Timing Analysis for the Precision Timed ARM Processor

Subarno Banerjee

Bericht Nr. 1212
September 2012
ISSN 2192-6247
Timing Analysis for the Precision Timed ARM Processor

Subarno Banerjee

Bericht Nr. 1212
September 2012
ISSN 2192-6247

e-mail: banerjee.subarno@gmail.com

This report has been prepared by Subarno Banerjee during his internship within the IAESTE program from May to September 2012 at the research group Real-Time and Embedded Systems. Advisors: Insa Fuhrmann, Christian Motika, and Reinhard v. Hanxleden
Acknowledgement

I’m sincerely grateful to Prof. Dr. Reinhard von Hanxleden for hosting me as an intern with his research group and giving me the wonderful exposure. I thank Dr. Hugues Cassé for his continuous guidance. I’m thankful to Insa Fuhrmann and Christian Motika for their valuable feedbacks and timely support. I want to thank all members of the Real Time & Embedded Systems group for their friendly support and encouragement.

I want to thank Mr. Jan Bensien for helping me with the internship procedures. I want to thank the IAESTE buddies- Markus, Marina, Nora and Jonathan for their friendship and care.

I thank my parents for their teachings and blessings.
# Table of Contents

1. Motivation 1

2. Precision Timed ARM 1-3
   2.1 Thread Interleaved Pipeline
   2.2 Memory Hierarchy
      2.2.1 Scratchpad Memory
      2.2.2 Predictable DRAM Controller
   2.3 Timing Analysis for PTARM

3. OTAWA: Open Toolbox for Adaptive WCET Analysis 3-4

4. Deadline Instructions 4-5
   4.1 Time Deadlines
   4.2 Sync Deadlines

5. Implementation 6-14
   5.1 Adapting OTAWA for Timing Analysis on PTARM
   5.2 Using OTAWA on Code Segments

6. Future Work 15-17
   5.1 Segment Annotations
   5.2 Segment Aligning

7. Conclusion 17-18

8. References 18

## Appendix

I. PTARM Pipeline description 19
II. PTARM Memory Layout description 20
III. PTARMBBTime.h 21
IV. ptarmta.cpp 25
V. test.c 27
VI. Producer, Consumer and Observer programs 28
VII. gel.h (old) 29
VIII. gel.h 31
IX. AdressDomain.h 34
X. SegILPBuilder.h 40
XI. segta.cpp 44
XII. SegPicker.h 46
XIII. deadta.cpp 50
1. Motivation

Real-time applications often have to deliver desired timing behaviour and satisfy time constraints. The timing constraints can be guaranteed by adopting a timing predictable (PRET) architecture and using static analysis tools to compute bounds on WCET for the target architecture. This ensures predictable and repeatable timing behaviour of the application. To deliver precise timing of events, Ip and Edwards proposed a processor extension\cite{1} that allows access to cycle-accurate timers through timing instructions. The deadline instruction waits for a timer to expire and then reloads it synchronously. This allows the embedded application programmer to explicitly specify the minimum number of cycles for a certain piece of code. Although the deadline mechanism does not restrict the missing of deadlines, it can be handled by raising exceptions. Such precise cycle-accurate timing control allowed for software implementations of controllers that previously could only be implemented in hardware. Also, it relieved the programmer from the burden of synthesizing timing accurate applications by carefully padding NOPs.

In most reactive applications, such precise timing control can be used to synchronize parts of concurrent threads- shared memory accesses, loops running in lock step, etc. For such synchronization requirements, the deadline values will depend on the execution time of the code segments and the type of synchronization. It will be a tedious task for the programmer to manually compute the values for the parameters of deadline instructions. In fact for fairly simple applications, this can be a very difficult task. We believe that if the synchronization behaviour of concurrent threads is known, then the deadline values can be derived through static analysis. In this summer project, we attempt to automate the computation of the deadline values.

2. Precision Timed ARM (PTARM)

The PTARM\cite{4} is a PRET\cite{2} processor built on the ARM instruction set architecture. It is developed by the CHESS research group at University of California, Berkeley, US. The timing predictable architecture employs a thread interleaved pipeline and an exposed memory hierarchy with Scratchpad memory system. It extends the ARMv4 ISA with timing instructions.

2.1 Thread Interleaved Pipeline

The PTARM implements a five stage thread-interleaved pipeline with fine grained (at every processor cycle) thread scheduling. The pipeline fetches instructions from different threads in a round robin fashion every cycle. This removes the pipeline hazards and improves throughput and predictability. For temporal isolation of threads, the no. of threads should at least be the no. of pipeline stages. The PTARM runs four threads to slightly improve thread latencies.
For a single thread in isolation, the processor behaves as a non-pipelined non-speculative processor with frequency equal to one fourth of the actual clock frequency. For the purpose of timing analysis, we may assume that there are no inter-thread interferences. So we can perform analysis on threads in isolation.

The processor model reduces to that of a non-pipelined one with frequency $f_{\text{thread}}$ given by-

$$f_{\text{thread}} = \begin{cases} \frac{f_{\text{proc}}}{4}, & \text{for } n \leq 4 \\ \frac{f_{\text{proc}}}{n}, & \text{for } n > 4 \end{cases}$$

where $n$ is the number of threads and PTARM clock frequency $f_{\text{proc}} = 100$MHz.

### 2.2 Memory Hierarchy

The PTARM has an exposed memory hierarchy which allows for predictable and analysable memory access latencies. The memory layout is shown in figure 2.

#### 2.2.1 ScratchPad Memory

Caches are replaced by scratchpads as fast access memory. Unlike caches, scratchpads expose the memory hierarchy to the programmer. The scratchpad occupies a distinct address space in the memory map. Depending on the accessed memory address, the request goes to the scratchpad or the main memory. The memory access latency depends only on the accessed address.

All instructions are statically compiled onto the Instruction Scratchpad.
2.2.2 Predictable DRAM Controller

The PTARM interfaces a 512MB DRAM. All access to the DRAM go through the predictable DRAM controller. The controller reserves private DRAM banks for each hardware thread to remove bank access conflicts. The controller consists of a frontend and a backend. The backend translates memory access requests into DRAM commands. The frontend interfaces the backend to the pipeline and is also responsible for scheduling refreshes for the DRAM device. Depending on the alignment of the pipeline and the DRAM controller backend, the DRAM load latency is 3 or 4 thread cycles. The DRAM store latency is 1 or 2 thread cycles depending on whether a store buffer can be used or not. For conservative estimates, the upper values are used.

2.3 Timing Analysis for PTARM

Since threads running on the PTARM are temporally isolated, timing analysis can be done separately for each thread. A thread cycle is the thread’s perceived notion of cycle. One thread cycle is the time interval after which an instruction from the same thread enters the pipeline. The PTARM processor is clocked at 100MHz and runs 4 threads. So, the rate at which instructions are fetched from a thread is \( \frac{100 \text{ MHz}}{4} = 25 \text{ MHz} \). So, each thread cycle is \( \frac{1}{25 \text{ MHz}} = 40\text{ ns} \).

The following table summarizes instruction execution times in terms of thread cycles.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Processing</td>
<td>1</td>
</tr>
<tr>
<td>Branch</td>
<td>1</td>
</tr>
<tr>
<td>Software Interrupt</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Accessed Memory Region</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SPM / Boot</td>
</tr>
<tr>
<td>Load Register</td>
<td>1</td>
</tr>
<tr>
<td>Store Register</td>
<td>1</td>
</tr>
<tr>
<td>Load Multiple</td>
<td>N</td>
</tr>
<tr>
<td>Store Multiple</td>
<td>N</td>
</tr>
</tbody>
</table>

Table 1[^1]: Timing properties of PTARM

[^1]: Upper values are used for conservative estimates. Tighter estimates require modelling the DRAM controller. N is the number of registers in the register list.

3. Open Toolbox for Adaptive WCET Analysis

The **OTAWA**[^5,7] tool provides a framework for static analysis of binary code and Worst Case Execution Time (WCET) computation. It is developed by the **TRACES** research group at the University of Toulouse, France. The tool supports several popular architectures including ARM, PowerPC, SPARC, etc. The tool is adaptive in nature; so it facilitates customized analyses in a simple and elegant manner.

**OTAWA** loads the program representation into the *workspace*. The program representation varies with the phases of the analysis. The program representation is annotated with *properties* that are used to attach / retrieve specific information. A typical
OTAWA application consists of a string of code processors running in sequence. A code processor is a class that performs an analysis of the program representation and provides a feature. A feature asserts that the associated analysis has been performed on the current workspace and consequently certain properties are available. A feature is generally associated to a default code processor that can provide it. If during an analysis, a code processor is invoked that requires a feature which is currently unavailable, the default code processor is automatically invoked. The code processors chain themselves according to required features. Thus an OTAWA analysis proceeds by running processors and providing features in a dependent series.

A typical WCET analysis in OTAWA involves the following series of processors:

- CFGBuilder: The Control Flow Graphs (CFG) are built
- CFGCollector: The involved CFGs are collected
- VarAssignment: Integer Linear Programming (ILP) variables are assigned
- BasicConstraintBuilder: The ILP flow constraints are built
- TrivialBBTime: The basic block execution times are computed
- BasicObjectFunctionBuilder: The ILP object function is built
- FlowFactLoader: The flow facts / loop bounds are loaded
- WCETComputation: The WCET is computed

A feature is computed by its default processor only if it is not already available. This behaviour of the feature system can be exploited to customize the analysis. Any new code processor providing the same feature may be explicitly invoked to specialize the analysis. This gives the possibility to use different analysis methods for different phases by exchanging default processors with new processors.

4. Deadline Instructions

A deadline instruction DEAD(n) blocks until the associated deadline timer reaches zero. Then it synchronously reloads the timer with the new value ‘n’ and returns control to the next instruction.

The deadline instruction is not part of the current PTARM implementation of Berkeley. The actual timing instructions are described in [4]. It is possible to achieve similar semantics by macro usage (see [4]). It is important to understand though that the current PTARM timing instructions are based on clock timestamps instead of cycles. In the context of this report, the deadline instruction should be understood as a concept and motivation for segment matching, not as part of the PTARM processor.
4.1 Time Deadlines

The deadline instruction can be used to explicitly control the timing behaviours of parts of a program. It divides the program into constant minimum execution time units - timed blocks. The timing requirements might be fully application dependent and the parameters for time deadlines must be provided by the programmer. However, there is a need to verify the deadline values ie. deadline values cannot be less than the WCET of the timed block.

---
DEAD(10);
PORTA = 0xFF;
DEAD(12);
PORTA = 0x00;
---

Figure 3(a): The deadline instructions ensure that PORTA remains HIGH (0xFF) for exactly 10 cycles before going to LOW (0x00).

---
while(1)
{
    DEAD(10);
    PORTA ^= 0xFF;
}
---

Figure 3(b): When placed inside a loop, the deadline instruction sets the period of the loop. PORTA toggles its state after each 10 cycles.

4.2 Sync Deadlines

The deadline instructions can also be used to synchronize parts of concurrent threads. This requires certain desired behaviour and thus can be derived if the synchronization requirement is known. This divides the threads into units of synchronization - segments.

Producer

```
int main(){
    DEAD(28);
    volatile unsigned int * buf = (unsigned int*) (0x3F800200);
    unsigned int i = 0;
    for (i = 0; ; i++ ){
        DEAD(26);
        *buf = i;
    }
    return 0;
}
```

Consumer

```
int main(){
    DEAD(41);
    volatile unsigned int * buf = (unsigned int*) (0x3F800200);
    unsigned int i = 0;
    int arr[8];
    for (i =0; i<8; i++)
    {
        arr[i] = 0;
        for (i = 0; ; i++ ){
            DEAD(26);
            register int tmp = *buf;
            arr[i%8] = tmp;
        }
    }
    return 0;
}
```

Observer

```
int main(){
    DEAD(41);
    volatile unsigned int * buf = (unsigned int*) (0x3F800200);
    volatile unsigned int * fd = (unsigned int*) (0x80000600);
    unsigned int i = 0;
    for (i = 0; ; i++ ){
        DEAD(26);
        *fd = *buf;
    }
    return 0;
}
```

Figure 4[3]: The deadline instructions ensure that the Producer enters its loop 13 cycles ahead of the Consumer and Observer each time.

A sync deadline can be identified as a deadline instruction with a blank parameter; its parameter has to be derived and then written by our tool.

[5]
5. Implementation

5.1 Adapting OTAWA for Timing Analysis on PTARM

The OTAWA tool provides processors that compute basic block execution times with different levels of tightness. The most trivial computation TrivialBBTime assumes a non-pipelined processor with constant instruction execution cycles for all instruction types. The next tighter computation BBTimeSimulator uses a simulator to capture effects of pipelining instructions. For an isolated thread, the PTARM processor behaves like a non-pipelined processor, but with different execution cycles for different instruction types. So, our analysis requires a processor with tightness in between that of TrivialBBTime and BBTimeSimulator.

Secondly, although no modelling is required for the Scratchpad memory, timing analysis will require performing some value analysis to get the address accessed by a memory instruction. Based on this address value, the access request can be dispatched to the scratchpad or the DRAM. So, we write our own code processor to implement the required basic block execution time computation with address value analysis for memory instructions.

OTAWA uses an architecture description format to specify the pipeline, memory and cache architecture. We describe the PTARM architecture using the prescribed format (Appendix I & II), and use it in our computation.

We extend the otawa::BBProcessor class to write the PTARMBBTime class (Appendix III). OTAWA ensures that the four functions of a code processor are always called in the following order:

- configure()
- setup()
- processWorkSpace()
- cleanup()

For a BBProcessor, the processBB function is called iteratively for all basic blocks of all CFGs in the current workspace.

The configure() method loads the pipeline and memory architecture description from the respective XML files into OTAWA internal objects. The cleanup() method deletes the processor and memory description objects.

The processBB method computes the execution cycles of a basic block. This method is automatically called for every basic block in the workspace. The basic block execution time is initialized to zero. The method then picks instructions in the basic block one-by-one, decides their latencies and adds it to the basic block execution time.

For memory access instructions, a private function memref() is invoked which performs address value analysis and returns the memory address accessed by the instruction. The read or write latency of the memory bank containing the address is added accordingly. Currently, the address value analysis supports only Immediate addressing and PC-relative or
FP-relative indirect addressing. This is because the runtime values of registers are difficult to predict statically. For PC-relative addressing the offset is added to address of the current instruction; and for FP-relative addressing the offset is added to address of the first instruction of the current CFG. Empirically, PC-relative and FP-relative addressing is extensively used by compilers.

For other instruction types, the functional unit to which the instruction is dispatched is retrieved and its latency is added.

The tool ptarmta (source code is given in Appendix IV) combines the PTARMBBTime with otawa::WCETComputation to get WCET of programs on the PTARM architecture.

**Usage:**

$ ptarmta binary_file entry_function [--verbose]

- **binary_file** – path of the binary file under analysis
- **entry_function** – name of the task entry function, default value is “main”

**Dependency:** The flow-fact file should be at the same directory as the binary file. It should have the same name as the binary file with .ff extension. The mkff tool may be used to generate a template flow-fact file. The loop bound values should be manually computed and written to the flow-fact file.

**Usage:**

$ mkff binary_file

For the test program (Appendix V), the generated flow fact template file is as follows:

```plaintext
checksum "otawa1" 0xae4b915d;

// Function icrc ../src/test.c:29
loop "icrc" + 0xe4 ?; // 000083c4
loop "icrc" + 0x264 ?; // 00008544

// Function icrc1 ../src/test.c:15
loop "icrc1" + 0xb8 ?; // 000082c4
```

The ? symbols should be replaced by appropriate loop bounds before running the timing analysis tool.
Sample Run:

$ ptarmta test icrc --verbose

---
Starting PTARMBBTime (1.0.0)
process CFG icrc
process BB 1 (000082e0)

BB 1

[000082e0] mov ip, sp, lsl #0 -> ALU
[000082e4] stmb db spl!, {, r4, fp, ip, lr, pc} ???????? -> DRAM 5
[000082e8] sub fp, ip, #4 -> ALU
[000082ec] sub sp, sp, #24 -> ALU
[000082f0] str r1, [fp, -#32] 000082c0 -> DSPM
[000082f4] str r3, [fp, -#40] 000082b8 -> DSPM
[00008300] ldr r3, [fp, -#28] 000082c4 -> DSPM
[00008304] strh r0, [fp, -#18] 000082ce -> DSPM
[00008308] ldr r3, [pc,#712] 000085d0 -> DRAM
[0000830c] ldrh r3, [r3,#0] ???????? -> DRAM
[00008310] cmp r3, #0 -> ALU
[00008314] bne 83d0 -> ALU

---

Ending PTARMBBTime

PROVIDED: otawa::BB_TIME by PTARMBBTime

---

<table>
<thead>
<tr>
<th>CFG</th>
<th>BB</th>
<th>CYCLES</th>
<th>COUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>icrc</td>
<td>ENTRY</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>BB 1 (000082e0)</td>
<td>29</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>BB 2 (00008318)</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>BB 3 (000083d0)</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>BB 4 (000083c4)</td>
<td>3</td>
<td>257</td>
</tr>
<tr>
<td></td>
<td>BB 5 (000083dc)</td>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>BB 6 (00008410)</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>BB 7 (00008330)</td>
<td>8</td>
<td>256</td>
</tr>
<tr>
<td></td>
<td>BB 8 (00008478)</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>BB 9 (0000841c)</td>
<td>35</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>BB 10 (00008350)</td>
<td>49</td>
<td>256</td>
</tr>
<tr>
<td></td>
<td>BB 11 (00008544)</td>
<td>4</td>
<td>43</td>
</tr>
<tr>
<td></td>
<td>BB 12 (00008554)</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>BB 13 (00008484)</td>
<td>3</td>
<td>42</td>
</tr>
<tr>
<td></td>
<td>BB 14 (00008560)</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>BB 15 (0000856c)</td>
<td>35</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>BB 16 (00008490)</td>
<td>27</td>
<td>42</td>
</tr>
<tr>
<td></td>
<td>BB 17 (000084cc)</td>
<td>17</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>BB 18 (000085c8)</td>
<td>19</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>BB 19 (000084f8)</td>
<td>25</td>
<td>42</td>
</tr>
<tr>
<td>icrc1</td>
<td>EXIT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>BB 1 (0000820c)</td>
<td>30</td>
<td>256</td>
</tr>
<tr>
<td></td>
<td>BB 2 (000082c4)</td>
<td>3</td>
<td>2304</td>
</tr>
<tr>
<td></td>
<td>BB 3 (000082d0)</td>
<td>15</td>
<td>256</td>
</tr>
<tr>
<td></td>
<td>BB 4 (00008268)</td>
<td>5</td>
<td>2048</td>
</tr>
<tr>
<td></td>
<td>BB 5 (0000827c)</td>
<td>12</td>
<td>2048</td>
</tr>
<tr>
<td></td>
<td>BB 6 (000082ac)</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>BB 7 (000082b8)</td>
<td>3</td>
<td>2048</td>
</tr>
<tr>
<td>icrc1</td>
<td>EXIT</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

WCET = 77377
The output prints the basic block execution times and execution counts for the involved CFGs and finally the WCET. The debug trace shows the analysis for each basic block. Each instruction is printed followed by the dispatched functional unit. For memory instructions, the accessed memory address is printed if determined and is followed by the memory bank – DSPM (Data Scratchpad) / DRAM. For multiple load /store instructions, the number of registers is also printed. Notice that undetermined addresses are assumed to access the DRAM leading to an overestimation.

5.2 Using OTAWA on Code Segments

The OTAWA tool computes the WCET for a specified task entry function. To obtain the WCET of an arbitrary piece of code, one can create a dummy function with the required code in its body and pass it as the task entry function to the OTAWA tool. Although this appears simple, it is erroneous due to –

- Additional assembly code generated to handle function call and return.
- Additional C code to re-declare out of scope variables in dummy function.
- The need for recompilation and problems arising from that.

5.2.1 Initial Approach

The other way is to modify the CFG program representation to contain the required code only and then run the analysis on this sub-CFG to obtain WCET of the sub-program. Although OTAWA internal classes make it difficult to modify a CFG program representation, it provides classes to rebuild new CFGs. The otawa::SubCFGBuilder class builds a new CFG starting at the specified CFG_START address and ending at CFG_STOP address(es). Since the OTAWA tool works with the assembly code, it requires to specify in terms of addresses. The addresses are available only after compilation and are thus not visible at the source level. We write the gelly tool (Appendix VII) using the GEL library, which provides a framework for loading, decoding and reading ELF (Executable and Linkable Format) files, to translate line numbers at the source level into addresses at the assembly level.

The gelly tool takes the annotated source file and the binary file as input and prints the addresses corresponding to the lines which are annotated with a ‘marker’ string.

Sample output of the address translation tool for the test code with marker "@SEG" (Appendix V):

<table>
<thead>
<tr>
<th>Line</th>
<th>LowAddress HighAddress</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>00008268 0000827c</td>
</tr>
<tr>
<td>20</td>
<td>000082ac 000082b8</td>
</tr>
<tr>
<td>41</td>
<td>00008484 00008490</td>
</tr>
<tr>
<td>45</td>
<td>000084f8 00008538</td>
</tr>
<tr>
<td>57</td>
<td>00008604 00008618</td>
</tr>
<tr>
<td>59</td>
<td>00008634 00008658</td>
</tr>
<tr>
<td>61</td>
<td>00008674 0000869c</td>
</tr>
</tbody>
</table>

We then build a sub-CFG for the code between the required lines by passing the translated addresses to the SubCFGBuilder and then resume with the PTARMBBTime and WCETComputation processors to obtain the WCET of the required piece of code.
However, we abandon this approach due to several issues arising from building sub-CFGs.

• Sub-CFG may be broken if STOP address is not reachable from the START address.
• START and STOP addresses may not be contained in the same loop or may be in different loops.
• Loop bounds within sub-CFGs are different from actual bounds.

5.2.2 Alternative Approach (under development):
Due to several issues with sub-CFG building, the TRACES team at University of Toulouse, France is exploring an alternate approach to obtain WCETs of arbitrary pieces of code. It is done in two steps-
1. WCET computation of the function containing the code is done as usual.
2. Using the basic block Execution Time and Execution Counts, the time spent in the target sub-CFG is computed.

As per the OTAWA authors, although this approach gives interesting results in some cases, it might be difficult to well tune and is currently under trial.

5.2.3 Current Solution
The current solution does not involve modifying / rebuilding CFGs. It performs the analysis on the original program representation but restricts its domain to the required code segment only. As it occurs, this requires an additional phase to determine the domain of the analysis and specialised analyses only during two phases – Basic Block Execution Time Computation and ILP System Generation. The analysis for a single code segment proceeds in the following phases-

1. Address Translation
The address translation tool- gelly (Appendix VIII) reads the source file line-by-line, identifies annotated lines and finds the addresses corresponding to the source line from the ELF Line Map in the binary file. The current version of the gelly tool returns an UpAddress-address of the instruction preceding the annotated line, and a DownAddress-address of the first instruction on or after the annotated line. These addresses become the boundaries of the segments. It stores this information which are used to derive the segment boundary addresses as described later in section 5.2.4.

2. Domain Analysis / Flood Analysis
This phase is implemented by the ptarm::AddressDomainBuilder processor (Appendix IX) and provides an additional feature ADDRESS_DOMAIN_FEATURE.

Domain analysis begins by locating the START and STOP addresses of the required code segment and marking the basic blocks containing them as StartBB and StopBB respectively. In the forward flood starting with the StartBB, we traverse down all out-edges and mark their target basic blocks. In the backward flood starting with the StopBB, we traverse up all in-edges and mark their source basic blocks. All basic blocks which are marked by both the forward and backward floods, belong to a path leading from the StartBB to the StopBB, are then marked as part of the domain and the range of their addresses are added to the Address Domain. The Address Domain is a list of contiguous address ranges that form the domain. For the StartBB and StopBB, the address range added to the Address Domain starts at START address and ends at STOP address respectively. This captures the fact that a segment can start and end anywhere within the same or different basic blocks. An edge
belongs to the domain only if both the last address of the source block and the first address of the target block belong to the Address Domain. The addresses for called functions are then recursively added to the Address Domain.

3. Domain Limited BB Time Analysis

The Basic Block Execution Time Computation remains the same as described in section 5.1. The PTARMBBTime processor (Appendix III) is slightly modified to add execution latencies of instructions belonging to the Address Domain only. So, only the instructions in the Address Domain contribute to basic block execution times. This allows timing analysis on arbitrary code segments without splitting the basic blocks.

The PTARMBBTime processor must be configured to perform domain limited analysis by setting the SUB_CFG property. Otherwise, the PTARMBBTime processor proceeds normally.

4. ILP System Generation

The ILP system consists of a set of Variables, a set of Constraints and an Objective. OTAWA assigns variables to each basic block and edge. These variables denote the execution counts of the respective elements.

The ILP Constraints used by the Implicit Path Enumeration Technique (IPET) are essentially the following –

i. **Structural Constraints**:
   For each basic block, execution count is equal to both the sum of execution counts for in-edges and that for out-edges. This expresses preservation of flow.

   Let variable associated with element \( a \) (basic block / edge) be denoted by \( x_a \).

   Then for all basic block \( n \),

   \[
   x_n = \sum_{i \in \text{in}(n)} x_i = \sum_{j \in \text{out}(n)} x_j
   \]

   where, \( \text{in}(n) \) is the set of in-edges to \( n \) and \( \text{out}(n) \) is the set of out-edges from \( n \).

ii. **Loop Constraints**:
   For each loop header block, the execution count of the back edge is upper bounded by the sum of the execution counts of the remaining in-edges multiplied by the loop bound. The general form of loop entry constraint is given by:

   \[
   \lambda \times \sum_{i \in \text{in}(n)-\text{back}(n)} x_i \geq \sum_{b \in \text{back}(n)} x_b
   \]

   where, \( n \) is a loop header block, \( \text{back}(n) \) is the set of back edges and \( \lambda \) is the loop bound.

iii. **Calling Constraints**:
   Since a basic block can call only one function in one execution, its execution count must be equal to the sum of function calls. The general form of function call constraint is given by:

   \[
   x_n = \sum_{f \in \text{called}(n)} x_{f^o}
   \]

   where, \( \text{called}(n) \) is the set of functions called by block \( n \) and \( f^0 \) is the entry block of function \( f \).

iv. **Program Entry Constraint**:
   To represent that the program is run once.

   \[
   x_{\text{main}^0} = 1
   \]

The Objective is to maximize \( \sum t_n \times x_n \), where \( t_n \) is the execution time of block \( n \).
The ILP system for the domain is generated as follows:

i. For each basic block \( n \) outside the domain, an **Out Of Domain Constraint** \( (x_n = 0) \) is added.

ii. The Structural and Calling Constraints are built only for the basic blocks inside the domain.

iii. Loop Constraints are added for only those loop header blocks for which both the back edge and the loop entry edge are in the domain.

The back edges are those which lead from the loop body to the header and the loop entry edge leads from the header to the body.

The Program Entry Constraint becomes \( x_{\text{StartBB}} = 1 \) instead of \( x_{\text{main}} = 1 \).

iv. Finally, the Objective function is \( \sum_{n \in \text{Domain}} t_n \times x_n \).

This phase is implemented using 3 processors- SegConstraint, SegLoopBreaker and SegObjective (Appendix X). The SegLoopBreaker processor only identifies broken loops and unmarks their header block. The actual Loop Constraints are then generated by the default FlowFactConstraintBuilder.

5. **WCET Analysis**

The function containing the segment is analysed as usual. The ILP solver is invoked for the generated ILP system. A solution to the ILP system is an assignment to each execution count variable. The value of the Objective function gives the composed WCET of the code segment.

The Domain Limited BB Time Analysis, as described in phase 3, ensures that only the instructions belonging to the required code segment contribute to the WCET. And the ILP system generated over the domain during phase 4 ensures that the analysis considers only the control flow within the domain.

The segta tool (Appendix XI) combines the processors for above phases to get WCET of a code segment. The segment is specified by annotating markers “//@START” and “//@STOP” in the source.

**Usage:** `$ segta binary_file source_file [--verbose]`

- `binary_file` – path of the binary file under analysis
- `source_file` – path of the annotated source file

**Dependency:** The loop bounds must be specified in the flow-fact file.
Sample Run: For Consumer program (Appendix VI)

```bash
$ segta consumer ../src/consumer.c
---
Line   UpAddress   DownAddress
13     000082c0   0000827c
14     000082c0   0000827c
---Address Domain---
Entry = [main]
Start-0000827c | BB7
Stop-000082c0  | BB5
Processor: ptarm [arm] © UC Berkeley
---

<table>
<thead>
<tr>
<th>BB</th>
<th>CYCLES</th>
<th>COUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BB 5  (000082b4)</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>BB 7  (0000827c)</td>
<td>18</td>
<td>1</td>
</tr>
</tbody>
</table>

WCET = 21
```

5.2.4 Matching Segment Boundaries

The segta tool extends OTAWA to perform timing analysis for a specified code segment. The next requirement is to automatically derive the segment boundaries by looking at the deadline instructions. In a program with several instances of the deadline instruction, matching deadline instructions to find segments is a non-trivial task. A deadline instruction may lead to several candidates on a path but only the nearest one is to be matched with. A deadline instruction can simultaneously match with different deadline instructions on disjoint program paths. A deadline instruction inside a loop may match with itself. A deadline instruction may not find a match at all.

The first step is to locate deadline instructions in the source and use the address translation tool to get their addresses. Next we perform reachability test on every possible ordered pair of deadline instructions. The pair forms a segment if it is possible to reach from the first deadline to the second one without going via any other deadline. The SegPicker processor (Appendix XII) performs the reachability test and finds all segments.

The deadta tool (Appendix XIII) combines the SegPicker processor with the previous processors. It identifies all segments in a program and performs timing analysis on each segment as described in section 5.2.3. Finally it creates a copy of the source file and writes the segment WCET values in the corresponding deadline parameters. It is important to note that a deadline instruction can match with multiple deadlines on disjoint program paths to form multiple segments. To ensure that deadlines are never missed, we write the highest execution time among these segments as the deadline parameter.

Usage:  

```
$ deadta binary_file source_file [--verbose]
```

- `binary_file` – path of the binary file under analysis
- `source_file` – path of the annotated source file

Dependency: The loop bounds must be specified in the flow-fact file.
Sample Run: For Consumer program (Appendix VI)

```
$ deadta consumer ../src/consumer.c
---
Line  UpAddress  DownAddress
  6    0000821c  0000821c
 14    000082c0  0000827c
---
Seg 14 [0000827c - 000082c0]
---
---Address Domain---
Entry = [main]
Start-0000827c  | BB7
Stop-000082c0  | BB5
---
Seg 6 [0000821c - 000082c0]
---
---Address Domain---
Entry = [main]
Start-0000821c  | BB1
Stop-000082c0  | BB5
---
Seg 14 [0000827c - 000082c0]  WCET= 21
Seg 6  [0000821c - 000082c0]  WCET= 136
```

Additionally, it creates a copy of the source file named “consumer.c.dead” and writes the deadline parameters.

```c
int main()  //@Consumer
{
    //@Consumer
    volatile unsigned int * buf = (unsigned int*)(0x00008000);
    unsigned int i = 0;
    int arr[8];
    for (i = 0; i<8; i++)
        arr[i] = 0;
    for (i = 0; i ; i++)
    {
        //@Consumer
        //DEAD(21);
        register int tmp = *buf;
        arr[i%8] = tmp;
    }
    return 0;
}
```
6. Future Work

6.1 Segment Annotations

Some synchronization requirements can be derived by studying the code itself. For example, segment ordering between concurrent threads with shared data access can be solved through a Read After Write policy. But it is a difficult task to derive such information and it might not always be possible. Synchronization might depend on functionality or interactions with the physical environment. However, the programmer is aware of such synchronization requirements and can easily annotate the information by writing structured comments.

We now describe the proposed annotations to describe the synchronization requirements. A thread must be named by commenting at the main() function beginning. For example:

```c
int main()    //@Producer
```

A segment is demarcated by two consecutive sync deadlines. A segment can be named by commenting at the start sync deadline. Optionally it can have a list of synchronization requirements with respect to other segments. A segment must be named in order to be aligned.

5 types of synchronizations are possible.

- **sa** - starts after
- **sw** - starts with
- **eb** - ends before
- **ew** - ends with
- **ls** - lock step
Example:

Figure 5: Example usage of segment annotations to specify custom synchronization requirements and one according alignment.

Using the segment annotations the synchronization requirements for the Producer-Consumer-Observer program can be specified as follows:

![Diagram of T1, T2, T3 with annotations]

Figure 6: Annotating synchronization requirements for the PCO-program.
6.2 Segment Aligning

Once the synchronization requirements are annotated to the source files, we load them and build a Segment Table with the information about segments for each thread.

The structure of the Segment Table is shown in figure 7.

The Execution Time is a constant and is derived by IPET analysis on the code segment.
The Sync Time is adjustable and is initially zero.
Clock is the cumulative Execution Time + Sync Time till that segment.
Each segment has a marker, initially unmarked, denoting whether it is aligned or not.

Figure 7: Segment Table structure

Segment Aligning Algorithm

1. Create segment table for each thread.
2. Compute Execution Time of each segment.
3. March alternatingly through the segment table of each thread and align the segments.
   1. Segment $S^a_i$ can be aligned w.r.t $S^b_j$ only if for all $y < j$, $S^b_y$ are already aligned.
   2. Check clocks of $S^a_i$ and $S^b_j$. If already aligned then do nothing.
   3. Else, if $sa / sw$ then adjust $ST(S^a_{i-1})$
      if $eb / ew$ then adjust $ST(S^b_j)$
      if $ls$ then adjust $ST(S^a_i)$ and $ST(S^b_j)$
   4. Update clocks and mark $S^a_i$.
   5. If new adjustments invalidate previous alignment, report error.

7. Conclusion

We have adapted the OTAWA tool for timing analysis on the PTARM architecture. Furthermore, we enable the OTAWA tool to perform timing analysis on code segments.

We provide the following 3 tools-
1. **ptarmta** – Performs WCET analysis for a program on the PTARM architecture.
2. **segta** – Performs WCET analysis for a specified code segment on the PTARM.
3. **deadta** – Picks segments from a code with deadline instruction as the segment boundaries and performs WCET analysis on each code segment.
Further improvement might include following:
1. Proper modelling for extensive address value analysis. Currently we can determine address values only for Immediate, PC-relative and FP-relative addressing.
2. Integrating with the ORange tool (provided by OTAWA) to auto-generate flow-fact constraints.

References


OTAWA API Interface- http://www.otawa.fr/doku/autodoc
I. PTARM Pipeline description

```xml
<?xml version="1.0" encoding="UTF-8"?>
<processor class="otawa::hard::Processor">
  <arch>arm</arch>
  <model>ptarm</model>
  <builder>UC Berkeley</builder>
  <frequency>25000000</frequency>
  <arch>
  </arch>
  <model>ptarm</model>
  <builder>UC Berkeley</builder>
  <frequency>25000000</frequency>
  <arch>
  </arch>
  <model>ptarm</model>
  <builder>UC Berkeley</builder>
</processor>
```

---

1. PTARM Pipeline description

This is the equivalent processor description for a single thread on PTARM. -->

```xml
<processor class="otawa::hard::Processor">
  <arch>arm</arch>
  <model>ptarm</model>
  <builder>UC Berkeley</builder>
  <frequency>25000000</frequency>
  <arch>
  </arch>
  <model>ptarm</model>
  <builder>UC Berkeley</builder>
  <frequency>25000000</frequency>
  <arch>
  </arch>
  <model>ptarm</model>
  <builder>UC Berkeley</builder>
</processor>
```

---

2. PTARM Pipeline description

This is the equivalent processor description for a single thread on PTARM. -->

```xml
<processor class="otawa::hard::Processor">
  <arch>arm</arch>
  <model>ptarm</model>
  <builder>UC Berkeley</builder>
  <frequency>25000000</frequency>
  <arch>
  </arch>
  <model>ptarm</model>
  <builder>UC Berkeley</builder>
  <frequency>25000000</frequency>
  <arch>
  </arch>
  <model>ptarm</model>
  <builder>UC Berkeley</builder>
</processor>
```

---

3. PTARM Pipeline description

This is the equivalent processor description for a single thread on PTARM. -->

```xml
<processor class="otawa::hard::Processor">
  <arch>arm</arch>
  <model>ptarm</model>
  <builder>UC Berkeley</builder>
  <frequency>25000000</frequency>
  <arch>
  </arch>
  <model>ptarm</model>
  <builder>UC Berkeley</builder>
  <frequency>25000000</frequency>
  <arch>
  </arch>
  <model>ptarm</model>
  <builder>UC Berkeley</builder>
</processor>
```

---

4. PTARM Pipeline description

This is the equivalent processor description for a single thread on PTARM. -->

```xml
<processor class="otawa::hard::Processor">
  <arch>arm</arch>
  <model>ptarm</model>
  <builder>UC Berkeley</builder>
  <frequency>25000000</frequency>
  <arch>
  </arch>
  <model>ptarm</model>
  <builder>UC Berkeley</builder>
  <frequency>25000000</frequency>
  <arch>
  </arch>
  <model>ptarm</model>
  <builder>UC Berkeley</builder>
</processor>
```

---

5. PTARM Pipeline description

This is the equivalent processor description for a single thread on PTARM. -->

```xml
<processor class="otawa::hard::Processor">
  <arch>arm</arch>
  <model>ptarm</model>
  <builder>UC Berkeley</builder>
  <frequency>25000000</frequency>
  <arch>
  </arch>
  <model>ptarm</model>
  <builder>UC Berkeley</builder>
  <frequency>25000000</frequency>
  <arch>
  </arch>
  <model>ptarm</model>
  <builder>UC Berkeley</builder>
</processor>
```

---

6. PTARM Pipeline description

This is the equivalent processor description for a single thread on PTARM. -->

```xml
<processor class="otawa::hard::Processor">
  <arch>arm</arch>
  <model>ptarm</model>
  <builder>UC Berkeley</builder>
  <frequency>25000000</frequency>
  <arch>
  </arch>
  <model>ptarm</model>
  <builder>UC Berkeley</builder>
  <frequency>25000000</frequency>
  <arch>
  </arch>
  <model>ptarm</model>
  <builder>UC Berkeley</builder>
</processor>
```

---

7. PTARM Pipeline description

This is the equivalent processor description for a single thread on PTARM. -->

```xml
<processor class="otawa::hard::Processor">
  <arch>arm</arch>
  <model>ptarm</model>
  <builder>UC Berkeley</builder>
  <frequency>25000000</frequency>
  <arch>
  </arch>
  <model>ptarm</model>
  <builder>UC Berkeley</builder>
  <frequency>25000000</frequency>
  <arch>
  </arch>
  <model>ptarm</model>
  <builder>UC Berkeley</builder>
</processor>
```

---

8. PTARM Pipeline description

This is the equivalent processor description for a single thread on PTARM. -->

```xml
<processor class="otawa::hard::Processor">
  <arch>arm</arch>
  <model>ptarm</model>
  <builder>UC Berkeley</builder>
  <frequency>25000000</frequency>
  <arch>
  </arch>
  <model>ptarm</model>
  <builder>UC Berkeley</builder>
  <frequency>25000000</frequency>
  <arch>
  </arch>
  <model>ptarm</model>
  <builder>UC Berkeley</builder>
</processor>
```
II. PTARM Memory Layout description

<?xml version="1.0" encoding="UTF-8"?>
<!-- This is the Memory Layout description for PTARM. -->
<memory>
  <banks>
    <bank>
      <!-- Boot Code -->
      <name>PROM</name>
      <address><offset>0x00000000</offset></address>
      <size>0x0000FFFF</size>
      <type>ROM</type>
      <latency>1</latency>
      <write_latency>1</write_latency>
      <writable>false</writable>
      <cachable>false</cachable>
    </bank>
    <bank>
      <!-- Instruction Scratchpad [256MB] -->
      <name>ISPM</name>
      <address><offset>0x40000000</offset></address>
      <size>0x10000000</size>
      <type>SPM</type>
      <latency>1</latency>
      <write_latency>1</write_latency>
      <writable>false</writable>
      <cachable>false</cachable>
    </bank>
    <bank>
      <!-- Data Scratchpad [256MB] -->
      <name>DSPM</name>
      <address><offset>0x50000000</offset></address>
      <size>0x10000000</size>
      <type>SPM</type>
      <latency>1</latency>
      <write_latency>1</write_latency>
      <writable>true</writable>
      <cachable>false</cachable>
    </bank>
    <bank>
      <!-- DRAM [512MB] -->
      <name>DRAM</name>
      <address><offset>0x80000000</offset></address>
      <size>0x20000000</size>
      <type>DRAM</type>
      <latency>4</latency>
      <write_latency>2</write_latency>
      <writable>true</writable>
      <cachable>false</cachable>
    </bank>
    <bank>
      <!-- Memory Mapped I/O -->
      <name>I/O</name>
      <address><offset>0xF0000000</offset></address>
      <size>0x0FFFFFFF</size>
      <type>IO</type>
      <latency>1</latency>
      <write_latency>1</write_latency>
      <writable>true</writable>
      <cachable>false</cachable>
    </bank>
  </banks>
</memory>
# PTARMBBTime.h

```c
/*
 * PTARMBBTime.h
 * Author: subarno
 */

#ifndef PTARM_BB_TIME_H_
#define PTARM_BB_TIME_H_
#include <stdlib.h>
#include <elm/string/String.h>
#include <otawa/otawa.h>
#include <otawa/proc/BBProcessor.h>
#include <otawa/proc/Feature.h>
#include <otawa/hard/Processor.h>
#include <otawa/hard/Memory.h>
#include <otawa/prop/Identifier.h>
#include <otawa/prop/PropList.h>
#include "AddressDomain.h"
using namespace otawa;
using namespace ipet;

namespace ptarm {

/*
 * Set this property to true for Domain Limited BB Time Analysis.
 */
Identifier<bool> SUB_CFG("SUB_CFG");

/*
 * Class: PTARMBBTime
 * This Code Processor computes Basic Block Execution Times on PTARM.
 */
class PTARMBBTime: public otawa::BBProcessor
{
    otawa::hard::Processor *proc;
    otawa::hard::Memory *mem;
    ptarm::AddressDomainBuilder *ActiveDomain;
    bool sub_cfg_proc;

public:
    otawa::BasicBlock *startbb, *stopbb;
    /*
     * Register Code Processor and declare required & provided Feature.
     */
    PTARMBBTime(void): BBProcessor("PTARMBBTime", Version(1, 0, 0))
    {
        require(otawa::COLLECTED_CFG_FEATURE);
        provide(otawa::ipet::BB_TIME_FEATURE);
    }

    /*
     * Load the pipeline and memory descriptions.
     * @param props PropertyList with PROCESSOR_PATH & MEMORY_PATH
     */
    void configure(const PropList& props)
    {
        BBProcessor::configure(props);
        sub_cfg_proc=ptarm::SUB_CFG(props);
        proc = otawa::hard::Processor::load(otawa::PROCESSOR_PATH(props));
        mem = otawa::hard::Memory::load(otawa::MEMORY_PATH(props));
        cout<<"Processor: "<<proc->getModel()<<" ["<<proc->getArch()<<"] © "<<
        proc->getBuilder()<<io::endl;
    }

    /*
     * If Domain Limiter BB Time Analysis, get the Address Domain.
     */
    void setup(WorkSpace *ws)
    {
        if(sub_cfg_proc)
            ActiveDomain=ptarm::ACTIVE_DOMAIN(ws);
    }
}
#endif
```

[21]
/* * Delete the pipeline and memory descriptions. */
void cleanup(WorkSpace *ws)
{
    delete proc;
    delete mem;
}

protected:
/* * Address value analysis
 * This method attempts to determine the memory address accessed by an instruction.
 * @param inst Memory access instruction
 * @param cfg CFG to which inst belongs
 * @return Address accessed by inst
 */

otawa::Address memref(otawa::Inst *inst, CFG *cfg)
{
    otawa::Address adr = 0;
    elm::StringBuffer *buf = new elm::StringBuffer;
    inst->dump(*buf);
    elm::String str = buf->toString();

    if(str.indexOf(']')>=0) //Register Indirect Addressing
    {
        str = str.substring(str.indexOf(']')+1,str.indexOf(']')-str.indexOf('[')-1);
        if(str.startsWith("pc")) //PC-relative
            adr = inst->address(); //base address = address of current instruction
        else if(str.startsWith("fp")) //FP-relative
            adr = cfg->firstInst()->address(); //base address = address of first instruction of current CFG
    }

    else if(str.indexOf("ls",1)>=0);
    else if(str.indexOf("#")>=0) //Add offset
    {
        if(str.charAt(str.indexOf('#')-1)=='-')
            adr -= atoi(str.substring(str.indexOf('#')+1).toString());
        else
            adr += atoi(str.substring(str.indexOf('#')+1).toString());
    }

    else if(str.indexOf('#')>=0) //Immediate addressing
        adr = atoi(str.substring(str.indexOf('#')+1).toString());

    else if(this->isVerbose())
    {
        if(adr.equals(0))
            cout<<"\t?????????";
        else
            cout<<'\t''<<adr;
    }

    delete buf;

    if(adr.equals(0)) //If address cannot be determined
    { //return starting address of DRAM (conservative overestimation)
        adr = mem->banks().get(mem->banks().count()-2)->address();
        return adr;
    }

    else
    {
        /* Basic Block Execution Time Computation
 * Sets otawa::ipet::TIME Property of the Basic Block to computed Execution Time.
 * @param bb Basic Block
 * @param cfg CFG to which bb belongs
 * @param fw current Workspace
 */
    void processBB(WorkSpace *fw, CFG *cfg, BasicBlock *bb)
    {
        otawa::ipet::TIME(bb)=0;
        if(this->isVerbose())
cout<<"BB "<<bb->number()<<io::endl;
if(bb->isEnd())
    return;
for(otawa::BasicBlock::InstIterator inst(bb); inst; inst++)
{  //For each Instruction
    if(sub_cfg_proc && !(ActiveDomain->belongsTo(inst)))
        //Ignore if Out of Domain
        continue;
    if(this->isVerbose())
    {
        cout<<"['<<inst->address()<<"]   ";
        inst->dump(cout);
    }
    for(int s=0; s<proc->getStages().count(); s++)
    {
        otawa::hard::Stage *stg=proc->getStages().get(s);
        if(stg->getType() == otawa::hard::Stage::EXEC)
        {
            if(inst->oneOf(otawa::Inst::IS_MEM))
            {//If Memory Instruction
                if(inst->oneOf(otawa::Inst::IS_MULTI))
                {//If Multiple Load/Store
                    elm::StringBuffer *buf = new elm::StringBuffer;
                    inst->dump(*buf);
                    elm::String str = buf->toString();
                    str=str.substring(str.indexOf('"')+1,str.indexOf('"')-1);
                    int n = 0;
                    for(int i=0; i<str.length(); i++)
                    {
                        if(str.charAt(i)=='\n')
                            n++;
                    }
                    const otawa::hard::Bank *bnk = mem->get(memref(inst,cfg));
                    //Get accessed memory bank
                    if(this->isVerbose())
                        cout<<"-> "<<bnk->name()<<"<<n<<io::endl;
                    if(inst->oneOf(otawa::Inst::IS_LOAD))
                    {//If Load, add read latency * no, of registers
                        otawa:ipect::TIME(bb)+= bnk->latency()*n;
                        else if(inst->oneOf(otawa::Inst::IS_STORE))
                        {//If Store, add write latency * no, of registers
                            otawa:ipect::TIME(bb)+= bnk->writeLatency()*n;
                        }
                    }
                    else  //If single Load/Store
                    {
                        const otawa::hard::Bank *bnk = mem->get(memref(inst,cfg));
                        //Get accessed memory bank
                        if(this->isVerbose())
                            cout<<"-> "<<bnk->name()<<"<<n<<io::endl;
                        if(inst->oneOf(otawa::Inst::IS_LOAD))
                        {//If Load, add read latency
                            otawa:ipect::TIME(bb)+= bnk->latency();
                            else if(inst->oneOf(otawa::Inst::IS_STORE))
                            {//If Store, add write latency
                                otawa:ipect::TIME(bb)+= bnk->writeLatency();
                            }
                        }
                    }
                }
            }
        }
    }
    else  //Data processing/branch instructions
    {
        for(int d=0; d<stg->getDispatch().count(); d++)
        {
            otawa::hard::Dispatch *dsp=stg->getDispatch().get(d);
            //Get dispatched functional unit
            if(inst->oneOf(dsp->getType()))
            {
            }
    }
if(this->isVerbose())
    cout<<" " <<dsp->getFU()->getName()<<endl;
    //add latency of the functional unit
    otawa::ipet::TIME(bb)+= dsp->getFU()->getLatency();
    break;
else
    otawa::ipet::TIME(bb)+= stg->getLatency();
/*
 * ptarmta.cpp
 * Author: subarno
 */

#include <elm/io.h>
#include <otawa/otawa.h>
#include <otawa/ipet.h>
#include <otawa/cfg/CFGBuilder.h>
#include "PTARMBBTime.h"

#define proc_path  ".../otawa-core/share/Otawa/scripts/ptarm/pipeline.xml"
#define mem_path  ".../otawa-core/share/Otawa/scripts/ptarm/memory.xml"

using namespace elm;
using namespace otawa;
using namespace ptarm;
using namespace otawa::ipet;

/*
 * PTARM Timing Analysis Tool
 * This tool runs the PTARM timing analysis on the given binary file
 * and prints Basic Block Execution Times, Execution Counts and WCET.
 * @param args Command Line Arguments
 * 1- binary file path
 * 2- task entry function [optional]
 * 3- "--verbose" [optional]
 */

int main(int argc, char **argv)
{
    try
    {
        if(argc<2)
        {
            cerr<<"Error: Usage- ptarmta <elf path> [entry function="main"] [-verbose]."<<io::endl;
            return 1;
        }

    otawa::PropList props;
    otawa::PROCESSOR_PATH(props) = proc_path;  //Set pipeline description path
    otawa::MEMORY_PATH(props) = mem_path;      //Set memory description path
    otawa::Processor::VERBOSE(props) = false;
    otawa::Manager manager;
    otawa::WorkSpace *ws = manager.load(argv[1], props);  //Load binary file

    if(argc>2)
        otawa::TASK_ENTRY(props)=argv[2];  //Set task entry function
    else
        otawa::TASK_ENTRY(props)="main";

    if(argc>3)
        otawa::Processor::VERBOSE(props) = true;
    ptarm::SUB_CFG(props)=false;  //NOT Domain Limited BB Time Analysis
    PTARMBBTime bbt;            //Compute BB execution times for PTARM
    bbt.process(ws, props);
    ipet::WCETComputation comp;  //Compute WCET
    comp.process(ws);
    ipet::WCETCountRecorder bbc; //Record BB execution counts
    bbc.process(ws, props);
    const otawa::CFGCollection *cfgs = INVOLVED_CFGS(ws);
    for(CFGCollection::Iterator cfg(cfgs); cfg; cfg++)
    {
        cfg->print(cout);
        for(CFG::BBIterator bb(cfg); bb; bb++)
        {
            cout<<'t';
            bb->print(cout);
            if(!bb->isEnd())
                cout<<'t'<<ipet::TIME(bb)<<'t'<<ipet::COUNT(bb)<<io::endl;
            else
                cout<<io::endl;
        }
cout << "WCET = " << ipet::WCET(ws) << io::endl;
delete ws;
} catch (elm::Exception& e) {
cerr << "ERROR: " << e.message() << io::endl;
V. test.c

```c
/* test.c
  *
  * Cyclic Redundancy Check Computation
  * @see SNU-RT WCET Benchmark suite
  * @link http://www.mrtc.mdh.se/projects/wcet/wcet_bench/crc/crc.c
*/

typedef unsigned char uchar;

#define LOBYTE(x) ((uchar)((x) & 0xFF))
#define HIBYTE(x) ((uchar)((x) >> 8))

unsigned char lin[256] = "asddfegawHAFEFaeDsFEawFdsFaefaerdjgp";

unsigned char icrc1(unsigned short crc, unsigned char onech) {
    int i;
    unsigned short ans = (crc ^ onech << 8);
    for (i = 0; i < 8; i++) {
        if (ans & 0x8000) //SEG
            ans = (ans << 1) ^ 4129;
        else
            ans <<= 1; //SEG
    }
    return ans;
}

unsigned short icrc(unsigned short crc, unsigned long len, short jinit, int jrev) {
    static unsigned short icrctb[256], init = 0;
    static uchar rchr[256];
    static unsigned short tmp1, tmp2, j, cword = crc;
    static uchar it[16] = { 0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15 };
    if (!init) {
        init = 1;
        for (j = 0; j <= 255; j++) {
            icrctb[j] = icrc1(j << 8, (uchar) 0);
            rchr[j] = (uchar) (it[j & 0xF] << 4 | it[j >> 4]);
        }
    }
    if (jinit >= 0)
        tmp2 = cword;
    else
        tmp2 = rchr[HIBYTE(cword)] | rchr[LOBYTE(cword)] << 8;
    for (j = 1; j <= len; j++) {
        if (jrev < 0) //SEG
            tmp1 = rchr[lin[j]] ^ HIBYTE(cword);
        else
            tmp1 = lin[j] ^ HIBYTE(cword);
        cword = icrctb[tmp1] ^ LOBYTE(cword) << 8; //SEG
    }
    return (tmp2);
}

int main(void) {
    unsigned short il, i2;
    unsigned long n;
    n = 40;
    lin[n + 1] = 0; //SEG
    i1 = icrc(0, n, (short) 0, 1);
    lin[n + 1] = HIBYTE(i1); //SEG
    lin[n + 2] = LOBYTE(i1);
    i2 = icrc(i1, n + 2, (short) 0, 1); //SEG
    return 0;
}
```
VI. Producer, Consumer and Observer programs

```c
/*
* producer.c
*/
int main() //@Producer
{
  //DEAD();
  volatile unsigned int * buf = (unsigned int*)(0x00000000);
  unsigned int i = 0;
  for (i = 0; i < 8; i++)
  {
    //DEAD();
    *buf = i;
  }
  return 0;
}

/*
* consumer.c
*/
int main() //@Consumer
{
  //DEAD();
  volatile unsigned int * buf = (unsigned int*)(0x00000000);
  unsigned int i = 0;
  int arr[8];
  for (i = 0; i < 8; i++)
  {
    arr[i] = 0;
  }
  for (i = 0; i < 8; i++)
  {
    //DEAD();
    register int tmp = *buf;
    arr[i%8] = tmp;
  }
  return 0;
}

/*
* observer.c
*/
int main() //@Observer
{
  //DEAD();
  volatile unsigned int * buf = (unsigned int*)(0x00000000);
  volatile unsigned int * fd = (unsigned int*)(0x00000000);
  unsigned int i = 0;
  for (i = 0; i < 8; i++)
  {
    //DEAD();
    *fd = *buf;
  }
  return 0;
}
```
VII. gel.h (old)

```c
/*
 * gel.h
 *
 * Author: subarno
 */
#include <stdio.h>
#include <elm/io.h>
#include <gel/gel.h>
#include <gel/file.h>
#include <gel/error.h>
#include <gel/gel_elf.h>
#include <gel/dwarf_line.h>

using namespace elm;
/*
 * Class: gelly
 *
 * This class uses the GEL Library functions to translate line numbers from the
 * source file to compiled addresses. Desired lines are annotated
 * with a 'marker'.
 */
class gelly
{
/*
 * This structure is used to store line and address information of desired source
 * lines. A single line in the source may compile to a number of instructions.
 * hiadr and loadr is used to describe the range of addresses for the source line.
 */
typedef struct lin_info_struct
{
   int line;
   otawa::address_t hiadr, loadr;
   String file, comment;
} lin_info;
public:
/*
 * List of line information.
 */
elm::genstruct::SLList<lin_info *> lins;
/*
 * This method identifies annotated source lines, extracts their addresses from the
 * Line Map and stores this information in the list.
 * @param src_path path of C source file
 * @param elf_path path of binary file
 * @param marker 'marker' string annotated to identify desired lines
 */
void get_segs(char * src_path, char *elf_path, char *marker)
{
   int line=0;
   char str[100];
   FILE *src = fopen(src_path,"r"); //Open source file in read mode.
   if(src == NULL)
   {
      cout<<"ERROR: Cannot open "<<src_path<<io::endl;
      return;
   }
   gel_file_t *file;
   dwarf_line_map_t *map;
   file = gel_open(elf_path, "", 0); //Open the ELF binary file
   if(file == NULL)
   {
      cout<<"ERROR: "<<gel_strerror()<<io::endl;
      return;
   }
   map = gel_new_line_map(file); //Load Line Map of the ELF
   if(!map)
```
{  
    cerr<<"ERROR: Cannot Load Line Map."<<io::endl;
    gel_close(file);
    return;
}

dwarf_line_iter_t iter;

dwarf_location_t loc;

while(fgets(str,100,src)!=NULL)  //Read source file line-by-line
{
    lin_info *lin;
    line++;
    if(strstr(str,marker)!=NULL)   //If source line is annotated by 'marker'
    {
        lin = new lin_info();   //Store line information and number
        lin->file=src_path;     
        lin->comment=str;      
        lin->line=line;        
        for(loc = dwarf_first_line(&iter, map);loc.file;loc =
            dwarf_next_line(&iter))
        {
            if(loc.line==line)    //Find line in the Line Map
                {
                    lin->loadr = loc.low_addr;  //Get line addresses.
                    lin->hiadr = loc.high_addr;
                }
        }
        lins.addLast(lin);
    }
    gel_delete_line_map(map);  //delete Line Map
    gel_close(file);  //Close source and binary files
    fclose(src);
}

/*
 * Print the list of 'marked' lines and corresponding addresses.
 */

void print_segs()
{
    cout<<"Line\tLowAddress\tHighAddress"<<io::endl;
    for(elm::genstruct::SLList<lin_info*>*::Iterator iter(lins); iter; iter++)
    {
        cout<"line<"\t"<iter->loadr<"\t"<iter->hiadr<<io::endl;
    }
};
/*
 * gel.h
 *      Author: subarno
 */

#ifndef GEL_H_
#define GEL_H_
#include <stdio.h>
#include <elm/io.h>
#include <gel/gel.h>
#include <gel/file.h>
#include <gel/error.h>
#include <gel/gel_elf.h>
#include <gel/dwarf_line.h>
#include "PTARMBBTime.h"

using namespace ptarm;
using namespace elm;

/*
 * This structure is used to store line and address information of desired source
 * lines. A single line in the source may compile to a number of instructions.
 * hiadr and loadr is used to describe the range of addresses for the source line.
 */

typedef struct lin_info_struct
{
    int line;
    long int wcet;
    String file, comment;
    otawa::address_t up, dn;
} lin_info;

/*
 * Class: gelly
 * This class uses the GEL Library functions to translate line numbers from the
 * source file to compiled addresses. Desired lines are annotated with a 'marker'.
 */

class gelly
{
    private:
        /*
         * This method compares the reverse of two strings until a terminal character.
         * Used to compare filenames in absolute and relative paths.
         * @param str1 string2
         * @param str2 string2
         * @param terminate terminal character
         * @return 0 if equal, else 1
         */
        char revcmp(const char *str1, const char *str2, char terminate)
        {
            while("\0"!=*str1)str1++;
            while("\0"!=*str2)str2++;
            while(*str1==*str2&.(*str1!=terminate)&.(*str2!=terminate))
            {
                str1--;
                str2--;
            }
            if(*str1==*str2&.(*str1==terminate))
                return 0;
            else
                return 1;
        };
        otawa::address_t start_adr, stop_adr;
        elm::genstruct::SLList<lin_info *> lins;
        /*
         * This method identifies annotated source lines and extracts addresses of the
         * preceding and following instructions from the Line Map.
         */
void get_lines(char * src_path, char *elf_path, char *start_marker, char *stop_marker) {
    int line=0;
    char str[100];

    FILE *src = fopen(src_path,"r"); //Open source file in read mode.
    if(src == NULL) {
        cout<<"ERROR: Cannot open "<<src_path<<endl;
        return;
    }

    gel_file_t *file;
    dwarf_line_map_t *map;
    file = gel_open(elf_path, "," , 0); //Open the ELF binary file
    if(file == NULL) {
        cout<<"ERROR: "<<gel_strerror()<<endl;
        return;
    }

    map = gel_new_line_map(file); //Load Line Map of the ELF
    if(!map) {
        cerr<<"ERROR: Cannot Load Line Map."<<endl;
        gel_close(file);
        return;
    }

    dwarf_line_iter_t iter;
    dwarf_location_t loc;
    int maxlin=0;
    for(loc = dwarf_first_line(&iter,map);loc.file;loc = dwarf_next_line(&iter)) {
        //Find highest line number in the Line Map
        if((revcmp(loc.file,src_path,'/')==0)&&(loc.line>maxlin))
            maxlin=loc.line;
    }

    while(fgets(str,100,src)!=NULL) //Read source file line-by-line
    {
        lin_info *lin;
        line++;
        if(strstr(str,start_marker)!=NULL||strstr(str,stop_marker)!=NULL) { //If source line is annotated by 'marker'
            lin = new lin_info(); //Store line information and number
            lin->file=src_path;
            lin->comment=str;
            lin->line=line;
            lin->up=0;
            lin->dn=0;

            for(int i=0; !((lin->up && lin->dn) && !(line-i-1<i) &&
                !(line+i>maxlin)); i++)
            {
                if(!lin->up) {
                    //Find previous line in the Line Map.
                    for(loc = dwarf_first_line(&iter,map);loc.file;loc = dwarf_next_line(&iter))
                    {
                        if((loc.line==line-i-1) &&
                            (revcmp(loc.file,src_path,'/')==0))
                        {
                            if(ptarm::cmp(GT,loc.high_addr,lin->up))
                                lin->up=loc.high_addr;
                        }
                    }
                }
            }
        }
    }
}
if(!line->dn)
{
    for(loc = dwarf_first_line(&iter, map); loc.file; loc =
        dwarf_next_line(&iter))
    {  //Find line in the Line Map.
        if((loc.line==line+i)&&(revcmp(loc.file, src_path, '/')==0))
            if(!line->dn||ptarm::cmp(LT, loc.low_addr, line->dn))
                line->dn=loc.low_addr;
    }
}

lins.addLast(line);
if strstr(str,start_marker)!=NULL)
    start_adr=line->dn;
else if strstr(str,stop_marker)!=NULL)
    stop_adr=line->up;
}
gel_delete_line_map(map);  //delete Line Map
gel_close(file);  //Close source and elf files
fclose(src);
}

/* Print the list of 'marked' lines and corresponding addresses.
*/
void print_lines()
{
    cout<<"Line\tUpAddress\tDownAddress"<io::endl;
    for(elm::genstruct::SLLList<lin_info *>::Iterator liter(lins); liter; liter++)
        cout<<liter->line<<'\t'<<liter->up<<'\t'<<liter->dn<<io::endl;
}
#endif
AddressDomainBuilder.h

/* AddressDomain.h */

Author: subarno

#ifndef ADDRESS_DOMAIN_H_
#define ADDRESS_DOMAIN_H_
#include <stdlib.h>
#include <elm/string/String.h>
#include <otawa/proc/BBProcessor.h>
#include <otawa/proc/Feature.h>
#include <otawa/prop/Identifier.h>
#include <otawa/prop/PropList.h>

using namespace otawa;
using namespace ipet;
namespace ptarm {

class AddressDomainBuilder;

/* Identifiers to pass START and STOP addresses. */
Identifier<otawa::address_t> CFG_START("CFG_START");
Identifier<otawa::address_t> CFG_STOP("CFG_STOP");

/* Flags to mark basic blocks during forward and backward flood. */
Identifier<bool> FD_FLAG("FD_FLAG");
Identifier<bool> BD_FLAG("BD_FLAG");

/* Flag to mark basic blocks and edges in the domain. */
Identifier<bool> DOMAIN_FLAG("DOMAIN_FLAG");

/* Identifier to associate Address Domain with the workspace. */
Identifier<AddressDomainBuilder *> ACTIVE_DOMAIN("ACTIVE_DOMAIN");

/* Additional feature provided by AddressDomainBuilder processor. */
Feature <AddressDomainBuilder> ADDRESS_DOMAIN_FEATURE("ptarm::ADDRESS_DOMAIN_FEATURE");

/* Compare two address values. */
@param op Operator. One out of {LT,GT,LE,GE,EQ}
@param a Left address operand.
@param b Right address operand.
@return (a op b)

enum compare {LT,GT,LE,GE,EQ};

bool cmp(compare op, otawa::address_t a, otawa::address_t b) {
    elm::t::size as, bs;
    as=a.page()+a.offset();
    bs=b.page()+b.offset();
    if(op==LT)
        return as<bs;
    else if(op==GT)
        return as>bs;
    else if(op==LE)
        return as<=bs;
    else if(op==GE)
        return as>=bs;
    else if(op==EQ)
        return as==bs;
}

/* Class: AddressDomainBuilder */
This Code Processor builds and stores the Address Domain.
class AddressDomainBuilder: public otawa::Processor
{
private:
    /*
     * Class: AddressRange
     * This class represents a contiguous address range.
     */
    class AddressRange
    {
        public:
            otawa::address_t low, high;
            /
            * Set the addresses.
            * @param lo Lower address bound.
            * @param hi Upper address bound.
            */
            AddressRange(otawa::Address lo, otawa::Address hi)
            {
                low=lo;
                high=hi;
            }
            /
            * Test for belongingness of an address.
            * @param adr Address to test.
            * @return true, if adr belongs to the AddressRange.
            * false, otherwise
            */
            bool belongs(otawa::Address adr)
            {
                if(cmp(GE,adr,low)&cmp(LT,adr,high))
                    return true;
                    return false;
            }
    }
    /*
     * Address Domain is a list of AddressRange objects.
     */
    elm::genstruct::SLList<AddressRange *> adrs;
public:
    /*
     * START and STOP addresses.
     */
    otawa::address_t start_adr, stop_adr;
    /*
     * StartBB and StopBB.
     */
    otawa::BasicBlock *start_bb, *stop_bb;
    /*
     * CFG containing the segment.
     */
    otawa::CFG *entry_cfg;
    /*
     * Register Code Processor and declare required & provided Feature.
     */
    AddressDomainBuilder(void): Processor("AddressDomainBuilder", Version(1, 0, 0))
    {
        require(otawa::COLLECTED_CFG_FEATURE);
        provide(ptarm::ADDRESS_DOMAIN_FEATURE);
    }
    /*
     * Get the START and STOP addresses from the PropertyList.
     * @param props PropertyList with CFG_START & CFG_STOP addresses.
     */
    void configure(const PropList& props)
    {
        Processor::configure(props);
        start_adr=ptarm::CFG_START(props);
stop_adr=ptarm::CFG_STOP(props);

*/
* Set the Address Domain for the workspace.
*/
void cleanup(WorkSpace *ws) {ptarm::ACTIVE_DOMAIN(ws)=this;}

/*
 Set the Address Domain for the workspace.
*/
void cleanup(WorkSpace *ws) {ptarm::ACTIVE_DOMAIN(ws)=this;}

/*
 Test for belongingness of an instruction.
*/
bool belongs(otawa::Inst *ins) {
  if(ins==NULL)
    return true;
  //Check for each AddressRange
  for(elm::genstruct::SLList<AddressRange *>::Iterator iter(adrs);
      iter; iter++)
    if(iter->belongs(ins->address()))
      //If ins->address() belongs to the Address Range, return true
      return true
  return false;  //If belongs to none, return false.
}

private:
/*
 Get the last address of a function.
*/
otawa::address_t lastadr(CFG *cfg) {
  otawa::address_t lst=0;
  for(otawa::BasicBlock::InIterator ins(cfg->exit()); ins; ins++)
    if(ins->source()->address()+ins->source()->size()>lst)
      lst=ins->source()->address()+ins->source()->size();
  return lst;
}

/*
 Add a new AddressRange to the AddressDomain.
*/
void add(AddressRange *adr) {
  if(this->isVerbose())
    cout<<"add "<<adr->low<<" - "<<adr->high<<io::endl;
  for(elm::genstruct::SLList<AddressRange *>::Iterator iter(adrs); iter; iter++)
    if((iter->belongs(adr->low))&&(iter->belongs(adr->high)))
      delete adr;
      return;
  else if(iter->belongs(adr->low))
    { iter->high=adr->high;
      delete adr;
      return;
    }
  else if(iter->belongs(adr->high))
    { iter->low=adr->low;
      delete adr;
    }
```cpp
return;
}

adrs.add(adr);

void addcfg(otawa::CFG *cfg)
{
    if (this->isVerbose())
        cout << "CFG " << cfg->label() << endl;
    add(new AddressRange(cfg->firstInst()->address(), lastadr(cfg)));
    for (otawa::CFG::BBIterator bbs(cfg); bbs; bbs++)
    {
        if (bbs->isCall())
            for (BasicBlock::OutIterator outs(bbs); outs; outs++)
                if (outs->kind() == otawa::Edge::CALL)
                    this->addcfg(outs->calledCFG());
        ptarm::DOMAIN_FLAG(bbs) = true;
    }
}

protected:

    void processWorkSpace(WorkSpace *ws)
    {
        const otawa::CFGCollection *cfgs = INVOLVED_CFGS(ws);
        for (CFGCollection::Iterator cfg(cfgs); cfg; cfg++)
        {
            //Find the CFG containing the START and STOP addresses.
            if (cmp(LE, cfg->firstInst()->address(), start_adr) && cmp(GE, lastadr(cfg), start_adr))
            {
                if (cmp(LE, cfg->firstInst()->address(), stop_adr) && cmp(GE, lastadr(cfg), stop_adr))
                {
                    entry_cfg = cfg;
                    for (CFG::BBIterator bbs(cfg); bbs; bbs++)
                    { //Locate the StartBB and StopBB
                        if (cmp(LT, bbs->address(), start_adr) && cmp(GT, bbs->address() + bbs->size(), start_adr))
                            start_bb = bbs;
                        if (cmp(LT, bbs->address(), stop_adr) && cmp(GE, bbs->address() + bbs->size(), stop_adr))
                            stop_bb = bbs;
                    }
                    break;
                }
            }
        }
    }
}
```

[ 37 ]
if (bbs.first() == stop_bb)
{
    ptarm::FD_FLAG(bbs.first()) = true;
    while (bbs.count() > 0)
        bbs.removeFirst();
    break;
}
for (BasicBlock::OutIterator outs (bbs.first()); outs; outs++)
{
    if (!ptarm::FD_FLAG(outs->target()))
        bbs.addLast(outs->target());
}
ptarm::FD_FLAG(bbs.first()) = true;
bbs.removeFirst();

//Backward Flood Analysis
bbs.addFirst(stop_bb);
while (bbs.count() > 0)
{
    if (bbs.first() == start_bb)
    {
        ptarm::BD_FLAG(bbs.first()) = true;
        break;
    }
    for (BasicBlock::InIterator ins (bbs.first()); ins; ins++)
    {
        if (!ptarm::BD_FLAG(ins->source()))
            bbs.addLast(ins->source());
    }
    ptarm::BD_FLAG(bbs.first()) = true;
bbs.removeFirst();
}
//Build Address Domain
for (CFG::BBIterator bbs (entry_cfg); bbs; bbs++)
{
    if (ptarm::FD_FLAG(bbs) && ptarm::BD_FLAG(bbs))
    {
        //If basic block is marked during both forward and backward flood.
        //Add the Address Range.
        if (this->isVerbose())
            cout << "BB " << bbs->number() << io::endl;
        if (start_bb == bbs || stop_bb == bbs)
        {
            if (start_bb == stop_bb && start_addr <= stop_addr)
                add(new AddressRange(start_addr, stop_addr));
            else
            {
                if (start_bb == bbs)
                    add(new AddressRange(start_addr, bbs->address() + bbs->size()));
                if (stop_bb == bbs)
                    add(new AddressRange(bbs->address(), stop_addr));
            }        
        }        else
            add(new AddressRange(bbs->address(), bbs->address() + bbs->size()));
        //Add called functions to the Address Domain.
        if (bbs->isCall())
        {
            for (BasicBlock::OutIterator outs (bbs); outs; outs++)
            {
                if (outs->kind() == otawa::Edge::CALL)
                    addcfg(outs->calledCFG());
            }
        }
        ptarm::DOMAIN_FLAG(bbs) = true;
    }
//Mark edges belonging to the domain.

for(CFG::BBIterator bbs(entry_cfg); bbs; bbs++)
{
    if(belongs(bbs->firstInst()))
        for(BasicBlock::InIterator edges(bbs);edges;edges++)
            if(belongs(edges->source())->lastInst())
                ptarm::DOMAIN_FLAG(edges)=true;

    if(belongs(bbs->lastInst()))
        for(BasicBlock::OutIterator edges(bbs);edges;edges++)
            if(belongs(edges->target())->firstInst())
                ptarm::DOMAIN_FLAG(edges)=true;

    if(adrs.count()==0)
        cerr<<"Error: Address domain could not be built."<<io::endl;
}
}
#endif
X. SegILPBuilder.h

/*
 * SegILPBuilder.h
 * Author: subarno
 */
#ifndef SEG_ILP_BUILDER_H_
#define SEG_ILP_BUILDER_H_
#include <otawa/ilp.h>
#include <otawa/ipet/IPET.h>
#include <otawa/cfg.h>
#include "AddressDomain.h"
using namespace otawa::ilp;
namespace ptarm {
   /*
   * Class: SegConstraint
   *
   * This Code Processor adds the Structural Constraints for basic blocks in the
   * domain and Out Of Domain Constraints for basic blocks outside the domain.
   */
   class SegConstraint: public BBProcessor {
   private:
      ptarm::AddressDomainBuilder *ActiveDomain;
   public:
      /*
      * Register Code Processor and declare required & provided Feature.
      */
      SegConstraint(void) : BBProcessor("SegConstraint", Version(1, 0, 0))
      {
         require(ASSIGNED_VARS_FEATURE);
         require(ILP_SYSTEM_FEATURE);
         require(ptarm::ADDRESS_DOMAIN_FEATURE);
         provide(CONTROL_CONSTRAINTS_FEATURE);
      }
      /*
      * Get the ILP System and add the Segment Entry Constraint.
      * @param ws Current workspace.
      */
      void setup(WorkSpace *ws)
      {
         ActiveDomain=ptarm::ACTIVE_DOMAIN(ws);
         string label = "entry constraint";
         System *system = SYSTEM(ws);
         Constraint *cons = system->newConstraint(label,Constraint::EQ, 1);
         cons->addLeft(1, VAR(ActiveDomain->start_bb));
         CALLING_CONSTRAINT(ActiveDomain->entry_cfg) = cons;
      }
   }
   /*
   * Add Calling Constraint for a function call.
   * @param system ILP System
   * @param cfg CFG of calling function.
   * @param bb Calling basic block.
   * @param called CFG of called function.
   * @param var Execution Count variable for bb.
   */
   void addEntryConstraint(System *system, CFG *cfg, BasicBlock *bb, CFG *called, otawa::ilp::Var *var)
   {
      Constraint *cons = CALLING_CONSTRAINT(called);
      if(!cons)
      {
         string label = _"calling constraint for "<<called->label();
         cons = system->newConstraint(Constraint::EQ);
         cons->addLeft(1, VAR(called->entry()));
      }
      CALLING_CONSTRAINT(called) = cons;
   }
protected:
/*
 * Add Structural Constraint for a basic block.
 * @param bb Basic block.
 * @param cfg CFG to which bb belongs.
 * @param ws Current Workspace.
 */

void processBB(WorkSpace *ws, CFG *cfg, BasicBlock *bb) {
  Constraint *cons;
  bool used;
  CFG *called = NULL;
  System *system = SYSTEM(ws);
  otawa::ilp::Var *bbv = VAR(bb);
  string label;
  if (!ptarm::DOMAIN_FLAG(bb)) {
    // Out Of Domain Constraints
    label = _"out of domain constraint of BB"<<INDEX(bb)<<"|"<<cfg->label();
    cons = system->newConstraint(label, Constraint::EQ, 0);
    cons->addLeft(1, bbv);
  } else {
    if (bb != ActiveDomain->start_bb) {
      // Structural Input Constraints
      label = _"structural input constraint of BB"<<INDEX(bb)<<"|"<<cfg->label();
      cons = system->newConstraint(label, Constraint::EQ);
      cons->addLeft(1, bbv);
      used = false;
      for (BasicBlock::InIterator edge(bb); edge; edge++)
        if ((edge->kind() != Edge::CALL) && (ptarm::DOMAIN_FLAG(edge)))
          cons->addRight(1, VAR(edge));
      used = true;
    }
    if (!used)
      delete cons;
  }
  if (bb != ActiveDomain->stop_bb) {
    // Structural Output Constraints
    bool many_calls = false;
    cons = system->newConstraint(label, Constraint::EQ);
    cons->addLeft(1, bbv);
    used = false;
    for (BasicBlock::OutIterator edge(bb); edge; edge++)
      if (edge->kind() != Edge::CALL)
        if (ptarm::DOMAIN_FLAG(edge))
          cons->addRight(1, VAR(edge));
        used = true;
    }
    else {
      if (!edge->calledCFG())
        throw ProcessorException(*this, _"unresolved call at "<<bb->address());
      if (called)
many_calls = true;

else
called = edge->calledCFG();
}

if(!used)
delete cons;
//Process function call(s)
if(called)
{
if(!many_calls)
addEntryConstraint(system, cfg, bb, called, bbv);
else
{
label <<"multiple call from BB"<<INDEX(bb)<<"|"<<cfg->label();
Constraint *call_cons = system->newConstraint(Constraint::EQ);
call_cons->addLeft(1, bbv);
for(BasicBlock::OutIterator edge(bb); edge; edge++)
{
if(edge->kind() == Edge::CALL)
{
CFG *called_cfg = edge->calledCFG();
String name;
name = "call_"<<bb->number()"_to_"<<called_cfg->label();
ottawa::ilp::Var *call_var = system->newVar(name);
addEntryConstraint(system, cfg, bb, called_cfg, call_var);
call_cons->addRight(1, call_var);
}
}
}
}
if(!called)
delete cons;
//Process function call(s)
if(called)
{
if(!many_calls)
addEntryConstraint(system, cfg, bb, called, bbv);
else
{
label <<"multiple call from BB"<<INDEX(bb)<<"|"<<cfg->label();
Constraint *call_cons = system->newConstraint(Constraint::EQ);
call_cons->addLeft(1, bbv);
for(BasicBlock::OutIterator edge(bb); edge; edge++)
{
if(edge->kind() == Edge::CALL)
{
CFG *called_cfg = edge->calledCFG();
String name;
name = "call_"<<bb->number()"_to_"<<called_cfg->label();
ottawa::ilp::Var *call_var = system->newVar(name);
addEntryConstraint(system, cfg, bb, called_cfg, call_var);
call_cons->addRight(1, call_var);
}
}
}
}
/*
* Class: SegObjective
* This Code Processor builds the Objective function of the ILP System.
*/
class SegObjective: public BBProcessor {
public:
/*
* Register Code Processor and declare required & provided Feauture.
*/
SegObjective(void): BBProcessor("SegObjective", Version(1, 0, 0))
{
require(ASSIGNED_VARS_FEATURE);
require(BB_TIME_FEATURE);
require(ILP_SYSTEM_FEATURE);
provide(OBJECT_FUNCTION_FEATURE);
}
/*
* Build Objective function.
* For each basic block bb in the domain, add TIME(bb)*VAR(bb).
*/
void processBB(WorkSpace *ws, CFG *cfg, BasicBlock *bb)
{
if(ptarm::DOMAIN_FLAG(bb))
SYSTEM(ws)->addObjectFunction(TIME(bb), VAR(bb));
}
/*
* Class: SegLoopBreaker
* This Code Processor finds broken loops and removes the LOOP_HEADER marker.
*/
class SegLoopBreaker: public BBProcessor {

public:

    /* Register Code Processor and declare required & provided Feature. */
    SegLoopBreaker(void): BBProcessor("SegLoopBreaker", Version(1, 0, 0))
    {
        require(COLLECTED_CFG_FEATURE);
        require(LOOP_HEADERS_FEATURE);
    }

    /* Set LOOP_HEADER to false for broken loops. */
    void processBB(WorkSpace *ws, CFG *cfg, BasicBlock *bb)
    {
        // If bb is a loop header block
        if (otawa::LOOP_HEADER(bb))
            // If bb belongs to the domain
            if (ptarm::DOMAIN_FLAG(bb))
                // Check if every instruction of bb belongs to the domain
                for (BasicBlock::InstIter inst(bb); inst; inst++)
                    if ((ptarm::ACTIVE_DOMAIN(ws) -> belongs(inst))
                        {
                            otawa::LOOP_HEADER(bb) = false;
                            if (this -> isVerbose())
                                cout << "Loop header out of domain. Broken Loop at BB " << bb -> number() << endl;
                            break;
                        }
        // Check if back edge belongs to the domain
        BasicBlock *back = NULL;
        for (BasicBlock::InIterator ins(bb); ins; ins++)
            if (Dominance::dominates(bb, ins -> source()))
                {
                    if (!ptarm::DOMAIN_FLAG(ins))
                        {
                            otawa::LOOP_HEADER(bb) = false;
                            if (this -> isVerbose())
                                cout << "Back edge " << ins -> source() -> number() << "-" << ins -> target() -> number() << " is out of domain. " << "Broken Loop at BB " << bb -> number() << endl;
                        }
                        back = ins -> source();
                        break;
                }
        // Check if loop entry edge belongs to the domain
        if (back)
            for (BasicBlock::OutIterator outs(bb); outs; outs++)
                if (outs -> target() == back || Dominance::dominates(outs -> target(), back))
                    if (!ptarm::DOMAIN_FLAG(outs))
                        {
                            otawa::LOOP_HEADER(bb) = false;
                            if (this -> isVerbose())
                                cout << "Loop entry edge " << outs -> source() -> number() << "-" << outs -> target() -> number() << " is out of domain. " << "Broken Loop at BB " << bb -> number() << endl;
                        }
                else
                    otawa::LOOP_HEADER(bb) = false;
    }

};

/* */
XI. segta.cpp

1 /*
2 * segta.cpp
3 *    Author: subarno
4 */
5 #include <stdlib.h>
6 #include <elm/io.h>
7 #include <otawa/ipet.h>
8 #include <otawa/cfg/CFGBuilder.h>
9 #include "gel.h"
10 #include "PTARMBBTime.h"
11 #include "AddressDomain.h"
12 #include "SegILPBuilder.h"
13 #define proc_path ".../otawa-core/share/Otawa/scripts/ptarm/pipeline.xml"
14 #define mem_path ".../otawa-core/share/Otawa/scripts/ptarm/memory.xml"
15 #define start_marker "//@START"
16 #define stop_marker "//@STOP"
17 using namespace elm;
18 using namespace otawa;
19 using namespace ptarm;
20 /*
21 * Segment Timing Analysis Tool
22 * 
23 * This tool runs the timing analysis for a code segment. The code segment is specified
24 * by annotating "//@START" and "//@STOP" markers in the source file.
25 * @param args Command Line Arguments
26 * 1- binary file path
27 * 2- source file path
28 * 3- "--verbose" [optional]
29 */
30 int main(int argc, char **argv)
31 {
32     try
33     {
34         if(argc<3)
35         {
36             cerr<<"Error: Usage- segta <elf path> <source path> 
[-verbose]."<<io::endl;
37             return 1;
38         }
39         otawa::PropList props;
40         otawa::PROCESSOR_PATH(props) = proc_path; //Set pipeline description path
41         otawa::MEMORY_PATH(props) = mem_path; //Set memory description path
42         otawa::Processor::VERBOSE(props) = false;
43         otawa::Manager manager;
44         otawa::WorkSpace *ws=manager.load(argv[1], props); //Load binary file
45         gelly gel; //Run Address Translation
46         gel.get_lines(argv[2],argv[1],start_marker,stop_marker);
47         gel.print_lines();
48         if(argc>3)
49             otawa::Processor::VERBOSE(props) = true;
50         ptarm::SUB_CFG(props)=true; //Set for Domain Limited Analysis
51         ptarm::CFG_START(props)=gel.start_adr; //Set START address
52         ptarm::CFG_STOP(props)=gel.stop_adr; //Set STOP address
53         AddressDomainBuilder domain; //Build Address Domain
54         domain.process(ws,props);
55         otawa::TASK_ENTRY(props)=domain.entry_cfg->label(); //Set Entry function
56         PTARMBBTime bttime; //Domain Limited BB Time Analysis
57         bttime.process(ws, props);
58         otawa::ipet::EXPLICIT(props) = true;
59         SegConstraint segc; //Build Segment ILP Constraints
60         segc.process(ws,props);
61         SegLoopBreaker loop; //Remove broken loops
62         loop.process(ws,props);
63         SegObjective sego; //Build ILP Objective function
64         sego.process(ws,props);
65         ipet::WCETComputation comp; //Compute WCET
comp.process(ws);
ipet::WCETCountRecorder bbc;  //Record Execution counts
bbc.process(ws, props);
if(otawa::Processor::VERBOSE(props))
{
    otawa::ilp::System *sys = otawa::ipet::SYSTEM(ws);
    sys->dumpSystem(cout);
    sys->dumpSolution(cout);
}
const otawa::CFGCollection *cfgs = INVOLVED_CFGS(ws);
for(CFGCollection::Iterator cfg(cfgs); cfg; cfg++)
{
    cfg->print(cout);
    cout<<\n;
    for(CFG::BBIterator bb(cfg); bb; bb++)
    {
        if(!ipet::TIME(bb))
            continue;
        cout<<'\t';
        bb->print(cout);
        if(!bb->isEnd())
            cout<<" "<<ipet::TIME(bb)<<'\t'<<ipet::COUNT(bb);
        else
            cout<<\n;
    }
    cout<<"WCET = "<<ipet::WCET(ws)<<\n;
    delete ws;
}
catch(elm::Exception& e)
{
cerr<<"ERROR: "<<e.message()<<\n;
XII. SegPicker.h

1 /*
2 * SegPicker.h
3 * Author: subarno
4 */
5 #ifndef SEGPICKER_H_
6 #define SEGPICKER_H_
7 #include <stdlib.h>
8 #include <elm/string/String.h>
9 #include <otawa/ipet/ConstraintLoader.h>
10 #include <otawa/util/Dominance.h>
11 #include "gel.h"
12 using namespace elm;
13 using namespace otawa;
14 namespace ptarm {
15 class SegPicker;
16 /*
17 * Identifier to associate Segment Information with the workspace.
18 */
19 Identifier<SegPicker *> SEG_INFO("SEG_INFO");
20 /*
21 * Identifier to pass Translated Address Information.
22 */
23 Identifier<gelly *> LINE_INFO("LINE_INFO");
24 /*
25 * Class: SegPicker
26 * This Code Processor performs Reachability Test for each ordered pair of
27 * deadline instructions and lists the possible segments.
28 */
29 class SegPicker : public otawa::Processor
30 {
31 WorkSpace *curr_ws;
32 public:
33 gelly *gel;
34 /*
35 * List of picked segments.
36 */
37 elm::genstruct::SLList<lin_info *> segs;
38 /*
39 * Register Code Processor and declare required Feature.
40 */
41 SegPicker( void): Processor("SegPicker", Version(1, 0, 0))
42 {
43 require(otawa::COLLECTED_CFG_FEATURE);
44 require(otawa::DOMINANCE_FEATURE);
45 }
46 /*
47 * Get the Translated Address Information from the workspace.
48 * @param ws Current workspace.
49 */
50 void setup(WorkSpace *ws)
51 {
52 curr_ws=ws;
53 gel=ptarm::LINE_INFO(ws);
54 }
55 /*
56 * Set the Segment Information for the workspace.
57 */
58 void cleanup(WorkSpace *ws) {ptarm::SEG_INFO(ws)=this;}
59 /*
60 * Print the list of picked segments and their WCETs.
61 */
62 void printSegs()
63 {
64 for(elm::genstruct::SLList<lin_info *> ::Iterator iseg(segs); iseg; iseg++)
65 cout<<"Seg "<<iseg->line"
66 ["<<iseg->dn" - "<<iseg->up"]\tWCET= "
67 "<iseg->wcet""\n";
/* Get WCET of a segment. */
/* @param line Start line number of the segment */
/* @return WCET of the segment starting at line */
/* NOTE: Multiple segments can start at the same line. Highest value is returned. */

long int getWCET(int line)
{
    long int ret=0;
    for(elm::genstruct::SLList<lin_info *>::Iterator iseg(segs); iseg; iseg++)
    if(iseg->line==line && iseg->wcet>ret)
        ret=iseg->wcet;
    return ret;
}

protected:

/* Test all possible ordered pairs of deadline instruction and list the segments. */
/* @param ws current Workspace */
void processWorkSpace(WorkSpace *ws)
{
    lin_info *end1, *end2;
    while(gel->lins.count())
    {
        end1=gel->lins.first();
        for(elm::genstruct::SLList<lin_info *>::Iterator lin(gel->lins);lin;lin++)
        {
            end2=lin;
            if(isSeg(end1,end2))
            {
                lin_info *newseg = new lin_info(*end1);
                newseg->up=end2->up;
                segs.add(newseg);
            }
            if(lin==gel->lins.first())
                continue;
            if(isSeg(end2,end1))
            {
                lin_info *newseg = new lin_info(*end2);
                newSeg->up=end1->up;
                segs.add(newseg);
            }
        }
        gel->lins.removeFirst();
    }
}

private:

/* Perform Reachability Analysis. */
/* @param end1 Translated Address Information for start deadline */
/* @param end2 Translated Address Information for stop deadline */
/* @return true, if possible to reach from start to stop deadline without going via any other deadline */
/* false, otherwise */
bool isSeg(lin_info * end1, lin_info * end2)
{
    address_t start=end1->dn;
    address_t stop=end2->up;
    BasicBlock *startbb=getBB(start,true);
    BasicBlock *stopbb=getBB(stop,false);
    if(startbb->cfg()!=stopbb->cfg())
        return false;
    if(startbb==stopbb)
    {
        if(start==stop)
return false;
else if (start<stop)
    return true;
else
    return reachable(start,stop,NULL);
}
if(reachable(start,stop,NULL))
{
    for(elm::genstruct::SLList<lin_info *>::Iterator mid(gel->lins);mid;mid++)
    {
        if(mid!=end1 && mid!=end2 && startbb->cfg()==getBB(mid->up,false)->cfg() &&
           !reachable(start,stop,mid->up))
            return false;
    }
    return true;
else
    return false;
}

BasicBlock *getBB(address_t addr, bool flag)
{
    for (CFGCollection::Iterator icfg(INVOLVED_CFGS(curr_ws)); icfg; icfg++)
        for (CFG::BBIterator ibb(icfg); ibb; ibb++)
            if(!ibb->isEnd())
                if(flag)
                { //Search Down Address
                    if(ibb->address()<=addr && addr<ibb->address()+ibb->size())
                        return ibb;
                }
                else
                { //Search Up Address
                    if(ibb->address()<addr && addr<=ibb->address()+ibb->size())
                        return ibb;
                }
    return NULL;
}

bool reachable(address_t adr1, address_t adr2, address_t adr3)
{    BasicBlock *bb1=getBB(adr1,true);
    BasicBlock *bb2=getBB(adr2,false);
    BasicBlock *midbb=NULL;
    if(adr3)
    {        midbb=getBB(adr3,false);
        if(bb1==bb2)
            return !cmp(ptarm::GE,adr3,adr1)&cmp(ptarm::LT,adr3,adr2);
        else if(bb1==midbb)
            return cmp(ptarm::LT,adr3,adr1);
    }
    else if(bb1==bb2)
    {        if(adr1<adr2)
return true;
}
elem::genstruct::SLList<otawa::BasicBlock *> bbs;
bbs.addFirst(bb1);
doi
for(BasicBlock::OutIterator outs(bbs.first());outs;outs++)
{
  if(!ptarm::FD_FLAG(outs->target()))
    bbs.addLast(outs->target());
}
ptarm::FD_FLAG(bbs.first())=true;
bbs.removeFirst();
if(adr3 && bbs.first()==midbb)
  break;
while(bbs.count()>0 && bbs.first()!=bb2);
for (CFGCollection::Iterator icfg(INVOLVED_CFGS(curr_ws)); icfg; icfg++)
  for (CFG::BBIterator ibb(icfg); ibb; ibb++)
    if(ptarm::FD_FLAG(ibb))
      ptarm::FD_FLAG(ibb)=false;
if(bbs.count() && bbs.first()==bb2)
  return true;
else if(bbs.count()==0)
  return false;
else if(adr3 && bbs.first()==midbb)
  return false;
/*
 * deadta.cpp
 * Author: subarno
 */

#include <stdlib.h>
#include <elm/io.h>
#include <otawa/ipet.h>
#include <otawa/cfg/CFGBuilder.h>
#include "gel.h"
#include "PTARMBBTime.h"
#include "AddressDomain.h"
#include "SegILPBuilder.h"
#include "SegPicker.h"

#define proc_path ".../otawa-core/share/Otawa/scripts/ptarm/pipeline.xml"
#define mem_path ".../otawa-core/share/Otawa/scripts/ptarm/memory.xml"

using namespace elm;
using namespace otawa;
using namespace ptarm;

/*
 * Segment Matching and Timing Analysis Tool
 * This tool identifies all segments in a program and performs WCET analysis for each segment.
 * @param args Command Line Arguments
 * 1- binary file path
 * 2- source file path
 * 3- "--verbose" [optional]
 */

int main(int argc, char **argv)
{
    try
    {
        if(argc>3)
        {
            cerr<<"Error: Usage- deadta <elf path> <source path> [-verbose]."<<io::endl;
            return 1;
        }
        otawa::PropList props;
        otawa::PROCESSOR_PATH (props) = proc_path; //Set pipeline description path
        otawa::MEMORY_PATH (props) = mem_path; //Set memory description path
        otawa::Processor::VERBOS (props) = false;
        otawa::Manager manager;
        otawa::WorkSpace *ws=manager.load(argv[1], props); //Load binary file
        gelly gel; //Run Address Translation
        gel.get_lines(argv[2],argv[1],"//DEAD()","//DEAD()");
        gel.print_lines();
        if(argc>3)
        {
            otawa::Processor::VERBOS (props) = true;
            ptarm::SUB_CFG(props)=true; //Set for Domain Limited Analysis
            ptarm::LINE_INFO (ws)=&gel;
            SegPicker segs;
            segs.process(ws,props);
            for(elm::genstruct::SLLList<lin_info *>::Iterator iseg(segs.segs);iseg;iseg++)
            {
                //For each picked segment
                if((otawa::Processor::VERBOS (props)))
                {
                    cout<<"#..........................................................\n";
                    cout<<"Seg " <<iseg->line<< "["<<iseg->dn<<" - "<<iseg->up<<"]\n";
                    ws=manager.load(argv[1], props); //Reload binary file
                    ptarm::CFG_START (props)=iseg->dn; //Set START address
                    ptarm::CFG_STOP (props)=iseg->up; //Set STOP address
                    AddressDomainBuilder domain; //Build Address Domain
                    domain.process(ws,props);
                    otawa::TASK_ENTRY (props)=domain.entry_cfg->label(); //Set Entry function
                    PTARMBBTime bbtime; //Domain Limited BB Time Analysis
                    bbtime.process(ws, props);
                }
            }
        }
    }
}
otawa::ipet::EXPLICIT(props) = true;
SegConstraint segc; //Build Segment ILP Constraints
segc.process(ws, props);
SegLoopBreaker loop; //Remove broken loops
loop.process(ws, props);
SegObjective sego; //Build ILP Objective function
sego.process(ws, props);
ipet::WCETComputation comp; //Compute WCET
comp.process(ws);
ipet::WCETCountRecorder bbc; //Record Execution counts
bbc.process(ws, props);
const otawa::CFGCollection *cfgs = INVOLVED_CFGS(ws);
if (otawa::Processor::VERBOSE(props)) {
  for (CFGCollection::Iterator cfg(cfgs); cfg; cfg++)
  {
    cfg->print(cout);
    cout<<io::endl;
    for (CFG::BBIterator bb(cfg); bb; bb++)
    {
      if (!ipet::TIME(bb))
        continue;
      cout<<'\t';
      bb->print(cout);
      if (!bb->isEnd())
        cout<<"\n"
        <<ipet::TIME(bb)<<'\t'<<ipet::COUNT(bb);
      else
        cout<<io::endl;
    }
    cout<<"WCET = "<<ipet::WCET(ws)<<io::endl;
  }
  lseg->wcet=ipet::WCET(ws);
delete ws;
}
segs.printSegs(); //Print Segment Information
FILE *src = fopen(argv[2],"r"); //Open source file in read mode.
FILE *newsrc= fopen(strcat(argv[2],".dead"),"w"); //Create copy of source file.
if (src!=NULL && newsrc!=NULL) {
  int line=0;
  char str[100];
  while (fgets(str,100,src)!=NULL) //Read source file line-by-line
  {
    line++;
    char *pos=strstr(str,"//DEAD()");
    if (pos!=NULL) //If source line is a deadline instruction
    {
      //Write deadline parameter
      pos+=7;
      *pos='0';
      fputs(str,newsrc);
      fprintf(newsrc,"%d",segs.getWCET(line));
      fputs(++pos,newsrc);
    }
    else
      fputs(str,newsrc);
  }fclose(src);
fclose(newsrc);
} else
  cerr<<"ERROR: Cannot write to source file."<<io::endl;
catch(elm::Exception& e)
  {cerr<<"ERROR: "<<e.message()<<io::endl;
}