INSTITUT FÜR INFORMATIK

Timing Analysis for the Precision Timed ARM Processor

Subarno Banerjee

Bericht Nr. 1212
September 2012
ISSN 2192-6247

CHRISTIAN-ALBRECHTS-UNIVERSITÄT

ZU KIEL

Institut für Informatik der
Christian-Albrechts-Universität zu Kiel
Olshausenstr. 40
D – 24098 Kiel

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

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[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[4] is a PRET[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.

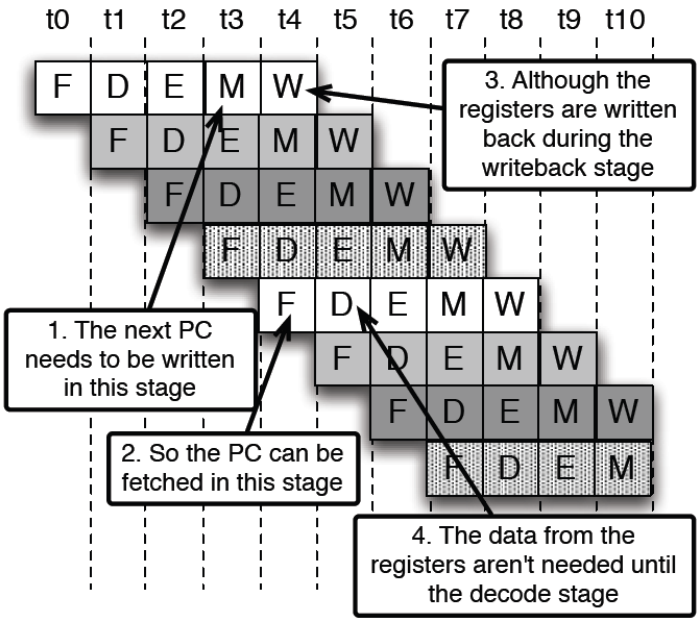


Figure 1[4]: Thread Interleaving on the PTARM

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_{thread} given by-

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

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

2.2 Memory Hierarchy

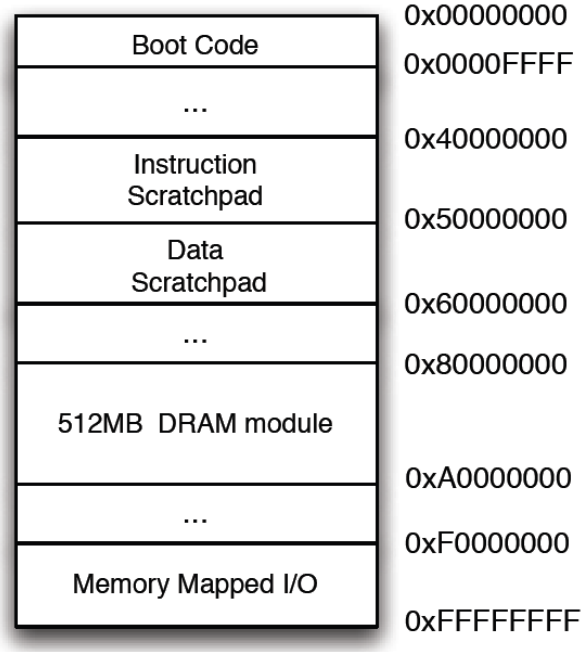


Figure 2[4]: Memory Layout of PTARM

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.

Instruction	Latency
Data Processing	1
Branch	1
Software Interrupt	1

	Accessed Memory Region	
Instruction	**SPM / Boot**	**DRAM**
Load Register	1	4*
Store Register	1	2*
Load Multiple	N	N × 4
Store Multiple	N	N × 2

Table 1[4]: Timing properties of PTARM

* 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*[1] 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. 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-

```
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]    stmdb sp!, {, 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
[000082f8]    strh r0, [fp,-#28]    000082c4 -> DSPM
[000082fc]    strh r2, [fp,-#36]    000082bc -> DSPM
[00008300]    ldrh r3, [fp,-#28]    000082c4 -> DSPM
[00008304]    strh r3, [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
        ---
CFG          BB            CYCLES    COUNT
icrc    ENTRY
        BB 1 (000082e0)        29        1
        BB 2 (00008318)        10        1
        BB 3 (000083d0)        3         1
        BB 4 (000083c4)        3         257
        BB 5 (000083dc)        13        0
        BB 6 (00008410)        3         1
        BB 7 (00008330)        8         256
        BB 8 (00008478)        3         1
        BB 9 (0000841c)        35        1
        BB 10 (00008350)       49        256
        BB 11 (00008544)       4         43
        BB 12 (00008554)       3         1
        BB 13 (00008484)       3         42
        BB 14 (00008560)       3         0
        BB 15 (0000856c)       35        1
        BB 16 (00008490)       27        42
        BB 17 (000084cc)       17        0
        BB 18 (000085c8)       19        1
        BB 19 (000084f8)       25        42
        EXIT
icrc1   ENTRY
        BB 1 (0000820c)        30        256
        BB 2 (000082c4)        3         2304
        BB 3 (000082d0)        15        256
        BB 4 (00008268)        5         2048
        BB 5 (0000827c)        12        2048
        BB 6 (000082ac)        3         0
        BB 7 (000082b8)        3         2048
        EXIT
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):

Line	LowAddrress	HighAddrress
17	00008268	0000827c
20	000082ac	000082b8
41	00008484	00008490
45	000084f8	00008538
57	00008604	00008618
59	00008634	00008658
61	00008674	0000869c

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)[6] 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,
$$x_n = \sum_{i \in in(n)} x_i \quad = \sum_{j \in out(n)} x_j$$
where, $in(n)$ is the set of in-edges to n and , 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:
$$\lambda \times \sum_{i \in in(n) - back(n)} x_i \quad \geq \sum_{b \in back(n)} x_b$$
where, n is a loop header block, back(n) is the set of back edges and λ 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:
$$x_n \quad = \sum_{f \in called(n)} x_{f^0}$$
where, $called(n)$ 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.
$$x_{main^0} = 1$$

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 $x_{main^0} = 1$.
iv. Finally, the Objective function is $\sum_{n \in Domain} t_n \times x_n$.

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)

```
$ 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
---
      BB              CYCLES    COUNT
main
   BB 5 (000082b4)       3       1
   BB 7 (0000827c)       18      1
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.

```
int main()   //@Consumer
{
  //DEAD(136);
  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++)
  {
       //DEAD(21);
       register int tmp = *buf;
       arr[i%8] = tmp;
  }
  return 0;
}
```

6. Future Work

6.1 Segment Annotations

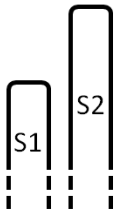
Some synchronization requirements can be derived by studying the code itself. For example- segment ordering between concurrent threads with shared data access can be solved through a Read After Write policy. But it is a difficult task to derive such information and it might not always be possible. Synchronization 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-

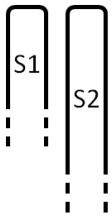
```
int main()   //@Producer
```

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.

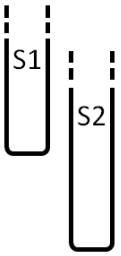
5 types of synchronizations are possible.



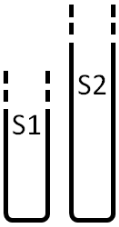
sa- starts after



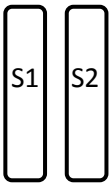
sw- starts with



eb- ends before



ew- ends with



ls- lock step

Example:

```
int main() //@T1
{
   ---
   DEAD() //@S1:
   ---
   DEAD() //@S2: sw
   T2.S2
   ---
   DEAD()
   ---
}
```

```
int main() //@T2
{
   ---
   DEAD() //@S1: sa
   T1.S1; eb T3.S2
   ---
   DEAD() //@S2:
   ---
   DEAD()
   ---
}
```

```
int main() //@T3
{
   ---
   DEAD() //@S1: eb
   T2.S1
   ---
   DEAD() //@S2:
   ---
   DEAD()
   ---
}
```

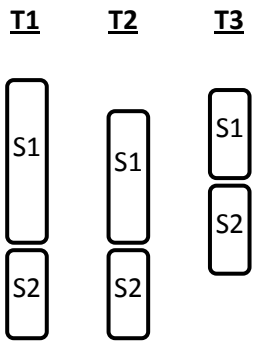


Figure 5: Example usage of segment annotations to specify custom synchronization requirements and one according alignment.

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

Producer

```
int main(){ //@P
   DEAD();   //@a:eb C.a
   volatile unsigned int
   * buf = (unsigned
   int*)(0x3F800200);
   unsigned int i = 0;
   for (i = 0; ; i++ ){
      DEAD(); //@b:ls C.b
      *buf = i;
   }
   return 0;
}
```

Consumer

```
int main(){ //@C
   DEAD();   //@a:sw P.a
   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();//@b:ls O.b
      register int tmp =
      *buf;
      arr[i%8] = tmp;
   }
   return 0;
}
```

Observer

```
int main(){ //@O
   DEAD();   //@a:ew C.a
   volatile unsigned int
   * buf = (unsigned
   int*)(0x3F800200);
   volatile unsigned
   int * fd = (unsigned
   int*)(0x80000600);
   unsigned int i = 0;
   for (i = 0; ; i++ ){
      DEAD();//@b:ls P.b
      *fd = *buf;
   }
   return 0;
}
```

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

6.2 Segment Aligning

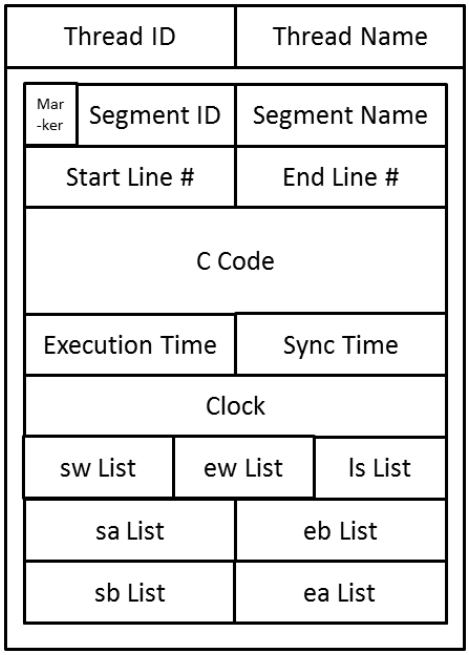


Figure 7: Segment Table structure

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.

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_i^a can be aligned w.r.t S_j^b only if for all y<j, S_y^b are already aligned.
    2. Check clocks of S_i^a and S_j^b. If already aligned then do nothing.
    3. Else, if sa / sw then adjust ST(S_{i-1}^a)
       if eb / ew then adjust ST(S_j^b)
       if ls then adjust ST(S_i^a) and ST(S_j^b)
    4. Update clocks and mark S_i^a.
    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 of 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

[1] Nicholas Jun Hao Ip and Stephen A. Edwards, “A Processor Extension for Cycle-Accurate Real-Time Software”, Embedded and Ubiquitous Computing, Lecture Notes in Computer Science, 2006, Volume 4096/2006, pages 449-458. [pdf]

[2] Stephen A. Edwards and Edward A. Lee, "The Case for the Precision Timed (PRET) Machine", 44th ACM/IEEE Design Automation Conference, 2007, pages 264-265. [pdf]

[3] Ben Lickly, Isaac Liu, Sungjun Kim, Hiren D. Patel, Stephen A. Edwards and Edward A. Lee, "Predictable Programming on a Precision Timed Architecture", In Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES'08), Atlanta, Georgia, USA, October, 2008. [pdf]

[4] Isaac Liu, "Precision Timed Machines", University of California, Berkeley, Technical Report No. UCB/EECS-2012-113, May 14, 2012. [pdf]

[5] Clément Ballabriga, Hugues Cassé, Christine Rochange and Pascal Sainrat, "OTAWA: An Open Toolbox for Adaptive WCET Analysis", Software Technologies for Embedded and Ubiquitous Systems, Lecture Notes in Computer Science, 2011, Volume 6399/2011, pages 35-46. [pdf]

[6] Greger Ottosson and Mikael Sjodin, "Worst-Case Execution Time Analysis for Modern Hardware Architectures", In Proceedings of ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems (LCT-RTS), 1997. [pdf]

[7] OTAWA Homepage- otawa.fr
OTAWA Programming Manual- http://www.otawa.fr/doku/omanual/manual.html
OTAWA API Interface- http://www.otawa.fr/doku/autodoc

APPENDIX

I. PTARM Pipeline description

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- This is the equivalent processor description for a single thread on PTARM. -->
<processor class="otawa::hard::Processor">
  <arch>arm</arch>
  <model>ptarm</model>
  <builder>UC Berkeley</builder>
  <frequency>25000000</frequency>          <!-- Thread execution frequency -->
                                           <!-- PTARM clock frequency = 100MHz -->
  <stages>
      <stage id="TIE">                     <!-- Single stage corresponding to a thread cycle -->
          <name>Thread Interleaved Execution</name>
          <type>EXEC</type>
          <width>1</width>
          <ordered>true</ordered>
          <fus>
              <fu id="MEML">               <!-- Load from DRAM -->
                  <name>MEML</name>
                  <latency>4</latency>
              </fu>
              <fu id="MEMS">               <!-- Store to DRAM -->
                  <name>MEMS</name>
                  <latency>2</latency>
              </fu>
              <fu id="SPM">                <!-- Scratchpad access -->
                  <name>SPM</name>
                  <latency>1</latency>
              </fu>
              <fu id="ALU">                <!-- Data processing -->
                  <name>ALU</name>
                  <latency>1</latency>
              </fu>
          </fus>
          <dispatch>                       <!-- Maps instruction types to functional units -->
              <inst>
                  <type>IS_LOAD</type>
                  <fu ref="MEML"/>
              </inst>
              <inst>
                  <type>IS_STORE</type>
                  <fu ref="MEMS"/>
              </inst>
              <inst>
                  <type>IS_MEM</type>
                  <fu ref="SPM"/>
              </inst>
              <inst>
                  <type>IS_FLOAT</type>
                  <fu ref="ALU"/>
              </inst>
              <inst>
                  <type>IS_MUL</type>
                  <fu ref="ALU"/>
              </inst>
              <inst>
                  <type>IS_DIV</type>
                  <fu ref="ALU"/>
              </inst>
              <inst>
                  <type>IS_INT</type>
                  <fu ref="ALU"/>
              </inst>
              <inst>
                  <type>IS_CONTROL</type>
                  <fu ref="ALU"/>
              </inst>
              <inst>
                  <type>IS_INTERN</type>
                  <fu ref="ALU"/>
              </inst>
          </dispatch>
      </stage>
  </stages>
  <queues>
  </queues>
</processor>
```

II. PTARM Memory Layout description

```xml
<?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>
```

III. PTARMBBTime.h

```
/*
 * 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 Feauture.
    */
   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);
   }
```

```
65    /*
66     * Delete the pipeline and memory descriptions.
67     */
68    void cleanup(WorkSpace *ws)
69    {
70          delete proc;
71          delete mem;
72    }
73 private:
74    /*
75     * Address value analysis
76     * This method attempts to determine the memory address accessed by an instruction.
77     * @param   inst  Memory access instruction
78     * @param   cfg         CFG to which inst belongs
79     * @return              Address accessed by inst
80     *If undeterminable, returns starting address of DRAM (conservative over-estimation)
81     */
82    otawa::Address memref(otawa::Inst *inst, CFG *cfg)
83    {
84          otawa::Address adr =0;
85          elm::StringBuffer *buf = new elm::StringBuffer;
86          inst->dump(*buf);
87          elm::String str = buf->toString();
88          if(str.indexOf('[')>=0)             //Register Indirect Addressing
89          {
90              str=str.substring(str.indexOf('[')+1,str.indexOf(']')-str.indexOf('[')-1);
91              if(str.startsWith("pc"))         //PC-relative
92                 adr=inst->address();    //base address=address of current instruction
93              else if(str.startsWith("fp"))    //FP-relative
94                 adr=cfg->firstInst()->address();    //base address=
                                                  address of first instruction of current CFG
95              if(str.indexOf("ls",1)>=0);
96              else if(str.indexOf('#')>=0)     //Add offset
97              {
98                 if(str.charAt(str.indexOf('#')-1)=='-')
99                       adr -= atoi(str.substring(str.indexOf('#')+1).toCString());
100                else
101                      adr += atoi(str.substring(str.indexOf('#')+1).toCString());
102             }
103         }
104         else if(str.indexOf('#')>=0)         //Immediate addressing
105               adr = atol(str.substring(str.indexOf('#')+1).toCString());
106         if(this->isVerbose())
107         {
108               if(adr.equals(0))
109                     cout<<"\t????????";
110               else
111                     cout<<'\t'<<adr;
112         }
113         delete buf;
114         if(adr.equals(0))              //If address cannot be determined
          //return starting address of DRAM (conservative overestimation)
115               adr=mem->banks().get(mem->banks().count()-2)->address();
116         return adr;
117   }
118 protected:
119   /*
120    * Basic Block Execution Time Computation
121    * Sets otawa::ipet::Time Property of the Basic Block to computed Execution Time.
122    * @param   bb          Basic Block
123    * @param   cfg         CFG to which bb belongs
124    * @param   fw          current Workspace
125    */
126   void processBB(WorkSpace *fw, CFG *cfg, BasicBlock *bb)
127   {
128         otawa::ipet::TIME(bb)=0;
129         if(this->isVerbose())
```

```
130             cout<<"BB "<<bb->number()<<io::endl;
131         if(bb->isEnd())
132             return;
133         for(otawa::BasicBlock::InstIterator inst(bb); inst; inst++)
134         {       //For each Instruction
135             if(sub_cfg_proc && !(ActiveDomain->belongs(inst)))
                //Ignore if Out of Domain
136                 continue;
137             if(this->isVerbose())
138             {
139                 cout<<'['<<inst->address()<<"]    ";
140                 inst->dump(cout);
141             }
142             for(int s=0; s<proc->getStages().count(); s++)
143             {
144                 otawa::hard::Stage *stg=proc->getStages().get(s);
145                 if(stg->getType() == otawa::hard::Stage::EXEC)
146                 {
147                     if(inst->oneOf(otawa::Inst::IS_MEM))
148                     {//If Memory Instruction
149                         if(inst->oneOf(otawa::Inst::IS_MULTI))
150                         {//If Multiple Load/Store
151                             elm::StringBuffer *buf = new elm::StringBuffer;
                            //Count no. of registers
152                             inst->dump(*buf);
153                             elm::String str = buf->toString();
154                             str=str.substring(str.indexOf('{')+1,str.indexOf('}')
                            -str.indexOf('{')-1);
155                             int n = 0;
156                             for(int i=0; i<str.length(); i++)
157                             {
158                                 if(str.charAt(i)==',')
159                                     n++;
160                             }
161                             const otawa::hard::Bank *bnk = mem->get(memref(inst,cfg));
                            //Get accessed memory bank
162                             if(this->isVerbose())
163                                 cout<<" -> "<<bnk->name()<<" "<<n<<io::endl;
164                             if(inst->oneOf(otawa::Inst::IS_LOAD))
                            //If Load, add read latency * no, of registers
165                                 otawa::ipet::TIME(bb)+= bnk->latency()*n;
166                             else if(inst->oneOf(otawa::Inst::IS_STORE))
                            //If Store, add write latency * no, of registers
167                                 otawa::ipet::TIME(bb)+= bnk->writeLatency()*n;
168                         }
169                         else       //If single Load/Store
170                         {
171                             const otawa::hard::Bank *bnk = mem->get(memref(inst,cfg));
                            //Get accessed memory bank
172                             if(this->isVerbose())
173                                 cout<<" -> "<<bnk->name()<<io::endl;
174                             if(inst->oneOf(otawa::Inst::IS_LOAD))
                            //If Load, add read latency
175                                 otawa::ipet::TIME(bb)+= bnk->latency();
176                             else if(inst->oneOf(otawa::Inst::IS_STORE))
                            //If Store, add write latency
177                                 otawa::ipet::TIME(bb)+= bnk->writeLatency();
178                         }
179                     }
180                     else        //Data processing/branch instructions
181                     {
182                         for(int d=0; d<stg->getDispatch().count(); d++)
183                         {
184                             otawa::hard::Dispatch *dsp=stg->getDispatch().get(d);
                            //Get dispatched functional unit
185                             if(inst->oneOf(dsp->getType()))
186                             {
```

```
187                                 if(this->isVerbose())
188                                     cout<<" -> "<<dsp->getFU()->getName()<<io::endl;
                                    //add latency of the functional unit
189                                 otawa::ipet::TIME(bb)+= dsp->getFU()->getLatency();
190                                 break;
191                             }
192                         }
193                     }
194                 }
195                  else
196                       otawa::ipet::TIME(bb)+= stg->getLatency();
197             }
198         }
199   }
200 };
201 }
202 #endif
```

IV. ptarmta.cpp

```
/*
 * 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;
```

```
66                  }
67            }
68            cout<<"WCET = "<<ipet::WCET(ws)<<io::endl;
69            delete ws;
70      }
71      catch(elm::Exception& e)
72            {cerr<<"ERROR: "<<e.message()<<io::endl;}
73  }
```

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] = "asdffeagewaHAFEFaeDsFEawFdsFaefaeerdjgp";

unsigned short 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];
  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)
        cword = ((uchar) jinit) | (((uchar) jinit) << 8);
  else if (jrev < 0)
        cword = 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
  }
  if (jrev >= 0)
        tmp2 = cword;
  else
        tmp2 = rchr[HIBYTE(cword)] | rchr[LOBYTE(cword)] << 8;
  return (tmp2);
}
int main(void) {
  unsigned short i1, 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*)(0x00008000);
   unsigned int i = 0;
   for (i = 0; i ; i++ )
   {
         //DEAD();
         *buf = i;
   }
   return 0;
}
```

```c
/*
 * consumer.c
 */
int main()      //@Consumer
{
   //DEAD();
   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++)
   {                  //@STOP
         //DEAD();    //@START
         register int tmp = *buf;
         arr[i%8] = tmp;
   }
   return 0;
}
```

```c
/*
 * observer.c
 */
int main()      //@Observer
{
   //DEAD();
   volatile unsigned int * buf = (unsigned int*)(0x00008000);
   volatile unsigned int * fd = (unsigned int*)(0x00008301);
   unsigned int i = 0;
   for (i = 0; i ; i++ )
   {
         //DEAD();
         *fd = *buf;
   }
   return 0;
}
```

VII. gel.h (*old*)

```
/*
 * 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
{
private:
   /*
    * 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)
```

```
66          {
67                cerr<<"ERROR: Cannot Load Line Map."<<io::endl;
68                gel_close(file);
69                return;
70          }
71          dwarf_line_iter_t iter;
72          dwarf_location_t loc;
73          while(fgets(str,100,src)!=NULL)           //Read source file line-by-line
74          {
75                lin_info *lin;
76                line++;
77                if(strstr(str,marker)!=NULL)  //If source line is annotated by 'marker'
78                {
79                      lin = new lin_info();         //Store line information and number
80                      lin->file=src_path;
81                      lin->comment=str;
82                      lin->line=line;
83                      for(loc = dwarf_first_line(&iter, map);loc.file;loc =
                              dwarf_next_line(&iter))
84                      {
85                            if(loc.line==line)            //Find line in the Line Map
86                            {
87                                  lin->loadr = loc.low_addr;    //Get line addresses.
88                                  lin->hiadr = loc.high_addr;
89                            }
90                      }
91                      lins.addLast(lin);
92                }
93          }
94          gel_delete_line_map(map);     //delete Line Map
95          gel_close(file);              //Close source and binary files
96          fclose(src);
97    }
98    /*
99     * Print the list of 'marked' lines and corresponding addresses.
100    */
101   void print_segs()
102   {
103         cout<<"Line\tLowAddress\tHighAddress"<<io::endl;
104         for(elm::genstruct::SLList<lin_info *>::Iterator iter(lins); iter; iter++)
105         {
106               cout<<iter->line<<'\t'<<iter->loadr<<'\t'<<iter->hiadr<<io::endl;
107         }
108   }
109 };
```

VIII. gel.h

```
/*
 * 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(*str1!='\0')str1++;
         while(*str2!='\0')str2++;
         while((*str1==*str2)&&(*str1!=terminate)&&(*str2!=terminate))
         {
               str1--;
               str2--;
         }
         if((*str1==*str2)&&(*str1==terminate))
               return 0;
         else
               return 1;
   }
public:
   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.
```

```
65     * @param   src_path          path of C source file
66     * @param   elf_path          path of binary file
67     * @param   start_marker      'marker' string annotated to identify START
68     * @param   stop_marker       'marker' string annotated to identify STOP
69     */
70    void get_lines(char * src_path, char *elf_path, char *start_marker,
      char *stop_marker)
71    {
72          int line=0;
73          char str[100];
74          FILE *src = fopen(src_path,"r");    //Open source file in read mode.
75          if(src == NULL)
76          {
77                cout<<"ERROR: Cannot open "<<src_path<<io::endl;
78                return;
79          }
80          gel_file_t *file;
81          dwarf_line_map_t *map;
82          file = gel_open(elf_path, "", 0);   //Open the ELF binary file
83          if(file == NULL)
84          {
85                cout<<"ERROR: "<<gel_strerror()<<io::endl;
86                return;
87          }
88          map = gel_new_line_map(file);       //Load Line Map of the ELF
89          if(!map)
90          {
91                cerr<<"ERROR: Cannot Load Line Map."<<io::endl;
92                gel_close(file);
93                return;
94          }
95          dwarf_line_iter_t iter;
96          dwarf_location_t loc;
97          int maxlin=0;
98          for(loc = dwarf_first_line(&iter, map);loc.file;loc = dwarf_next_line(&iter))
99          {     //Find highest line number in the Line Map
100               if((revcmp(loc.file,src_path,'/')==0)&&(loc.line>maxlin))
101                     maxlin=loc.line;
102         }
103         while(fgets(str,100,src)!=NULL)     //Read source file line-by-line
104         {
105               lin_info *lin;
106               line++;
107               if(strstr(str,start_marker)!=NULL||strstr(str,stop_marker)!=NULL)
108               {//If source line is annotated by 'marker'
109                     lin = new lin_info();         //Store line information and number
110                     lin->file=src_path;
111                     lin->comment=str;
112                     lin->line=line;
113                     lin->up=0;
114                     lin->dn=0;
115                     for(int i=0; !(lin->up && lin->dn) && !(line-i-1<1) &&
                      !(line+i>maxlin); i++)
116                     {
117                       if(!lin->up)
118                       {  //Find previous line in the Line Map.
119                          for(loc = dwarf_first_line(&iter, map);loc.file;loc =
                           dwarf_next_line(&iter))
120                          {
121                            if((loc.line==line-i-1) &&
                             (revcmp(loc.file,src_path,'/')==0))
122                             {
123                                 if(ptarm::cmp(GT,loc.high_addr,lin->up))
124                                    lin->up=loc.high_addr;
125                             }
126                          }
127                       }
```

```
128                         if(!lin->dn)
129                         {
130                            for(loc = dwarf_first_line(&iter, map);loc.file;loc =
                               dwarf_next_line(&iter))
131                            {  //Find line in the Line Map.
132                               if((loc.line==line+i)&&(revcmp(loc.file,src_path,'/')==0))
133                               {
134                                   if(!lin->dn||ptarm::cmp(LT,loc.low_addr,lin->dn))
135                                       lin->dn=loc.low_addr;
136                               }
137                            }
138                         }
139                      }
140                      lins.addLast(lin);
141                      if(strstr(str,start_marker)!=NULL)
142                            start_adr=lin->dn;
143                      else if(strstr(str,stop_marker)!=NULL)
144                            stop_adr=lin->up;
145               }
146         }
147         gel_delete_line_map(map);      //delete Line Map
148         gel_close(file);               //Close source and elf files
149         fclose(src);
150   }
151   /*
152    * Print the list of 'marked' lines and corresponding addresses.
153    */
154   void print_lines()
155   {
156         cout<<"Line\tUpAddress\tDownAddress"<<io::endl;
157         for(elm::genstruct::SLList<lin_info *>::Iterator liter(lins); liter; liter++)
158               cout<<liter->line<<'\t'<<liter->up<<'\t'<<liter->dn<<io::endl;
159   }
160 };
161 #endif
```

IX. AddressDomainBuilder.h

```
 1 /*
 2  * AddressDomain.h
 3  *       Author: subarno
 4  */
 5 #ifndef ADDRESS_DOMAIN_H_
 6 #define ADDRESS_DOMAIN_H_
 7 #include <stdlib.h>
 8 #include <elm/string/String.h>
 9 #include <otawa/proc/BBProcessor.h>
10 #include <otawa/proc/Feature.h>
11 #include <otawa/prop/Identifier.h>
12 #include <otawa/prop/PropList.h>
13 using namespace otawa;
14 using namespace ipet;
15 namespace ptarm {
16 class AddressDomainBuilder;
17 /*
18  * Identifiers to pass START and STOP addresses.
19  */
20 Identifier<otawa::address_t> CFG_START("CFG_START");
21 Identifier<otawa::address_t> CFG_STOP("CFG_STOP");
22 /*
23  * Flags to mark basic blocks during forward and backward flood.
24  */
25 Identifier<bool> FD_FLAG("FD_FLAG");
26 Identifier<bool> BD_FLAG("BD_FLAG");
27 /*
28  * Flag to mark basic blocks and edges in the domain.
29  */
30 Identifier<bool> DOMAIN_FLAG("DOMAIN_FLAG");
31 /*
32  * Identifier to associate Address Domain with the workspace.
33  */
34 Identifier<AddressDomainBuilder *> ACTIVE_DOMAIN("ACTIVE_DOMAIN");
35 /*
36  * Additional feature provided by AddressDomainBuilder processor.
37  */
38 Feature <AddressDomainBuilder> ADDRESS_DOMAIN_FEATURE("ptarm::ADDRESS_DOMAIN_FEATURE");
39 /*
40  * Compare two address values.
41  * @param      op    Operator. One out of {LT,GT,LE,GE,EQ}
42  * @param      a     Left address operand.
43  * @param      b     Right address operand.
44  * @return     (a op b)
45  */
46 enum compare {LT,GT,LE,GE,EQ};
47 bool cmp(compare op, otawa::address_t a, otawa::address_t b)
48 {
49    elm::t::size as, bs;
50    as=a.page()+a.offset();
51    bs=b.page()+b.offset();
52    if(op==LT)
53          return as<bs;
54    else if(op==GT)
55          return as>bs;
56    else if(op==LE)
57          return as<=bs;
58    else if(op==GE)
59          return as>=bs;
60    else if(op==EQ)
61          return as==bs;
62 }
63 /*
64  * Class: AddressDomainBuilder
65  *       This Code Processor builds and stores the Address Domain.
66  */
```

```
67 class AddressDomainBuilder: public otawa::Processor
68 {
69 private:
70    /*
71     * Class: AddressRange
72     *          This class represents a contiguous address range.
73     */
74    class AddressRange
75    {
76    public:
77          otawa::address_t low, high;
78          /*
79           * Set the addresses.
80           * @param   lo    Lower address bound.
81           * @param   hi    Upper address bound.
82           */
83          AddressRange(otawa::Address lo, otawa::Address hi)
84          {
85                low=lo;
86                high=hi;
87          }
88          /*
89           * Test for belongingness of an address.
90           * @param   adr   Address to test.
91           * @return  true,  if adr belongs to the AddressRange.
92           *                 false, otherwise
93           */
94          bool belongs(otawa::Address adr)
95          {
96                if(cmp(GE,adr,low)&&cmp(LT,adr,high))
97                      return true;
98                return false;
99          }
100   };
101   /*
102    * Address Domain is a list of AdressRange objects.
103    */
104   elm::genstruct::SLList<AddressRange *> adrs;
105 public:
106   /*
107    * START and STOP addresses.
108    */
109   otawa::address_t start_adr, stop_adr;
110   /*
111    * StartBB and StopBB.
112    */
113   otawa::BasicBlock *start_bb, *stop_bb;
114   /*
115    * CFG containing the segment.
116    */
117   otawa::CFG *entry_cfg;
118   /*
119    * Register Code Processor and declare required & provided Feauture.
120    */
121   AddressDomainBuilder(void): Processor("AddressDomainBuilder", Version(1, 0, 0))
122   {
123         require(otawa::COLLECTED_CFG_FEATURE);
124         provide(ptarm::ADDRESS_DOMAIN_FEATURE);
125   }
126   /*
127    * Get the START and STOP addresses from the PropertyList.
128    * @param   props PropertyList with CFG_START & CFG_STOP addresses.
129    */
130   void configure(const PropList& props)
131   {
132         Processor::configure(props);
133         start_adr=ptarm::CFG_START(props);
```

```
134         stop_adr=ptarm::CFG_STOP(props);
135   }
136   /*
137    * Set the Address Domain for the workspace.
138    */
139   void cleanup(WorkSpace *ws) {ptarm::ACTIVE_DOMAIN(ws)=this;}
140   /*
141    * Test for belongingness of an instruction.
142    * @param   ins   Instruction to test.
143    * @return  true,  if ins belongs to the AddressDomain.
144    *                 false, otherwise
145    */
146   bool belongs(otawa::Inst *ins)
147   {
148         if(ins==NULL)
149               return true;
150         //Check for each AddressRange
151         for(elm::genstruct::SLList<AddressRange *>::Iterator iter(adrs); iter; iter++)
152               if(iter->belongs(ins->address()))
                  //If ins->address() belongs to the Address Range, return true
153                     return true
154         return false;     //If belongs to none, return false.
155   }
156 private:
157   /*
158    * Get the last address of a function.
159    * @param   cfg   CFG of required function.
160    * @return  address of the last instruction in cfg.
161    */
162   otawa::address_t lastadr(CFG *cfg)
163   {
164         otawa::address_t lst=0;
165         for(otawa::BasicBlock::InIterator ins(cfg->exit()); ins; ins++)
166         {
167               if(ins->source()->address()+ins->source()->size()>lst)
168                     lst=ins->source()->address()+ins->source()->size();
169         }
170         return lst;
171   }
172   /*
173    * Add a new AddressRange to the AddressDomain.
174    * @param   adr   New AddressRange.
175    * If adr overlaps with an AddressRange in the AddressDomain,
176    * the overlapping AddressRange bounds are changed
177    * otherwise, adr is added to the list of AddressRange.
178    */
179   void add(AddressRange *adr)
180   {
181         if(this->isVerbose())
182               cout<<"add "<<adr->low<<" - "<<adr->high<<io::endl;
183         for(elm::genstruct::SLList<AddressRange *>::Iterator iter(adrs); iter; iter++)
184         {
185               if((iter->belongs(adr->low))&&(iter->belongs(adr->high)))
186               {
187                     delete adr;
188                     return;
189               }
190               else if(iter->belongs(adr->low))
191               {
192                     iter->high=adr->high;
193                     delete adr;
194                     return;
195               }
196               else if(iter->belongs(adr->high))
197               {
198                     iter->low=adr->low;
199                     delete adr;
```

```
200                     return;
201               }
202         }
203         adrs.add(adr);
204   }
205   /*
206    * Add an entire function to the AddressDomain.
207    * Recursively add called functions too.
208    * @param   cfg   CFG of the added function.
209    * NOTE: A CFG occupies a contiguous AddressRange.
210    */
211   void addcfg(otawa::CFG *cfg)
212   {
213         if(this->isVerbose())
214               cout<<"CFG "<<cfg->label()<<io::endl;
215         add(new AddressRange(cfg->firstInst()->address(),lastadr(cfg)));
216         for(otawa::CFG::BBIterator bbs(cfg);bbs;bbs++)
217         {
218               if(bbs->isCall())
219                     for(BasicBlock::OutIterator outs(bbs);outs;outs++)
220                           if(outs->kind()==otawa::Edge::CALL)
221                                 this->addcfg(outs->calledCFG());
222               ptarm::DOMAIN_FLAG(bbs)=true;
223         }
224   }
225 protected:
226   /*
227    * Build the AddressDomain.
228    * @param   ws    Current workspace.
229    */
230   void processWorkSpace(WorkSpace *ws)
231   {
232         const otawa::CFGCollection *cfgs = INVOLVED_CFGS(ws);
233         for(CFGCollection::Iterator cfg(cfgs); cfg; cfg++)
234         {     //Find the CFG containing the START and STOP addresses.
235               if(cmp(LE,cfg->firstInst()->
                     address(),start_adr)&&cmp(GE,lastadr(cfg),start_adr))
236               {
237                     if(cmp(LE,cfg->firstInst()->
                           address(),stop_adr)&&cmp(GE,lastadr(cfg),stop_adr))
238                     {     entry_cfg=cfg;
239                           for(CFG::BBIterator bbs(cfg); bbs; bbs++)
240                           {     //Locate the StartBB and StopBB
241                                 if(cmp(LE,bbs->address(),start_adr) && cmp(GT,bbs->
                                       address()+bbs->size(),start_adr))
242                                       start_bb=bbs;
243                                 if(cmp(LT,bbs->address(),stop_adr) && cmp(GE,bbs->
                                       address()+bbs->size(),stop_adr))
244                                       stop_bb=bbs;
245                           }
246                           break;
247                     }
248                     else
249                     {
250                           cerr<<"Start and Stop addresses must be in same CFG.";
251                           return;
252                     }
253               }
254         }
255         cout<<"---Address Domain---"<<io::endl<<"Entry = ["<<entry_cfg->label()<<"]"
256         <<io::endl<<"Start-"<<start_adr<<" |BB"<<start_bb->number()<<io::endl
257         <<"Stop-"<<stop_adr<<" |BB"<<stop_bb->number()<<io::endl;
258         //Forward Flood Analysis
259         elm::genstruct::SLList<otawa::BasicBlock *> bbs;
260         bbs.addFirst(start_bb);
261         while(bbs.count()>0)
262         {
```

```
263                 if(bbs.first()==stop_bb)
264                 {
265                       ptarm::FD_FLAG(bbs.first())=true;
266                       while(bbs.count()>0)
267                             bbs.removeFirst();
268                       break;
269                 }
270                 for(BasicBlock::OutIterator outs(bbs.first());outs;outs++)
271                 {
272                       if(!ptarm::FD_FLAG(outs->target()))
273                             bbs.addLast(outs->target());
274                 }
275                 ptarm::FD_FLAG(bbs.first())=true;
276                 bbs.removeFirst();
277           }
278           //Backward Flood Analysis
279           bbs.addFirst(stop_bb);
280           while(bbs.count()>0)
281           {
282                 if(bbs.first()==start_bb)
283                 {
284                       ptarm::BD_FLAG(bbs.first())=true;
285                       break;
286                 }
287                 for(BasicBlock::InIterator ins(bbs.first());ins;ins++)
288                 {
289                       if(!ptarm::BD_FLAG(ins->source()))
290                             bbs.addLast(ins->source());
291                 }
292                 ptarm::BD_FLAG(bbs.first())=true;
293                 bbs.removeFirst();
294           }
295           //Build Address Domain
296           for(CFG::BBIterator bbs(entry_cfg); bbs; bbs++)
297           {
298                 if(ptarm::FD_FLAG(bbs) && ptarm::BD_FLAG(bbs))
299                 {     //If basic block is marked during both forward and backward flood.
300                       //Add the Address Range.
301                       if(this->isVerbose())
302                             cout<<"BB "<<bbs->number()<<io::endl;
303                       if(start_bb==bbs || stop_bb==bbs)
304                       {
305                             if(start_bb==stop_bb && start_adr<=stop_adr)
306                                   add(new AddressRange(start_adr,stop_adr));
307                             else
308                             {
309                                   if(start_bb==bbs)
310                                         add(new AddressRange(start_adr,bbs->
                                               address()+bbs->size()));
311                                   if(stop_bb==bbs)
312                                         add(new AddressRange(bbs->address(),stop_adr));
313                             }
314                       }
315                       else
316                             add(new AddressRange(bbs->address(),bbs->address()+bbs->
                                   size()));
317                       //Add called functions to the Address Domain.
318                       if(bbs->isCall())
319                       {
320                             for(BasicBlock::OutIterator outs(bbs);outs;outs++)
321                             {
322                                   if(outs->kind()==otawa::Edge::CALL)
323                                         addcfg(outs->calledCFG());
324                             }
325                       }
326                       ptarm::DOMAIN_FLAG(bbs)=true;
327                 }
```

```
328         }
329         //Mark edges belonging to the domain.
330         for(CFG::BBIterator bbs(entry_cfg); bbs; bbs++)
331         {
332               if(belongs(bbs->firstInst()))
333                     for(BasicBlock::InIterator edges(bbs);edges;edges++)
334                           if(belongs(edges->source()->lastInst()))
335                                 ptarm::DOMAIN_FLAG(edges)=true;
336               if(belongs(bbs->lastInst()))
337                     for(BasicBlock::OutIterator edges(bbs);edges;edges++)
338                           if(belongs(edges->target()->firstInst()))
339                                 ptarm::DOMAIN_FLAG(edges)=true;
340         }
341         if(adrs.count()==0)
342               cerr<<"Error: Address domain could not be built."<<io::endl;
343   }
344 };
345 }
346 #endif
```

X. SegILPBuilder.h

```
1  /*
2   * SegILPBuilder.h
3   *       Author: subarno
4   */
5  #ifndef SEG_ILP_BUILDER_H_
6  #define SEG_ILP_BUILDER_H_
7  #include <otawa/ilp.h>
8  #include <otawa/ipet/IPET.h>
9  #include <otawa/cfg.h>
10 #include <otawa/ipet/VarAssignment.h>
11 #include <otawa/ipet/ILPSystemGetter.h>
12 #include <otawa/util/Dominance.h>
13 #include "AddressDomain.h"
14 using namespace otawa::ilp;
15 namespace ptarm {
16 /*
17  * Class: SegConstraint
18  *       This Code Processor adds the Structural Constraints for basic blocks in the
19  *       domain and Out Of Domain Constraints for basic blocks outside the domain.
20  */
21 class SegConstraint: public BBProcessor
22 {
23    ptarm::AddressDomainBuilder *ActiveDomain;
24 public:
25    /*
26     * Register Code Processor and declare required & provided Feauture.
27     */
28    SegConstraint(void)      : BBProcessor("SegConstraint", Version(1, 0, 0))
29    {
30          require(ASSIGNED_VARS_FEATURE);
31          require(ILP_SYSTEM_FEATURE);
32          require(ptarm::ADDRESS_DOMAIN_FEATURE);
33          provide(CONTROL_CONSTRAINTS_FEATURE);
34    }
35    /*
36     * Get the ILP System and add the Segment Entry Constraint.
37     * @param   ws    Current workspace.
38     */
39    void setup(WorkSpace *ws)
40    {
41          ActiveDomain=ptarm::ACTIVE_DOMAIN(ws);
42          string label = "entry constraint";
43          System *system = SYSTEM(ws);
44          Constraint *cons = system->newConstraint(label,Constraint::EQ, 1);
45          cons->addLeft(1, VAR(ActiveDomain->start_bb));
46          CALLING_CONSTRAINT(ActiveDomain->entry_cfg) = cons;
47    };
48 private:
49    /*
50     * Add Calling Constraint for a function call.
51     * @param   system      ILP System
52     * @param   cfg         CFG of calling function.
53     * @param   bb          Calling basic block.
54     * @param   called      CFG of called function.
55     * @param   var         Execution Count variable for bb.
56     */
57    void addEntryConstraint(System *system, CFG *cfg, BasicBlock *bb, CFG *called,
      otawa::ilp::Var *var)
58    {
59          Constraint *cons = CALLING_CONSTRAINT(called);
60          if(!cons)
61          {
62                string label= _<<"calling constraint for "<<called->label();
63                cons = system->newConstraint(Constraint::EQ);
64                cons->addLeft(1, VAR(called->entry()));
65                CALLING_CONSTRAINT(called) = cons;
```

```
66          }
67          cons->addRight(1, var);
68    }
69 protected:
70    /*
71     * Add Structural Constraint for a basic block.
72     * @param   bb          Basic block.
73     * @param   cfg         CFG to which bb belongs.
74     * @param   ws          Current Workspace.
75     */
76    void processBB(WorkSpace *ws, CFG *cfg, BasicBlock *bb)
77    {
78          Constraint *cons;
79          bool used;
80          CFG *called = NULL;
81          System *system = SYSTEM(ws);
82          otawa::ilp::Var *bbv = VAR(bb);
83          string label;
84          if(!ptarm::DOMAIN_FLAG(bb))
85          {     //Out Of Domain Constraints
86                label = _<<"out of domain constraint of BB"<<INDEX(bb)<<"|"<<cfg->
                      label();
87                cons = system->newConstraint(label, Constraint::EQ,0);
88                cons->addLeft(1, bbv);
89          }
90          else
91          {
92                if(bb!=ActiveDomain->start_bb)
93                {     //Structural Input Constraints
94                      label = _<<"structural input constraint of BB"
                            <<INDEX(bb)<<"|"<<cfg->label();
95                      cons = system->newConstraint(label, Constraint::EQ);
96                      cons->addLeft(1, bbv);
97                      used = false;
98                      for(BasicBlock::InIterator edge(bb); edge; edge++)
99                            if((edge->kind()!=Edge::CALL)&&(ptarm::DOMAIN_FLAG(edge)))
100                           {
101                                 cons->addRight(1, VAR(edge));
102                                 used = true;
103                           }
104                     if(!used)
105                           delete cons;
106               }
107               if(bb!=ActiveDomain->stop_bb)
108               {     //Structural Output Constraints
109                     label = _<<"structural output constraint of BB"
                            <<INDEX(bb)<<"|"<<cfg->label();
110                     bool many_calls = false;
111                     cons = system->newConstraint(label, Constraint::EQ);
112                     cons->addLeft(1, bbv);
113                     used = false;
114                     for(BasicBlock::OutIterator edge(bb); edge; edge++)
115                     {
116                           if(edge->kind() != Edge::CALL)
117                           {
118                                 if(ptarm::DOMAIN_FLAG(edge))
119                                 {
120                                       cons->addRight(1, VAR(edge));
121                                       used = true;
122                                 }
123                           }
124                           else
125                           {
126                                 if(!edge->calledCFG())
127                                       throw ProcessorException(*this, _<<
                                            "unresolved call at "<<bb->address());
128                                 if(called)
```

```
129                                     many_calls = true;
130                               else
131                                     called = edge->calledCFG();
132                           }
133                     }
134                     if(!used)
135                           delete cons;
136                     //Process function call(s)
137                     if(called)
138                     {
139                       if(!many_calls)
140                         addEntryConstraint(system, cfg, bb, called, bbv);
141                       else
142                       {
143                         label <<"multiple call from BB"<<INDEX(bb)<<"|"<<cfg->label();
144                         Constraint *call_cons = system->newConstraint(Constraint::EQ);
145                         call_cons->addLeft(1, bbv);
146                         for(BasicBlock::OutIterator edge(bb); edge; edge++)
147                           if(edge->kind() == Edge::CALL)
148                           {
149                             CFG *called_cfg = edge->calledCFG();
150                             String name;
151                             name = _<<"call_"<<bb->number()<<'_'<<cfg->
                                  label()<<"_to_"<<called_cfg->label();
152                             otawa::ilp::Var *call_var = system->newVar(name);
153                             addEntryConstraint(system,cfg,bb,called_cfg,call_var);
154                             call_cons->addRight(1, call_var);
155                           }
156                       }
157                     }
158               }
159         }
160   }
161 };
162 /*
163  * Class: SegObjective
164  *            This Code Processor builds the Objective function of the ILP System.
165  */
166 class SegObjective: public BBProcessor {
167 public:
168   /*
169    * Register Code Processor and declare required & provided Feauture.
170    */
171   SegObjective(void): BBProcessor("SegObjective", Version(1, 0, 0))
172   {
173         require(ASSIGNED_VARS_FEATURE);
174         require(BB_TIME_FEATURE);
175         require(ILP_SYSTEM_FEATURE);
176         provide(OBJECT_FUNCTION_FEATURE);
177   }
178   /*
179    * Build Objective function.
180    * For each basic block bb in the domain, add TIME(bb)*VAR(bb).
181    * @param   bb          Basic block.
182    * @param   cfg         CFG to which bb belongs.
183    * @param   ws          Current Workspace.
184    */
185   void processBB(WorkSpace *ws,CFG *cfg,BasicBlock *bb)
186   {
187         if(ptarm::DOMAIN_FLAG(bb))
188               SYSTEM(ws)->addObjectFunction(TIME(bb),VAR(bb));
189   }
190 };
191 /*
192  * Class: SegLoopBreaker
193  *      This Code Processor finds broken loops and removes the LOOP_HEADER marker.
194  */
```

```
195 class SegLoopBreaker: public BBProcessor {
196 public:
197   /*
198    * Register Code Processor and declare required & provided Feauture.
199    */
200   SegLoopBreaker(void): BBProcessor("SegLoopBreaker", Version(1, 0, 0))
201   {
202         require(COLLECTED_CFG_FEATURE);
203         require(LOOP_HEADERS_FEATURE);
204   }
205   /*
206    * Set LOOP_HEADER to false for broken loops.
207    * @param   bb          Basic block.
208    * @param   cfg         CFG to which bb belongs.
209    * @param   ws          Current Workspace.
210    */
211   void processBB(WorkSpace *ws,CFG *cfg,BasicBlock *bb)
212   {     //If bb is a loop header block
213        if(otawa::LOOP_HEADER(bb))
214        {    //If bb belongs to the domain
215             if(ptarm::DOMAIN_FLAG(bb))
216             {    //Check if every instruction of bb belongs to the domain
217                  for(BasicBlock::InstIter inst(bb);inst;inst++)
218                    if(!ptarm::ACTIVE_DOMAIN(ws)->belongs(inst))
219                    {
220                      otawa::LOOP_HEADER(bb)=false;
221                      if(this->isVerbose())
222                       cout<<"Loop header out of domain. Broken Loop at BB "
                                    <<bb->number()<<io::endl;
223                      break;
224                    }
225                  //Check if back edge belongs to the domain
226                  BasicBlock *back=NULL;
227                  for(BasicBlock::InIterator ins(bb);ins;ins++)
228                    if(Dominance::dominates(bb,ins->source()))
229                    {
230                      if(!ptarm::DOMAIN_FLAG(ins))
231                      {
232                           otawa::LOOP_HEADER(bb)=false;
233                           if(this->isVerbose())
234                                 cout<<"Back edge "<<ins->source()->number()<<"-"<<
                                       ins->target()->number()<<" is out of domain. "
235                                 <<"Broken Loop at BB "<<bb->number()<<io::endl;
236                      }
237                      back=ins->source();
238                      break;
239                    }
240                  //Check if loop entry edge belongs to the domain
241                  if(back)
242                    for(BasicBlock::OutIterator outs(bb);outs;outs++)
243                    if(outs->target()==back||Dominance::dominates(outs-> target(),back))
244                      if(!ptarm::DOMAIN_FLAG(outs))
245                      {
246                       otawa::LOOP_HEADER(bb)=false;
247                       if(this->isVerbose())
248                          cout<<"Loop entry edge "<<outs->source()->number()<<"-"<<
                              outs->target()->number()<<" is out of domain. "
249                          <<"Broken Loop at BB "<<bb->number()<<io::endl;
250                      }
251             }
252                else
253                     otawa::LOOP_HEADER(bb)=false;
254         }
255   }
256 };
257 }
258 #endif
```

XI. segta.cpp

```
/*
 * segta.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"
#define proc_path    ".../otawa-core/share/Otawa/scripts/ptarm/pipeline.xml"
#define mem_path     ".../otawa-core/share/Otawa/scripts/ptarm/memory.xml"
#define start_marker "//@START"
#define stop_marker  "//@STOP"
using namespace elm;
using namespace otawa;
using namespace ptarm;
/*
 * Segment Timing Analysis Tool
 *
 * This tool runs the timing analysis for a code segment. The code segment is specified
 * by annotating "//@STAR" and "//@STOP" markers in the source file.
 * @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- segta <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::VERBOSE(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],start_marker,stop_marker);
         gel.print_lines();
         if(argc>3)
               otawa::Processor::VERBOSE(props) = true;
         ptarm::SUB_CFG(props)=true;                   //Set for Domain Limited Analysis
         ptarm::CFG_START(props)=gel.start_adr;        //Set START address
         ptarm::CFG_STOP(props)=gel.stop_adr;          //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
```

```
66          comp.process(ws);
67          ipet::WCETCountRecorder bbc;              //Record Execution counts
68          bbc.process(ws, props);
69          if(otawa::Processor::VERBOSE(props))
70          {
71                otawa::ilp::System *sys = otawa::ipet::SYSTEM(ws);
72                sys->dumpSystem(cout);
73                sys->dumpSolution(cout);
74          }
75          const otawa::CFGCollection *cfgs = INVOLVED_CFGS(ws);
76          for(CFGCollection::Iterator cfg(cfgs); cfg; cfg++)
77          {
78                cfg->print(cout);
79                cout<<io::endl;
80                for(CFG::BBIterator bb(cfg); bb; bb++)
81                {
82                      if(!ipet::TIME(bb))
83                            continue;
84                      cout<<'\t';
85                      bb->print(cout);
86                      if(!bb->isEnd())
87                            cout<<"   "<<ipet::TIME(bb)<<'\t'<<ipet::COUNT(bb);
88                      else
89                            cout<<io::endl;
90                }
91          }
92          cout<<"WCET = "<<ipet::WCET(ws)<<io::endl;
93          delete ws;
94    }
95    catch(elm::Exception& e)
96          {cerr<<"ERROR: "<<e.message()<<io::endl;}
97 }
```

XII. SegPicker.h

```
/*
 * SegPicker.h
 *       Author: subarno
 */
#ifndef SEGPICKER_H_
#define SEGPICKER_H_
#include <stdlib.h>
#include <elm/string/String.h>
#include <otawa/ipet/ConstraintLoader.h>
#include <otawa/util/Dominance.h>
#include "gel.h"
using namespace elm;
using namespace otawa;
namespace ptarm {
class SegPicker;
/*
 * Identifier to associate Segment Information with the workspace.
 */
Identifier<SegPicker *> SEG_INFO("SEG_INFO");
/*
 * Identifier to pass Translated Address Information.
 */
Identifier<gelly *> LINE_INFO("LINE_INFO");
/*
 * Class: SegPicker
 *       This Code Processor performs Reachability Test for each ordered pair of
 *       deadline instructions and lists the possible segments.
 */
class SegPicker : public otawa::Processor
{
   WorkSpace *curr_ws;
public:
   gelly *gel;
   /*
    * List of picked segments.
    */
   elm::genstruct::SLList<lin_info *> segs;
   /*
    * Register Code Processor and declare required Feauture.
    */
   SegPicker(void): Processor("SegPicker", Version(1, 0, 0))
   {
         require(otawa::COLLECTED_CFG_FEATURE);
         require(otawa::DOMINANCE_FEATURE);
   }
   /*
    * Get the Translated Address Information from the workspace.
    * @param   ws    Current workspace.
    */
   void setup(WorkSpace *ws)
   {
         curr_ws=ws;
         gel=ptarm::LINE_INFO(ws);
   }
   /*
    * Set the Segment Information for the workspace.
    */
   void cleanup(WorkSpace *ws) {ptarm::SEG_INFO(ws)=this;}
   /*
    * Print the list of picked segments and their WCETs.
    */
   void printSegs()
   {
         for(elm::genstruct::SLList<lin_info *>::Iterator iseg(segs); iseg; iseg++)
               cout<<"Seg "<<iseg->line<<" ["<<iseg->dn<<" - "<<iseg->up<<"]\tWCET= "
                     <<iseg->wcet<<"\n";
```

```
66    }
67    /*
68     * Get WCET of a segment.
69     * @param   line  Start line number of the segment
70     * @return  WCET of the segment starting at line
71     * NOTE: Multiple segments can start at the same line. Highest value is returned.
72     */
73    long int getWCET(int line)
74    {
75          long int ret=0;
76          for(elm::genstruct::SLList<lin_info *>::Iterator iseg(segs); iseg; iseg++)
77                if(iseg->line==line && iseg->wcet>ret)
78                      ret=iseg->wcet;
79          return ret;
80    }
81 protected:
82    /*
83     * Test all possible ordered pairs of deadline instruction and list the segments.
84     * @param   ws    current Workspace
85     */
86    void processWorkSpace(WorkSpace *ws)
87    {
88          lin_info *end1, *end2;
89          while(gel->lins.count())
90          {
91              end1=gel->lins.first();
92              for(elm::genstruct::SLList<lin_info *>::Iterator lin(gel->lins);lin;lin++)
93              {
94                  end2=lin;
95                  if(isSeg(end1,end2))
96                  {
97                      lin_info *newseg = new lin_info(*end1);
98                      newseg->up=end2->up;
99                      segs.add(newseg);
100                 }
101                 if(lin==gel->lins.first())
102                     continue;
103                 if(isSeg(end2,end1))
104                 {
105                     lin_info *newseg = new lin_info(*end2);
106                     newseg->up=end1->up;
107                     segs.add(newseg);
108                 }
109             }
110             gel->lins.removeFirst();
111         }
112   }
113 private:
114   /*
115    * Perform Reachability Analysis.
116    * @param   end1  Translated Address Information for start deadline
117    * @param   end2  Translated Address Information for stop deadline
118    * @return  true,       if possible to reach from start to stop deadline
                             without going via any other deadline
119    *                false,      otherwise
120    */
121   bool isSeg(lin_info * end1, lin_info * end2)
122   {
123         address_t start=end1->dn;
124         address_t stop=end2->up;
125         BasicBlock *startbb=getBB(start,true);
126         BasicBlock *stopbb=getBB(stop,false);
127         if(startbb->cfg()!=stopbb->cfg())
128               return false;
129         if(startbb==stopbb)
130         {
131               if(start==stop)
```

```
132                     return false;
133               else if (start<stop)
134                     return true;
135               else
136                     return reachable(start,stop,NULL);
137         }
138         if(reachable(start,stop,NULL))
139         {
140            for(elm::genstruct::SLList<lin_info *>::Iterator mid(gel->lins);mid;mid++)
141            {
142                if(mid!=end1 && mid!=end2 && startbb->cfg()==getBB(mid->up,false)->cfg()
&&
143                !reachable(start,stop,mid->up))
144                     return false;
145            }
146               return true;
147         }
148         else
149               return false;
150   }
151   /*
152    * Get Basic Block containing a given address
153    * @param   addr  Search address
154    * @param   flag  Search down / up address
155    * @return  pointer to basic block containing addr
156    */
157   BasicBlock *getBB(address_t addr, bool flag)
158   {
159         for (CFGCollection::Iterator icfg(INVOLVED_CFGS(curr_ws)); icfg; icfg++)
160            for (CFG::BBIterator ibb(icfg); ibb; ibb++)
161                if(!ibb->isEnd())
162                   if(flag)
163                   { //Search Down Address
164                     if(ibb->address()<=addr && addr<ibb->address()+ibb->size())
165                           return ibb;
166                   }
167                   else
168                   { //Search Up Address
169                     if(ibb->address()<addr && addr<=ibb->address()+ibb->size())
170                           return ibb;
171                   }
172         return NULL;
173   }
174   /*
175    * Perform Reachability Test.
176    * @param   adr1  Start Address
177    * @param   adr2  Stop Address
178    * @param   adr3  Mid Address [Optional]
179    * @return  true,       if adr2 can be reached from adr1 without going via adr3
180    *                false,      otherwise
181    */
182   bool reachable(address_t adr1, address_t adr2, address_t adr3)
183   {
184         BasicBlock *bb1=getBB(adr1,true);
185         BasicBlock *bb2=getBB(adr2,false);
186         BasicBlock *midbb=NULL;
187         if(adr3)
188         {
189               midbb=getBB(adr3,false);
190               if(bb1==bb2)
191                     return !cmp(ptarm::GE,adr3,adr1)&&cmp(ptarm::LT,adr3,adr2);
192               else if(bb1==midbb)
193                     return cmp(ptarm::LT,adr3,adr1);
194         }
195         else if(bb1==bb2)
196         {
197               if(adr1<adr2)
```

```
198                     return true;
199             }
200             elm::genstruct::SLList<otawa::BasicBlock *> bbs;
201             bbs.addFirst(bb1);
202             do
203             {
204                   for(BasicBlock::OutIterator outs(bbs.first());outs;outs++)
205                   {
206                         if(!ptarm::FD_FLAG(outs->target()))
207                               bbs.addLast(outs->target());
208                   }
209                   ptarm::FD_FLAG(bbs.first())=true;
210                   bbs.removeFirst();
211                   if(adr3 && bbs.first()==midbb)
212                         break;
213             }while(bbs.count()>0 && bbs.first()!=bb2);
214             for (CFGCollection::Iterator icfg(INVOLVED_CFGS(curr_ws)); icfg; icfg++)
215                   for (CFG::BBIterator ibb(icfg); ibb; ibb++)
216                         if(ptarm::FD_FLAG(ibb))
217                               ptarm::FD_FLAG(ibb)=false;
218             if(bbs.count() && bbs.first()==bb2)
219                   return true;
220             else if(bbs.count()==0)
221                   return false;
222             else if(adr3 && bbs.first()==midbb)
223                   return false;
224       }
225 };
226 }
227 #endif
```

XIII. deadta.h

```
1  /*
2   * deadta.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 #include "SegPicker.h"
14 #define proc_path    ".../otawa-core/share/Otawa/scripts/ptarm/pipeline.xml"
15 #define mem_path     ".../otawa-core/share/Otawa/scripts/ptarm/memory.xml"
16 using namespace elm;
17 using namespace otawa;
18 using namespace ptarm;
19 /*
20  * Segment Matching and Timing Analysis Tool
21  *
22  * This tool identifies all segments in a program and performs WCET analysis
      for each segment.
23  * @param args Command Line Arguments
24  *                   1- binary file path
25  *                   2- source file path
26  *                   3- "--verbose"          [optional]
27  */
28 int main(int argc, char **argv)
29 {
30    try
31    {
32          if(argc<3)
33          {
34                cerr<<"Error: Usage- deadta <elf path> <source path>
                        [-verbose]."<<io::endl;
35                return 1;
36          }
37          otawa::PropList props;
38          otawa::PROCESSOR_PATH(props) = proc_path; //Set pipeline description path
39          otawa::MEMORY_PATH(props) = mem_path;     //Set memory description path
40          otawa::Processor::VERBOSE(props) = false;
41          otawa::Manager manager;
42          otawa::WorkSpace *ws=manager.load(argv[1], props);    //Load binary file
43          gelly gel;                                 //Run Address Translation
44          gel.get_lines(argv[2],argv[1],"//DEAD()","//DEAD()");
45          gel.print_lines();
46          if(argc>3)
47                otawa::Processor::VERBOSE(props) = true;
48          ptarm::SUB_CFG(props)=true;                //Set for Domain Limited Analysis
49          ptarm::LINE_INFO(ws)=&gel;
50          SegPicker segs;                            //Run Segment Picker
51          segs.process(ws,props);
52          for(elm::genstruct::SLList<lin_info *>::Iterator iseg(segs.segs);iseg;iseg++)
53          {     //For each picked segment
54                if(otawa::Processor::VERBOSE(props))
55                cout<<"#######################################################\n";
56                cout<<"Seg "<<iseg->line<<" ["<<iseg->dn<<" - "<<iseg->up<<"]\n";
57                ws=manager.load(argv[1], props);    //Reload binary file
58                ptarm::CFG_START(props)=iseg->dn;   //Set START address
59                ptarm::CFG_STOP(props)=iseg->up;    //Set STOP address
60                AddressDomainBuilder domain;        //Build Address Domain
61                domain.process(ws,props);
62                otawa::TASK_ENTRY(props)=domain.entry_cfg->label();//Set Entry function
63                PTARMBBTime bbtime;                 //Domain Limited BB Time Analysis
64                bbtime.process(ws, props);
```

```
65                otawa::ipet::EXPLICIT(props) = true;
66                SegConstraint segc;                    //Build Segment ILP Constraints
67                segc.process(ws,props);
68                SegLoopBreaker loop;                   //Remove broken loops
69                loop.process(ws,props);
70                SegObjective sego;                     //Build ILP Objective function
71                sego.process(ws,props);
72                ipet::WCETComputation comp;            //Compute WCET
73                comp.process(ws);
74                ipet::WCETCountRecorder bbc;           //Record Execution counts
75                bbc.process(ws, props);
76                const otawa::CFGCollection *cfgs = INVOLVED_CFGS(ws);
77                if(otawa::Processor::VERBOSE(props))
78                {
79                      for(CFGCollection::Iterator cfg(cfgs); cfg; cfg++)
80                      {
81                            cfg->print(cout);
82                            cout<<io::endl;
83                            for(CFG::BBIterator bb(cfg); bb; bb++)
84                            {
85                                  if(!ipet::TIME(bb))
86                                        continue;
87                                  cout<<'\t';
88                                  bb->print(cout);
89                                  if(!bb->isEnd())
90                                     cout<<" "<<ipet::TIME(bb)<<'\t'<<ipet::COUNT(bb);
91                                  else
92                                     cout<<io::endl;
93                            }
94                      }
95                      cout<<"WCET = "<<ipet::WCET(ws)<<io::endl;
96                }
97                iseg->wcet=ipet::WCET(ws);
98                delete ws;
99          }
100         segs.printSegs();                            //Print Segment Information
101         FILE *src = fopen(argv[2],"r");              //Open source file in read mode.
102         FILE *newsrc= fopen(strcat(argv[2],".dead"),"w");//Create copy of source file.
103         if(src!=NULL && newsrc!=NULL)
104         {
105               int line=0;
106               char str[100];
107               while(fgets(str,100,src)!=NULL)      //Read source file line-by-line
108               {
109                     line++;
110                     char *pos=strstr(str,"//DEAD()");
111                     if(pos!=NULL)      //If source line is a deadline instruction
112                     {     //Write deadline parameter
113                           pos+=7;
114                           *pos='\0';
115                           fputs(str,newsrc);
116                           fprintf(newsrc,"%d)",segs.getWCET(line));
117                           fputs(++pos,newsrc);
118                     }
119                     else
120                           fputs(str,newsrc);
121               }
122               fclose(src);
123               fclose(newsrc);
124         }
125         else
126               cerr<<"ERROR: Cannot write to source file."<<io::endl;
127   }
128   catch(elm::Exception& e)
129         {cerr<<"ERROR: "<<e.message()<<io::endl;}
130 }
```