 ns (x 1)
```

```
int half(int x)
{
  return x / 2;
}

int times2(int x)
{
  return x * 2;
}

int test(int res)
{
  for (int i = 0; i < 10; i++)
  {
    res = (res > 5) ? half(res) : times2(res);
  }
  return res;
}

int main(){
  return test(5);
}
```

```
0x1008a _TouchRegisters:
0x100a4 _SetupTraps:
0x100aa __free_libc_hwlock:
0x100ac half:
                    0x100ac    entsp (u6) 0x3
                    0x100ae    stw (ru6) r0, sp[0x2]
                    FNOP
                    0x100b0    mkmsk (rus) r1, 0x5
                    0x100b2    ldw (ru6) r2, sp[0x2]
                    0x100b4    shr (3r) r1, r2, r1
                    0x100b6    add (3r) r1, r2, r1
                    0x100b8    ashr (l2rus) r1, r1, 0x1
                    0x100bc    stw (ru6) r1, sp[0x0]
                    0x100be    stw (ru6) r1, sp[0x1]
                    FNOP
                    0x100c0    ldw (ru6) r0, sp[0x1]
                    0x100c2    retsp (u6) 0x3
0x100c4 times2:
0x100d4 test:
0x10110 main:
0x1011e _DoException:
```

Figure 17: xTIMEcomposer Timing Perspective.

10.1.1 Routes View

This view is split into two parts. The upper panel consists of a list of the currently analyzed routes. The lower panel displays the structure, in a tree format, of the currently selected route (i.e. the route that is selected in the upper panel).

The view shows the best and worst case timing values for a route.

10.1.2 Disassembly View

This view shows the disassembled instructions for the currently loaded application.

There is one tab per xCORE tile found in the binary.

10.1.3 Console View

This view gives the user access to the XTA console.

10.1.4 Visualizations View

Contains multiple different visualizations, each in its own tab:

- Structure Displays the static structure for the currently selected route. This is the pictorial equivalent of the lower panel of the *Routes View* which shows the same structural information in a tree based format.
- Function Displays the dynamic behavior of either the worst or base case path though the currently selected route. In many ways, this is the pictorial equivalent of the trace in console functionality.
- Distribution Displays the shape of the distribution of the time taken for each of the paths through the currently selected route.

10.1.5 Info View

This view displays the current exclusion list, the unknowns/knowns for the currently selected route, the current defines and the contents of the XTA specific binary sections.

10.1.6 Problems View

Any errors/warnings detected during the operation of the tool are logged here.

10.1.7 Files View

The tool does support timing of binaries that have been built outside of an xTIME-composer project. In order to open a binary which is not part of an xTIMEcomposer project open the *Files View*.

Choose **Windows** ▶ **Show View** ▶ **Other** ▶ **Timing** ▶ **Files**.

The *Files View* has an **Open File** button which allows any binary to be opened. Once the binary is open the *Files View* shows a list of the source files for the binary opened.

10.2 Editors

In the *xTIMEcomposer Timing Perspective* the source editors are annotated with XTA information. They show where valid endpoints are located in the vertical ruler bar to the left of the editor. When a route is selected they show through which source lines the path passes.

⚠ The annotations in source editors are only available if the binary is newer than the source. This is to prevent confusion of incorrectly annotating a source which no longer matches the binary.

When in the *xTIMEcomposer Timing Perspective* all editors corresponding to the binary which has been opened are changed to be read-only to prevent them from being accidentally modified.

11 Working With Assembly

IN THIS CHAPTER

When writing programs in assembly it is still possible to label code to make it portable using assembler directives.

11.1 Assembly Directives

The XMOS Timing Analyzer directives add timing metadata to ELF sections.

- `xtabranch` specifies a comma-separated list of locations that may be branched to from the current location.
- `xtacall` marks the current location as a function call with the specified label.
- `xtaendpoint` marks the current location as an endpoint with the specified label.
- `xtalabel` marks the current location using the specified label.
- `xtacorestart` specifies that a logical core may be initialized to start executing at the current location.
- `xtacorestop` specifies that a logical core executing the instruction at the current location will not execute any further instructions.

The `xtacall`, `xtaendpoint`, `xtalabel` directives are intended for use by the compiler only. They are used to link lines of source code with assembly instructions. All other XTA functionality provided by these directives (timing, exclusions) should be possible through the use of labels in the assembly code.

Strings used by the XTA for `xtacall`, `xtaendpoint` and `xtalabel` must not contain spaces.

11.2 Branch Table Example

If a branch table is written in assembly, the user must add branch target information in order for the XTA to be able to analyze the assembly properly . This information is given in the form of a `.xtabranch` directive. For example, consider the code in Figure 18.

```
. type f, @function
. globl f
f:
    entsp 1
. xtabranch Ltarget1 , Ltarget2 , Ltarget3
    bru r0
Ltarget1 :
    bl taskA
    retsp 1
Ltarget2 :
    bl taskB
    retsp 1
Ltarget3 :
    bl taskC
    retsp 1
```

Figure 18: Setting branch targets

The XTA is not able to determine where the `bru` instruction will branch to because it is branching off a register value which is an argument to `main`. With the directive the XTA can consider the `bru` instruction to have the three targets (`Ltarget1`, `Ltarget2`, `Ltarget3`) and the XTA can successfully time the function.

11.3 Core Start/Stop Example

By default the XTA, assumes that the initial logical core starts executing at the RAM base. However, if the user adds another core in assembly then for the XTA to know that the code is reachable the user needs to insert `.xtacorestart` and `.xtacorestop` directives.

For example, consider the code in Figure 19.

With the `xtacorestart` and `xtacorestop` directives the XTA knows that the code after the label `secondCore` is reachable and hence can be analyzed.

```
. type main , @function
. globl main
main :
    getr r1 , XS1 \ _RES \ _TYPE \ _CORE
    ldap r11 , secondCore
    init t[r1 ]:pc , r11
    start t[r1]
    ldc r1 , 0
loop :
    bf r1 , loop
    retsp 0

secondCore :
. xtacorestart
    ldc r0 , 1
    tsetmr r1 , r0
. xtacorestop
    freet
```

Figure 19: Setting core start and stop points.

12 Console Command Reference

IN THIS CHAPTER

12.1 add

`add branch <from BRANCH> [<to INSTRUCTION>]+`

Adds the given from/to references to the branches list

`add tile <tile id|*>`

Add xCORE tile to active set

`add exclusion <ANY>`

Adds the given reference to the list of exclusions

`add functiontime <FUNCTION> <value> <MODE>`
Adds the given function time to the list of defines

`add instructiontime <ENDPOINT> <value> <MODE>`
Adds the given instruction time to the list of defines

`add loop <ANY> <iterations>`
Adds the given loop count define to the list of defines

`add looppath <ANY> <iterations>`
Adds the given loop path count define to the list of defines

`add loopscope <ANY> <SCOPE>`
Adds the given loop scope define to the list of defines

`add pathtime <from ENDPOINT> <to ENDPOINT> <value> <MODE>`
Adds the given path time to the list of defines

12.2 analyze

`analyze endpoints <from ENDPOINT> <to ENDPOINT>`
Analyzes between the specified endpoints

`analyze function <FUNCTION>`
Analyzes the given function

`analyze loop <ANY>`
Analyzes the given loop

12.3 config

`config case <best/worst>`
Sets the case (currently: worst)

`config Ewarning <on/off>`
Treats errors as warnings or not (currently: off)

`config freq <node id> <tile frequency>`
Sets the operating frequency in MHz for the given node

`config from <ENDPOINT>`
Sets the from endpoint

`config looppoint <ANY>`
Sets the loop point

`config scale <true/false>`
: Configures whether results are scales (currently: true)

`config srcpaths <paths>`
: Sets the (semicolon separated) source search path

`config cores <tile id> <num cores>`
: Sets number of cores currently executing for the given tile

`config timeout <seconds>`
: Sets the tools timeout on load

`config Terror <on/off>`
: Treat timing failures as errors or not (currently: on)

`config to <ENDPOINT>`
: Sets the to endpoint

`config verbosity <level>`
: Sets the tool verbosity level (range: -10 -> +10, default: 0)

`config Werror <on/off>`
: Treats warnings as errors or not (currently: off)

12.4 clear

`clear()`
: Clears the screen (GUI mode only)

12.5 debug

`debug dumpactiveexclusions()`
: Dumps a list of PCs that the exclusions have resolved to

`debug dumpcachedfunction <FUNCTION>`
: Dumps the cached function structure

`debug dumpcallgraph()`
: Dumps the call graph for all tiles in dot (graphviz) format

`debug dumpcontrolflow <FUNCTION>`
: Dumps the control flow graph for the given function in dot (graphviz) format

`debug dumpmanual()`
: Dumps the console reference chapter of the manual in tex format

`debug dumpstacknodes <REFERENCE>`
: Dumps the stack nodes for the given reference

`debug dumpunresolvedinstructions()`
: Dumps a list of instructions that are unresolved

`debug verifyreference <ANY>`
: Verifies the existance of the given reference

`debug frompoints()`
: Displays the from endpoints currently configured

`debug topoints()`
: Displays the to endpoints currently configured

`debug instructiontime <route id> <node id>`
: Displays the instruction time set for the given node in the given route

`debug loop <route id> <node id>`
: Displays the loop iterations set for the given node in the given route

`debug looppath <route id> <node id>`
: Displays the loop path iterations set for the given node in the given route

`debug loopscope <route id> <node id>`
: Displays the loop scope set for the given node in the given route

`debug listglobalreferences <ANY>`
: Lists all the matching references for the given reference on the global tree

`debug listroutereferences <route id> <ANY>`
: Lists all the matching references for the given reference on the given route

`debug memusage()`
: Displays the current memory usage for the JVM

`debug getmemthreshold()`
: Displays the current memory usage threshold

`debug setmemthreshold <threshold>`
: Sets the memory threshold to the given value (0.0 - 1.0)

12.6 echo

`echo "text"`
: Prints the text to the console

12.7 exit

`exit()`
: Quits the application

12.8 help

`help [command|command subcommand|option]`

Displays help message for the given arguments

12.9 history

`history()`

Displays the command history

12.10 load

`load <xe file>`

Loads the given XMOS executable file

12.11 list

`list allcalls()`

Lists all the possible locations for calls

`list allendpoints()`

Lists all the possible locations for endpoints

`list branches [route id]`

Lists the branches - optionally for the specified route

`list calls()`

Lists the calls

`list tiles()`

Lists the active xCORE tiles

`list endpoints()`

Lists the endpoints

`list exclusions [route id]`

Displays the exclusions - optionally for the specified route

`list functions()`

Lists the functions in the loaded application

`list functiontimes()`

Displays the function time defines

`list instructiontimes()`

Displays the instruction time defines

`list knowns <route id>`
: Displays the list of knowns set for the given route

`list labels()`
: Lists the labels

`list loops()`
: Displays the loop defines

`list looppaths()`
: Displays the loop path defines

`list loopscopes()`
: Displays the loop scope defines

`list pathtimes()`
: Displays the path time defines

`list sources()`
: Lists the source files

`list srccommands()`
: Displays the command list embedded in the loaded executable

`list srcloops()`
: Displays the loop counts embedded in the loaded executable

`list srclabels()`
: Lists the source labels

`list allsrclabels()`
: Lists all the possible locations for source labels

`list corestartpoints()`
: Lists the logical core start points

`list corestoppoints()`
: Lists the logical core stop points

`list unknowns <route id>`
: Displays the list of unknowns for the given route

12.12 print

`print summary()`
: Shows routes summary (verbosity -2|-1|0)

`print structure <route id> [node id]`

Displays the structure for given route/node (verbosity 0|1)

`print asm <route id> [node id]`
Displays annotated assembly for the given route/node

`print src <route id> [node id]`
Displays annotated source file(s) for given route/node

`print trace <route id> [node id]`
Displays instruction trace for the worst case path of the given route/node

`print routeinfo <route id>`
Shows detailed information for the given route

`print nodeinfo <route id> <node id>`
Shows detailed information for the given node in the given route

`print warnings()`
Prints all timing warnings

`print distribution <route id> [node id]`
Displays time distribution for the given route/node

12.13 pwd

`pwd()`
Displays the current working directory

12.14 remove

`remove branch <from BRANCH|*> [<to INSTRUCTION|*>]+`
Removes the given from/to references from the branches list

`remove tile <tile id|*>`
Removes xCORE tile from active set

`remove exclusion <ANY|*>`
Removes the given reference (or all if '*') from the list of exclusions

`remove functiontime <FUNCTION|*>`
Removes the given functon time from the list of defines

`remove instructiontime <ENDPOINT|*>`
Removes the given instruction time from the list of defines

`remove loop <ANY|*>`
Removes the given loop count define to the list of defines

`remove looppath <ANY|*>`
: Removes the given loop path count define to the list of defines

`remove loopscope <ANY|*>`
: Removes the given loop scope define to the list of defines

`remove pathtime <from ENDPOINT|*> <to ENDPOINT|*>`
: Removes the given path time from the list of defines

`remove route <route id>`
: Removes the route with the given id from the current analysis

12.15 scripter

`scripter disable <ANY>`
: Disables a mapping

`scripter dump()`
: Dumps script which represents the current state - also embeds active pragmas into source

`scripter embed <filename>`
: Embeds the script into the designated file - also embed active pragmas into source

`scripter enable <ANY>`
: Enables a mapping

`scripter listrefs()`
: Lists all references which will be used in the script

`scripter rename <ANY> <TO_NAME>`
: Renames a mapping

12.16 set

`set exclusion <route id> <ANY>`
: Sets an exclusion on the given reference

`set functiontime <route id> <FUNCTION> <value> <MODE>`
: Sets timing requirement for the given function on the given route

`set instructiontime <route id> <ENDPOINT> <value> <MODE>`
: Sets the time taken for the instruction at the given pc

`set loop <route id> <ANY> <iterations>`
: Sets the number of iterations for the loop identified

`set looppath <route id> <ANY> <iterations>`

Sets the number of iterations for the path identified

`set loopscope <route id> <ANY> <SCOPE>`

Sets the scope of the referenced loop

`set pathtime <route id> <from ENDPOINT> <to ENDPOINT> <value> <MODE>`

Sets timing requirement for the given path on the given route

`set required <route id> <value> <MODE>`

Sets the maximum allowed time taken for the given route

12.17 source

`source <file name> [args]`

Sources the given script file

12.18 status

`status()`

Displays current status

12.19 version

`version()`

Displays the version information

12.20 Pragmas

`#pragma xta label "name"`

Provides a label that can be used to specify timing constraints.

`#pragma xta endpoint "name"`

Specifies an endpoint. It may appear before an input or output statement.

`#pragma xta call "name"`

Defines a label for a (function) call point. Use to specify a particular called instance of a function. For example, if a function contains a loop, the iterations for this loop can be set to a different value depending on which call point the function was called from.

`#pragma xta command "command"`

Allows XTA commands to be embedded into source code. All commands are run every time the binary is loaded into the XTA. Commands are executed in the order they occur in the file, but the order between commands in different source files is not defined.

`#pragma xta loop "integer"`

Applies the given loop XTA iterations to the loop containing the pragma.

12.21 Timing Modes

The available timing modes are:

`ns()`
: nanoseconds

`us()`
: microseconds

`ms()`
: milliseconds

`MHz()`
: megahertz

`KHz()`
: kilohertz

`Hz()`
: hertz

`cycles()`
: The core cycle count is the number of scheduled slots that the logical core required to perform the sequence. The relationship between core cycles and time is a function of the number of cores currently running and the xCORE tile frequency.

12.22 Loop Scopes

Supported values for scope are:

`relative/r()`
: Iteration number propagates to the enclosing path (Default)

`absolute/a()`
: Absolute number of iterations

12.23 Reference Classes

12.23.1 FUNCTION

`FunctionPc()`
: Raw program counter specified in the format: 0x*

`Function()`
: Any function

12.23.2 BRANCH

`EndpointPC()`
: Raw program counter specified in the format: 0x*

`CallPc()`
: Call specified in the format: 0x*

`CallFileLine()`
: Call specified in the format: ‘file name:line number’

`Call()`
: Call specified using the source level pragma mechanism

`Label()`
: Any source or assembly level symbol defined with respect to an executable section

`CallLabel()`
: Any source or assembly level symbol defined with respect to an executable section

12.23.3 INSTRUCTION

`EndpointPC()`
: Raw program counter specified in the format: 0x*

`FunctionPc()`
: Raw program counter specified in the format: 0x*

`Function()`
: Any function

`Label()`
: Any source or assembly level symbol defined with respect to an executable section

12.23.4 ENDPOINT

`EndpointPC()`
: Raw program counter specified in the format: 0x*

`EndpointFileLine()`
: Endpoint specified in the format: ‘file name:line number’

`Endpoint()`
: Endpoint specified using the source level pragma mechanism

`CallPc()`
: Call specified in the format: 0x*

`CallFileLine()`
: Call specified in the format: ‘file name:line number’

`Call()`
: Call specified using the source level pragma mechanism

`Label()`
: Any source or assembly level symbol defined with respect to an executable section

`CallLabel()`
: Any source or assembly level symbol defined with respect to an executable section

12.23.5 ANY

`SrcLabelPc()`
: Raw program counter specified in the format: 0x*

`EndpointPC()`
: Raw program counter specified in the format: 0x*

`EndpointFileLine()`
: Endpoint specified in the format: ‘file name:line number’

`Endpoint()`
: Endpoint specified using the source level pragma mechanism

`CallPc()`
: Call specified in the format: 0x*

`CallFileLine()`
: Call specified in the format: ‘file name:line number’

`Call()`
: Call specified using the source level pragma mechanism

`SrcLabelFileLine()`
: Source label specified in the format: ‘file name:line number’

`SrcLabel()`
: Source label specified using the source level pragma mechanism

`Label()`
: Any source or assembly level symbol defined with respect to an executable section

`CallLabel()`
: Any source or assembly level symbol defined with respect to an executable section

12.23.6 FUNCTION_WITH_EVERYTHING

`EverythingReference()`
: Matches everything: '*'

`FunctionPc()`
: Raw program counter specified in the format: 0x*

`Function()`
: Any function

12.23.7 BRANCH_WITH_EVERYTHING

`EverythingReference()`
: Matches everything: '*'

`EndpointPC()`
: Raw program counter specified in the format: 0x*

`CallPc()`
: Call specified in the format: 0x*

`CallFileLine()`
: Call specified in the format: 'file name:line number'

`Call()`
: Call specified using the source level pragma mechanism

`Label()`
: Any source or assembly level symbol defined with respect to an executable section

`CallLabel()`
: Any source or assembly level symbol defined with respect to an executable section

12.23.8 INSTRUCTION_WITH_EVERYTHING

`EverythingReference()`
: Matches everything: '*'

`EndpointPC()`
: Raw program counter specified in the format: 0x*

`FunctionPc()`
: Raw program counter specified in the format: 0x*

`Function()`
: Any function

`Label()`
: Any source or assembly level symbol defined with respect to an executable section

12.23.9 ENDPOINT_WITH_EVERYTHING

`EverythingReference()`
: Matches everything: '*'

`EndpointPC()`
: Raw program counter specified in the format: 0x*

`EndpointFileLine()`
: Endpoint specified in the format: 'file name:line number'

`Endpoint()`
: Endpoint specified using the source level pragma mechanism

`CallPc()`
: Call specified in the format: 0x*

`CallFileLine()`
: Call specified in the format: 'file name:line number'

`Call()`
: Call specified using the source level pragma mechanism

`Label()`
: Any source or assembly level symbol defined with respect to an executable section

`CallLabel()`
: Any source or assembly level symbol defined with respect to an executable section

12.23.10 ANY_WITH_EVERYTHING

`EverythingReference()`
: Matches everything: '*'

`SrcLabelPc()`
: Raw program counter specified in the format: 0x*

`EndpointPC()`
: Raw program counter specified in the format: 0x*

`EndpointFileLine()`
: Endpoint specified in the format: 'file name:line number'

`Endpoint()`
: Endpoint specified using the source level pragma mechanism

`CallPc()`
: Call specified in the format: 0x*

`CallFileLine()`
: Call specified in the format: 'file name:line number'

`Call()`
: Call specified using the source level pragma mechanism

`SrcLabelFileLine()`
: Source label specified in the format: 'file name:line number'

`SrcLabel()`
: Source label specified using the source level pragma mechanism

`Label()`
: Any source or assembly level symbol defined with respect to an executable section

`CallLabel()`
: Any source or assembly level symbol defined with respect to an executable section

13 Tool Limitations

Limitations of the current XTA are:

- Recursion: the tool does not process any program which contains recursion.
- Par: the tool only times one logical core at a time. The code has to be timed on each core manually in order to determine the overall worst-case timing. The tool gives a warning if code causing parallel execution is detected.
- Limited support in C/C++ compilers: the compilers support XTA *source labels*, *call labels* and *endpoints* in XC code only, not in C or C++.
- Loop iterations: the compiler only performs loop analysis with optimizations turned on (`-O1` and greater). Therefore, with optimization turned off and for the cases where the compiler cannot determine a static loop count, the user has to specify them.

14 Code Reference Grammar

A code reference constructed of a *back trail* and a base reference of the form:

```
code-ref        ::=  back-trail base-ref

back-trail      ::=  base-ref
                 |   base-ref , back-trail

base-ref        ::=  pc-ref
                 |   label-ref
                 |   function-ref
                 |   endpoint-ref
                 |   srclabel-ref
                 |   call-ref

pc-ref          ::=  pc-class hex-constant

label-ref       ::=  label-class label-string

function-ref    ::=  function-class function-name
                 |   function-class hex-constant

endpoint-ref    ::=  endpoint-class file-line
                 |   endpoint-class endpoint-label
                 |   endpoint-class hex-constant

srclabel-ref    ::=  srclabel-class label-string

call-ref        ::=  call-class label-string
                 |   call-class hex-constant

pc-class        ::=
                 |   PC:

label-class     ::=
                 |   LABEL:

functionclass   ::=
                 |   FUNCTION:

endpointclass   ::=
                 |   ENDPOINT:
```

srclabelclass	::=	
	|	*SRCLABEL:*
call-class	::=	
	|	*CALL:*
file-line	::=	*file-name : integer-constant*