HARD REAL-TIME OPERATING SYSTEM KERNEL STUDY

TASK 8, Volume E

IMPLEMENTATION OF THE WORST-CASE EXECUTION ANALYSER

Charles Forsyth
York Software Engineering Limited, UK
July 31st 1992

Abstract

This document discusses the design and implementation of the Worst-case Execution Analyser (hereafter 'the analyser'), specified in the Task 4 deliverable, *Definition of the Tools*[1].

ESTEC/Contract No. 9198/90/NL/SF

ESTEC TECHNICAL MANAGER: Mr F.Gomez-Molinero

© York Software Engineering Limited, 1992

TABLE OF CONTENTS

SECTION	TITLE	PAGE
1	INTRODUCTION	1
2	RESTRICTIONS ON ADA	2
2.1	Pragma LOOPCOUNT	2
2.2	Further Requirements of This Implementation	3
3	IMPLEMENTATION OVERVIEW	4
4	OBJECT CODE ANALYSIS	4
4.1	Disassembler	4
4.2	Basic Blocks and Flow Graphs	6
4.3	Ada Structure Graph	7
4.4	Instruction Timing Data	9
4.5	Correlating Basic Blocks and Ada Blocks	10
4.6	Output Files	10
4.7	Program Size	10
5	SEGMENTS AND SEGMENT TREES	11
6	PROBLEMS PECULIAR TO THIS IMPLEMENTATION	12
6.1	Name Mapping	13
6.2	Code Generated Out of Order	14
7	EXAMPLE	15
	REFERENCES	17
A	FORMAT OF THE .B INTERMEDIATE FILE	18
B	FORMAT OF THE .W INTERMEDIATE FILE	22
C	FORMAT OF THE .S INTERMEDIATE FILE	25
D	RESOLVER OUTPUT	27

1. INTRODUCTION

The analyser estimates the time that will be spent in sections of an Ada program, specifically the time between task interactions (or references to protected tasks). It must therefore find or estimate the execution time of the corresponding object code. There are at least three ways:

1. Change the Ada compiler to account for the execution time of instructions as it generates them.
2. Read the generated instructions from the compiler's assembly code output.
3. Read the instructions from the generated object code.

The first technique is used by some systems, notably the Real-Time Euclid compiler[2]. It is easy to do when the code generator is written with that in mind; it can be harder to add later. For instance, the York Ada compiler produces assembly-language text, not binary code, using a type of macro expansion to piece together instructions and sequences of instructions. Most macro strings are found together in a few tables, but not all, and it would still require extensive work to add timing information to those strings.

The last two methods are also more general, since they might be used to analyse languages other than Ada, such as C or assembly code that might be used in Ada run-time support. Furthermore, although assembly code has the same logical content as object code, reading the latter ensures that the analyser sees precisely what the machine will see, and need not reproduce the actions of the assembler (or attempt to predict them). For instance, an assembler might produce different object code instructions for the same assembly source text, depending on operand values or the distance of a branch from its target label. Some assemblers for RISC machines go further: for instance, those from Sun (SPARC) and MIPS, can optionally do peephole optimisation, instruction scheduling, and basic block movement within a module. All these transformations affect execution times. Object code analysis is therefore used here.

Having only the machine code to hand is not always sufficient. The executable part of an Ada program includes `if`, `loop`, and `case` statements. Unduly pessimistic times are obtained if the times for the `then` and `else` arms of an `if` statement are just added together, since only one of its arms will be chosen by any execution of the `if`. Similarly, the number of times a `loop` body is executed must be known when estimating its execution time. An analyser must therefore know which machine instructions lie in the body of a loop, or in the arms of a multi-way selector such as `if` or `case`. At the object code level, however, although there is much-needed detail, the larger structural cues in the source have been lost.

Some of the information required might be recovered by flow analysis techniques similar to those used by optimising compilers: the object code can be divided into basic blocks, and a flow graph built with which one can detect multi-way branches, and find natural loops and loop headers[3]. For example, the UNIX program *struct[4]* successfully uses similar algorithms to convert an (apparently) unstructured Fortran program into a structured program in Ratfor.

The current system instead uses a hybrid technique to associate instruction-level timing data with the correct construction in the Ada source program. The York Ada compiler adds the structural data from the source code to the symbolic debugging section of each file of object code. This requires only a few lines of code to be added in a few places, unlike instrumenting all generated code. A separate program takes the final object file, disassembles it, and breaks the resulting instruction sequence into (machine code) basic blocks. It uses the structural summary from the compiler to assign the time for each basic block to the corresponding high-level construction (eg, `then` clause or `loop` body).

2. RESTRICTIONS ON ADA

The main restriction is: any construction the programmer uses in the timing critical regions of the program must not take an arbitrarily long time to execute. This constraint does not apply to sections of a program that are not time-critical (for instance, initialisation sections of a program, such as the elaboration of library package specifications).

2.1. Pragma LOOPCOUNT

The most obvious source of difficulty is a `loop` statement where it is not known in general until run-time, how many times the loop body will run. When such a loop appears, the current system requires the programmer to use a new pragma `LOOPCOUNT` to tell it the maximum number of times the loop will run:

```
pragma LOOPCOUNT(1000);
while f(j) loop
      L;
end loop;
```

The pragma takes one parameter, which must be a static universal integer. The pragma must appear immediately before the loop. In the example above, without knowledge of the Boolean function *f*, but given the pragma, the analyser can for timing purposes allow for 1001 executions of `f(j)` and 1000 executions of the loop body *L*. (If the programmer's estimate is exceeded, the current task will exhaust its budgeted time, and an exception will be raised.)

The compiler issues a warning when it encounters a loop without a preceding `pragma LOOPCOUNT`, except in the following two special cases.

When the compiler sees a `for` loop with constant bounds, for example:

```
for i in 1 .. 100 loop
      L;
end loop;
```

it can deduce from the loop bounds that loop body *L* will run at most 100 times, saving the programmer from having to write an explicit `LOOPCOUNT`.

A loop of the form:

```
loop
      -- do something
end loop;
```

that implements a cycle need not have a `LOOPCOUNT`. (Internally, the compiler uses a loop count of one to represent an infinite loop or cycle.)

2.2. Further Requirements of This Implementation

The following further restrictions are necessary for Ada source to be timed in this implementation.

• *Bodies and specifications in separate source files*

This condition is imposed to allow the time for a library unit's initialisation initialisation (the code generated for a library unit's specification) to be kept separate from the time for the execution of the unit's body.

• *Only one library unit body per source file*

The York Ada compiler builds a library (archive of object files) for any source file containing more than one specification or body. The analyser program *wet0* cannot read a library. Also, when several units are presented at once for analysis, the UNIX link editor is called to bind their object files together. It will not include an object file from a library unless it needs a symbol it defines; another program would be required to generate a list of references to force the object files in the library to be used. For our purposes, it is simpler to restrict the assignment of source text to source files.

• *No GOTO statements*

A `goto` statement either cuts short a construction (eg, a loop) by jumping out of it, or implements a loop by jumping back to an earlier label. (Ada prevents a `goto` from jumping into a loop or conditional clause.) In the first case, analyser could ignore the possibility of an early exit to find the worst case time, as it does with the `exit` statement. The second case would require a pragma `LOOPCOUNT` from the programmer, and the loop's structural markers to be generated internally by the compiler. This case is hard to detect in the current compiler. The simplest solution is to ban all `goto` statements. The compiler issues a diagnostic when it sees one.

• *No recursion, including mutual recursion*

As with `goto` statements, a set of recursive calls might or might not give rise to a loop. The compiler does not ban such calls (it cannot detect them in general), but one of the analyser programs described below (*segs*) rejects a program containing a nested sequence of calls more than 50 calls deep. This also detects and rejects recursive calls.

• *No hidden loops with non-static bounds*

A 'hidden loop' is one the compiler generates to implement Ada operations on arrays, such as assignment, comparison, or concatenation. Provided the bounds of the array(s) are static, the compiler will automatically generate the internal equivalent of a `pragma LOOPCOUNT` for an array operation. A hidden loop can also be generated to initialise a subprogram's stack frame, and to create aggregates containing arrays. Thus, the size of all objects allocated on the run-time stack must be known statically, as must the size of any aggregate created. (The compiler issues a diagnostic if it finds operations on objects of non-static size when running in the 'ESA' mode.)

• *Elaboration checks must be suppressed*

For reasons given in §4.2, the run-time checks required to implement this must be suppressed when the code is to be analysed.

3. IMPLEMENTATION OVERVIEW

Two programs are basic to the analyser. The first, *wet0*, collects data from the object file, and writes a summary to intermediate files. The second program, *segs*, builds a special graph structure from that data. It tries to reduce the graph to a singleton node that tells the worst-case time for the unit represented by the object file, by resolving internal function calls and finding the worst-case path of multi-way selections, all the while taking loop counts into account. It produces two types of output, depending on options. The overall flow is summarised in the diagram below:

Remaining sections give details of the design and implementation of each program.

4. OBJECT CODE ANALYSIS

The object code is analysed in three phases by the program *wet0*. First, an object file is disassembled. Next, the code is split into basic blocks. Finally, a graph describing the Ada source program's structure is traversed, and each basic block is assigned to the corresponding source program segment.

The object file format is that used by SunOS on the Sun-3 and Sun-4 for statically-linked programs in releases before Solaris 2.0 (UNIX System V.4)[5]. (It is basically the original UNIX `a.out` format.) For our purposes, an object file has a *text* section containing the program's object code, a *data* section containing statically-initialised data, two optional sections containing relocation information for the text and data, and a symbol table (which optionally includes extra information for debuggers). Symbol table entries can refer to any location in text or data; thus, any desired data can be associated with a given location in a program. *Wet0* requires the relocation sections and symbol table (as discussed below).

4.1. Disassembler

The heart of the analyser is a form of disassembler. It reads the text segment of the object file, converting each machine instruction and its operand descriptors into a convenient internal form, noting in small data structures the specific variant of each operation and the addressing modes of its operands.

Each operand of a disassembled instruction is represented by a value of a (machine dependent) record type *Address*, with the following structure:

```
type Register is (D0, ..., D7, A0, ..., A7,
                  PC, CCR, SR, USP, None);
type Displacement is long_integer;
type Size is (None, Byte, Word, Long);

type Addressing_mode is (Data, Address, Indirect,
      Autoincrement, ...);

type Address is record
      mode: Addressing_mode;
      reg: Register;
      index_reg: Register;
      scale: integer range 1..4; -- specifically 1,2,4
      disp: Displacement;
      index_size: Size;
      displacement_size: Size;
end record;
```

The machine's addressing modes are given enumerated type names by the type *Addressing_mode*. Each instance of the type Address saves in a convenient form the interesting values that were packed and encoded in the machine's representation. Note that some of this information is really only essential during debugging (to check the disassembler's output against the assembly language). For example, the scaling of index values does not affect timing. The 68020 'outer displacement' addressing mode is not implemented, since our compilers do not use it (it happens to be slower than the equivalent sequence of instructions).

Each machine operation (or interesting variant) is assigned an internal name which is a member of the enumerated data type *Opcode*. A complete instruction is represented by a value of a record type *Instruction*, which links the instruction's Opcode to its operand Addresses:

```
type Instruction(na: Number_of_Addresses) is record
      op: Opcode;
      operands: array(1..na) of Address;
end record;
```

The use of such internal representations for instructions and addresses simplifies the rest of the analyser, since it hides the messy details of the machine's representation.

The complexity of this phase is directly related to the complexity of the target machine's instruction and address encoding. Older machines often have many instruction formats, and many addressing modes. In particular, the 68020 has 18 addressing modes, and 15 groups of instructions. Details of the encoding of any two instructions can differ within one group. New instructions and addressing modes were squeezed into the original 68000 to form first the 68010, then the 68020, making the encoding more idiosyncratic. Disassembly code would be simpler with an orthogonal order code like those of the SPARC, MIPS, or POWER architectures, since there are few instructions formats and fewer addressing modes. Just over 20% of *wet0* is dedicated to 68020 disassembly.

Two special cases must be considered by the current implementation: instruction alignment, and tables for indexed jumps. Owing to the design of the instruction cache and prefetch system on the 68020, it is best to align the targets of a jump or call on a 32-bit longword boundary. Since instructions are multiples of 16 bits, it is sometimes necessary to add a 16-bit padding value to align the location counter before placing a label. This often occurs immediately after after a return instruction, since an alignment directive is issued before each procedure. The assembler uses a value of 0 as the padding value. This is an unfortunate choice, since a zero word corresponds to the prefix of a valid multi-word instruction (a variant of 'or byte'), confusing the disassembler. To avoid this, the disassembler skips a zero word following a return instruction. This is safe, since no Ada procedure can begin with the instruction word 0.

The second difficulty is caused by the indexed jumps generated for some Ada `case` statements. The indexed jump is immediately followed by a table of pointers to the code for each `when` clause. The disassembler must distinguish the table's contents from the instruction stream. Fortunately, this is easy to do. The table's entries refer to code segment locations. They must have entries in the object file's relocation section. On disassembling an indexed jump, the disassembler checks whether the next location has a relocation entry. If so, it must be a jump table entry, since the first word of a valid instruction cannot be relocated (only the second and succeeding words). A similar argument holds for succeeding 16-bit or 32-bit entry of the table, and thus by checking the relocation section at increasing addresses until no relocation entry is found, the disassembler can determine the extent of the table. (It saves it in a special structure for use by the flow graph construction phase.) To note that a table is present, an indexed jump that has one is given the special internal Opcode `jsw`. If this technique could not be used, a compiler could be changed to annotate the code, via the symbol table, to mark the location and extent of the table. Another method, used by some C compilers, is to follow the indexed jump by an illegal instruction, which alerts the disassembler to the presence of the table. Johnson[6] discusses another solution to similar problems in a more general setting, but requiring an addition to the object file format.

4.2. Basic Blocks and Flow Graph

In the next phase, the analyser finds the basic blocks in the object file, and builds a flow graph. It makes two passes over the code.

During the first pass, it builds a list of addresses. It finds each instruction that changes the flow of control (eg, branches, calls, and returns), and appends the address of the following instruction to the list. If the instruction is a branch or a call it also appends the target address to the list. The list is then sorted. It contains the addresses that are potential targets of the flow of control: potential basic block headers.

Each basic block in the code is represented by a value of the record type *Block*. Each Block contains its starting address, a list of Instructions, and a reference count. The last Instruction is always one that alters the flow of control. One Block is allocated for each unique address in the list of addresses just built. The reference count of a given Block is set equal to the number of times its address appears in the list, representing the number of ways control could flow to that Block. A map is built that yields the Block, if there is one, for a given address. The Blocks are also linked together in address order.

The analyser uses these structures to build the flow graph during a final pass over the object code. The analyser has a current Block, initially the first on the Block list. As it reads Instructions, it adds them to the current Block until it has added a control-flow Instruction. Each Block has a list of predecessors and a list of successors. If the last instruction of a block is a branch or call to a known location, the corresponding destination Block is obtained from the map, and it is made a successor of the current Block. The current Block is added to the destination Block's list of predecessors. The current Block now contains a maximal sequence of sequential instructions ending with a potential change of the flow of control. That is the definition of a 'basic block'; the current Block is therefore complete. The analyser advances to the next Block on the list.

When a branch or call instruction is indirect, the target address can only be known at run-time, preventing the construction of a complete flow graph. This can happen in just two ways in the code generated by the York compiler: indirect calls are used in the implementation of elaboration checks, and the code for an Ada `case` statement might select the appropriate alternative using an indexed jump. Let us consider each case in turn.

The specification of a subprogram can be elaborated separately from its body. Between the two events, it is erroneous to call the subprogram: at the point of call the system must raise the exception `PROGRAM_ERROR` instead. The York Ada compiler does the check using indirect calls, as follows. Each subprogram for which the error is possible has an associated pointer variable. Elaboration of the subprogram's specification initialises the pointer variable to point to a run-time routine that raises `PROGRAM_ERROR`. Elaboration of the subprogram's body sets the pointer variable to point to the code for the body (the real entry point). All potentially erroneous calls to the subprogram cause the compiler to generate a call indirect through the pointer variable. The elaboration check is thus accomplished at the cost of an extra memory access per call (assuming that global optimisation cannot preload the pointer variable in a given scope).

The simplest solution, adopted here, is to suppress elaboration checks. This restriction can be eliminated. As described in §4.3, the compiler plants special markers in the object file's symbol table at the point of each call to an Ada subprogram or entry. When it finds an indirect jump, the flow-graph builder could check for such an entry. If one is found, the block corresponding to the body of the called entity would be made a successor of the calling block. Otherwise, the analyser would complain.

An Ada `case` statement can be implemented by an indexed jump. The value of the `case` expression indexes a table of pointers to the code corresponding to each `when` clause. All that is required when building the flow graph is to find the table, and make the each block of code to which its entries point a potential target of the indexed jump (and thus a potential successor of the block containing the jump). As noted earlier, the disassembler already addresses this problem for different reasons; the flow graph code simply checks for a `jsw` Opcode and examines its associated table, saved by the disassembler.

4.3. Ada Structure Graph

The Ada code generator annotates its output using assembler directives that the assembler copies to the object file's symbol table. The symbol table already marked the beginning and end of Ada subprograms and blocks, using the UNIX `N_LBRAC` and `N_RBRAC` debugging records,† but without distinguishing different types of blocks. Another record

†They correspond to Left BRACe and Right BRACe in C.

type, N_PC, provided by SunOS for use by the Pascal compiler, was conscripted for use as a block type marker. The Ada compiler was changed to generate a N_PC debugging record after each N_LBRAC record; one of the values in the N_PC record tells the type of nested block. The following types are used:

Value	Internal Name	Block Type
0	Bproc	subprogram body
1	Btask	task body
2	Bblock	`begin` block
3	Bswitch	`case` statement
4	Bcase	`when` clause
5	Bif	`if` statement
6	Belif	`elif` clause
7	Bthen	`then` clause
8	Belse	`else` clause
9	Bloop	`loop` statement
10	Bloopbody	`loop` body (ie, excluding iteration control)
11	Bpkg	`package` body
64	Bcall	subprogram call
65	Becall	entry call
66	Bpcall	protected entry call

The record marking the start of a loop also includes the value given by the LOOPCOUNT pragma.

Each record includes a reference into the program's instructions, to the point where the specified Ada construction begins or ends. *Wet0* reads these records from the symbol table, and builds a tree of 'Ada block' descriptors, which have the following form:

```
type Block_type is (Subprogram, Task_body, Begin_block, ...);

type Adablock_reference is access Adablock;

type Adablock(type: Block_type) is record
      start, end: Code_location;
      inner_block: Adablock_reference;
      next_on_this_level: Adablock_reference;
      case type in
      when Loop =>
            loop_count: integer;
      when Call =>
            entry_name: string;
      when others =>
            null;
      end case;
end record;
```

Each node of the tree holds the type of the Ada construction, its starting and ending instruction addresses, and a reference to the Ada block descriptor of the first construction nested inside it (if there is one). If several blocks are nested inside another, as for instance the several `when` clauses of a `case` statement, their Ada block descriptors are linked together.

4.4. Instruction Timing Data

The analyser represents timing information as instances of the record type *Cycles*, shown below:

```
type Cycles is record
      tot: Cycle_count;  -- total number of clocks
      rd: Cycle_count;   -- number of read cycles
      ia: Cycle_count;   -- number of fetch/prefetch cycles
      wr: Cycle_count;   -- number of write cycles
end record;

function "+"(x, y: Cycles) return Cycles;
function max(x, y: Cycles) return Cycles;
```

Functions are provided to accumulate Cycles, and take the largest of two Cycles values.

The 68020 processor handbook gives several possible times for each addressing mode and instruction, often in particular combinations. It gives best-case, cache-case, and worst-case times, which are represented in the analyser by the following structure:

```
type Cycleset is record
      best, cache, worst: Cycles;
end record;
```

A group of machine-dependent functions in the analyser takes an Instruction, selects the appropriate timing tables for its operand Addresses, and returns the correct Cycleset value. The timing tables are just those printed in the 68020 processor handbook. The outer displacement addressing modes and some instructions not found in compiled code are excluded. This reduces the size of the tables somewhat.

When several table entries might apply to a given Instruction, the one with the largest (worst) times is used. For instance, branches are assumed always to be taken. There is one exception to this rule. On the 68020, the Ada compiler generates a `trapv` instruction following most arithmetic instructions it generates, since the machine does not trap automatically on arithmetic overflow. It is assumed that overflow does not occur, and that `trapv` therefore never traps. (Note that the exception `CONSTRAINT_ERROR` will still be raised at run-time if this assumption is wrong.)

The analyser sums the worst-case time of each Instruction in each basic Block, to obtain the worst-case time for that Block.

4.5. Correlating Basic Blocks and Ada Blocks

Having completed the previous stages, *wet0* has the (timed) basic Blocks at the object code level linked in a flow graph, and another graph structure describing the Ada program's structure by Adablocks. It now correlates the two: associating groups of basic blocks with the Ada constructions that caused the compiler to generate them. It recursively traverses the Adablock tree (following each *inner_block* reference), associating each Block of the object file with the Adablock that most closely contains it. During a second recursive traversal of the Adablock graph, a summary of the correlated graphs is written to an intermediate file. In the file, a textual representation of each Adablock structure is followed by a textual summary of each Block it contains, including its worst-case execution times. (The file also includes symbolic references to the Ada source file, by source file name and line number.)

4.6. Output Files

Two output files are produced by this analysis phase for a given input object file, *file*`.o`:

file`.B` contains a listing of the program's basic blocks and basic flow graph data

file`.W` contains a summary of the program's timing data (this is an intermediate file, for internal use only)

The file formats are given in Appendices A and B.

4.7. Program Size

The sizes in lines of source code of the main components of *wcet0* are shown in the table below:

Lines	Purpose
174	(data structure declarations)
116	(internal names for 68020 operations)
620	read words from object file; read symbol table; read relocation
297	extract addressing mode; Address constructors; print Address
72	Instruction constructors
461	read one MC68020 instruction and build Instruction
411	build flow graph of basic Blocks
112	find the target of a jump, branch, or call
219	look up an Instruction's time in tables
238	look up an effective Address's time in tables
377	interpret Ada compiler's annotations
166	correlate annotations and Blocks
79	generic List package
42	timing tables for 68020 `move` instruction
164	timing tables for other 68020 instructions

5. SEGMENTS AND SEGMENT TREES

The analysis program *wet0* yields the worst-case times of each basic block, and a summary that associates each basic block with the Ada construction that produced it. The Schedulability Analyser needs a more concise summary of the timing properties of each unit. A second program *segs* takes the output of the first stage of analysis and reduces it to a form from which Schedulability input can be derived easily. In particular, *segs* resolves references to internal procedure calls and known library subprograms, reduces to a singleton the set of times corresponding to the alternatives of `case` and `if` statements, and finds the worst-case time for the execution of loops.

A *segment tree* is a tree of basic blocks, as defined by Stoyenko, Hamacher, and Holt[2]. Their Real-Time Euclid compiler builds a segment tree for each unit during code generation. Here, it is not built by the Ada compiler, but by the separate program *segs*. The set of *segment* types is an extended subset of that used by the Euclid compiler. The various segment types are shown in Table 1. Compared to Stoyenko et. al, several types peculiar to the Euclid implementation have been dropped, and a loop segment type has been added.

Type	Interpretation
simple segment	This corresponds to a basic block. It contains the total worst-case time for all instructions in the block.
internal call segment	The segment contains a reference to the called routine.
external call segment	The segment contains the name of the called external routine.
selector segment	Such a segment represents a multi-way selection, including `if` and `case` statements. A selector segment has one subtree for each alternative of the Ada construction: the subtree corresponds to the segment tree for the statement or statements of that alternative.
loop segment	Each such segment corresponds to a loop in the program. The loop segment has one subtree: the segment tree corresponding to the loop body. A loop segment also holds a count: the maximum number of times the loop will be executed.

Table 1. Segment Types

Segs proceeds in two phases. First, it reads the summary file (the `.W` file) produced by *wet0*, and builds the corresponding segment tree. A separate segment tree is built for each body of code described in the file, corresponding to a library unit's body, or to the body of a procedure, function, or task nested inside another. Once all trees have been built, each external call segment is checked. If it mentions the name of a unit whose segment tree has been built, it is replaced by an internal call segment referring to that tree. If the external call segment refers to a known support routine, the segment is replaced by a simple segment that gives the routine's worst-case execution time.

In the second phase, *segs* tries to reduce the segment tree to a single node, using an iterative application of a recursive algorithm similar to that described in sections III.A and III.B(1) of Stoyenko *et al*[2]. Starting at the root of each segment tree, the algorithm works down the tree, depth-first:

- A simple segment cannot be reduced further.
- If an internal call segment refers to a segment tree that is a simple segment, replace the call segment by that simple segment.
- Replace adjacent simple segments by a single simple segment that contains the sum of their worst-case times.
- Replace all alternatives of a selector segment that are simple segments by one simple segment that contains the largest worst-case time of any alternative.
- Replace any selector segment that has just one descendent that is a simple segment by that simple segment.
- If a loop contains only a simple segment, replace the loop by a simple segment whose worst-case time is the product of the loop's count and the worst-case time of the segment representing the body.

The algorithm is applied to the segment tree for each unit until no further reductions can be made.

Exception handlers for a block are represented in the structure graph as a multi-way selection. The corresponding selector segment is resolved as above, yielding the worst-case time for a handler equal to its most expensive alternative (the most expensive exception to handle). Note that the code in an exception handler will typically refer to some run-time support routines, and thus an exception handling block can only be completely resolved if times are provided for all such routines. If not, the output from the analyser will mark the entity containing the handler as 'unresolved'.

The `select` statement is represented as a sequence of segments evaluating the guards, interspersed with call segments to run-time routines that 'post' the results, then a call segment referring to the the support routine `and_eselect`, the `accept` bodies). The corresponding segment tree can be resolved by the algorithm above.

As soon as none of the rules above can be applied, *segs* writes the results to its standard output. There are two output formats. By default, *segs* writes a printable representation of each segment tree, labelled with the name of the corresponding Ada entity. Normally, this output is directed to the file *file*`.S`; it is documented in Appendix C.

If the `-s` option is given, *segs* instead writes a set of summary lines to the standard output. Each segment tree has one line, telling whether or not the tree was successfully resolved. This output format is described in Appendix D. It is suitable for conversion to the input format for the Schedulability Analyser.

6. PROBLEMS PECULIAR TO THIS IMPLEMENTATION

Working with the York Ada compiler posed a few problems that might or might not appear in another implementation (ie, they are not inherent to timing analysis).

6.1. Name Mapping

Calls to subprograms in library packages

The original UNIX object file format on the PDP-11 and VAX systems restricted the length of symbols in the object file symbol table. This restriction was removed by one of the UNIX implementations from the University of California (Berkeley); years later, the Berkeley changes were adopted by AT&T in System V. Meanwhile, the York Ada compiler had to cope with the restriction: it mapped long Ada names to shorter linker names. The compiler still does this (although a restriction it originally imposed on the length of library unit names was removed long ago).

Basically, the names appearing in a package are numbered, starting at 1; for each name, a textual representation of this number is concatenated to the unit name to form an external name for use in object files. The external name for something declared in a package specification has `S_` before the unit name and index number, and finally all external names start with `_` if that convention is used by the UNIX implementation for external names in C. In the object code, calls to a procedure in a library package, or references to a variable in a library package all use the appropriate external name.

For example, given the package specification below:

```
package SOMETHING is
      procedure LONG_COMPLEX_NAME;
      procedure another_long_name;
      pithy_name: integer;
end;
```

the compiler would use the following mapping on the Sun 3:

Ada name	*External name*
`SOMETHING.LONG_COMPLEX_NAME`	`_S_something01`
`something.another_long_name`	`_S_something02`
`something.pithy_name`	`_S_something03`

In practice, when calculating the numbering one must allow for hidden variables added by the compiler, for instance to hold the values of non-static bounds of ranges, or hidden pointer values that implement elaboration checks. This makes it harder to guess the link name of an object by examining the source.

How does this affect the timing analyser? The object file's symbol table contains only the external name. The disassembler can convert the address of the target of a direct call or jump instruction into an external name by looking in the symbol table for names with that address. It cannot deduce the corresponding Ada name from that external name without some help from the compiler.

Entry calls

Calls to task entries, including the entries of a protected task, are even harder for the disassembler to resolve, since they are done indirectly. Each entry is assigned an index number within its task, and each entry call is replaced at object code level by a call to a run-time support routine, with the entry's index and a reference to the appropriate task as parameters. For example, for each protected entry call the disassembler will find a call to the run-time routine `_protenter`. The index number of the corresponding entry is buried in the parameter evaluation code preceding the call, and will be hard (or impossible) to find in general.

Reversing the name mapping

A simple change to the compiler allows both cases to be handled easily. The compiler inserts a symbol table record immediately before each call it generates. The symbol table record gives the full name of the Ada subprogram or entry being invoked, and the type of call (eg, subprogram, entry, or protected entry). The disassembler first finds the external name that is the target of a call, by looking up the target address in the symbol table; it then looks up the address of the call instruction in the symbol table to see if there is a corresponding record produced by the compiler. If so, the disassembler includes the Ada name contained in that record in its summary output for the segment analyser.

6.2. Code Generated Out of Order

The code generated for an Ada procedure has a prologue, which allocates a stack frame and builds a save area, the code for the procedure body, and an epilogue, including the code to return to a caller. On the 68020, the prologue code is produced in two parts. One part appears in the obvious place before the body's code, The other is generated *after* the procedure's body. (The code in that part of the prologue depends on the code generated for the body.)

This ordering does not affect segment resolution. A body is either resolved or unresolved. If it is not resolved, no worst-case time can be given for it. If the body is resolved, the execution time for the two parts of its prologue must have been added together, and since the addition of execution times is commutative, it does not matter that the two parts of the prologue were initially in separate segments of the segment tree for the body.

The contents of the `.S` intermediate file for an *unresolved* entity must be interpreted with this in mind. In the representation of the segment tree that appears there, the second prologue segment will follow the segment subtree for the entity's body.

This peculiarity could be removed in two ways. The Ada compiler could be changed to keep its generated code in order. A more general method is to make use of the flow graph built by *wet0*. Before starting to correlate the basic blocks and the Ada structure, *wet0* could traverse the flow graph and apply a simple code motion algorithm (moving a block that is the target of one unconditional jump to follow the block containing that jump). The corresponding segments would then be generated in the right order in the segment tree.

7. EXAMPLE

In this example, the analyser's phases will be applied to the simple subprogram body below, to show how a segment tree is collapsed (resolved) to a single segment. The start of each logical block in the Ada source is marked:

```
procedure F(i: integer) is     •b0
      j: integer := 0;
begin
      if i > 10 then •b1
            j := j+5;
      else  •b2
            j := j-5;
      end if;
      •b3
end;
```

Here is a simplified representation of the segment tree produced by *segs* on reading the summary `.W` file produced by the disassembler:

```
 1     unresolved: body f eg1.A:10x0eg1.A:10x0
 2  basic 1 91 0.00455 - 2 1 3 6    •b0
 3  oneof 1
 4    alt 1
 5      basic 1 37 0.00185 - 4 13 4 13     •b1
 6    alt 1
 7      basic 1 22 0.0011 - 5 20 5 20      •b2
 8  basic 1 79 0.00395 - 6 26 8 36  •b3
 9  excall 1 0 0 _domemf 8 36 8 36
10  basic 1 129 0.00645 - 9 40 10 49
```

Considering only the values of interest here, the general form of each segment type is:

`alt` *loopcount*
`basic` *loopcount worst_case_clockticks*
`excall` *loopcount external_name*[`,` *Ada_name*]
`oneof` *loopcount*

Only these values will be included when showing transformations of the segment tree.

A `oneof` segment is a selector segment: the root of a tree of alternatives, each rooted at an `alt` segment. An `excall` segment represents an external call.

Note that all the code from the start of the procedure to the conditional jump of the `if` statement is included in the basic segment on line 2. (Line 9 is part of the procedure's prologue, which appears after the body for reasons given earlier.)

Having built this tree, *segs* proceeds to resolve it: it attempts to reduce the tree to a single segment. The `oneof` tree corresponding to the Ada `if` statement has two alternatives, both simple basic segments. *Segs* replaces the whole `oneof` tree (lines 3 to 7) by the most expensive alternative:

```
 2  basic 1 91
 5      basic 1 37
 8  basic 1 79
 9  excall 1 _domemf
10  basic 1 129
```

Three simple basic segments are now adjacent (lines 2, 5, and 8), and can be combined:

```
 2′ basic 1 207
 9  excall 1 _domemf
10  basic 1 129
```

Segs looks up the external routine `_domemf` in a table of 'known execution times'. This routine is a run-time support routine that can only be called when the run-time stack has overflowed (it raises the exception `storage_error`).

Assume we have decided this will be a fatal error, and thus for timing purposes it will not be executed, and can simply declare its execution time to be zero. (The actual value used, or indeed whether a value will be provided at all, is entirely under control of the analyser's user.)

```
 2′ basic 1 207
 9′ basic 1 0
10  basic 1 129
```

Now three basic segments are adjacent, and can be collapsed into the single segment below:

```
basic 1 336
```

The summary output from *segs* would contain the line:

```
f eg1.A:1@0x0  #2     1      resolved      0.0168
```

(The value of 336 clock cycles has been converted to milliseconds for a 20 MHz clock.)

REFERENCES

1. A.Burns and A.J.Wellings, ‘‘Timing Analysis Tool’’, pp. 7-15 in *Task 4 Deliverable on ESTEC Contract 9198/90/NL/SF*, Department of Computer Science, University of York (September 1991).
2. A.D.Stoyenko, V.C.Hamacher and R.C.Holt, ‘‘Analyzing Hard-Real-Time Programs for Guaranteed Schedulability’’, *IEEE Transactions on Software Engineering* **17**(8) (August 1991).
3. A.V.Aho, R.Sethi and J.D.Ullman, *Compilers: Principles, Techniques, Tools*, Addison Wesley, Reading, Mass. (1986).
4. B.S.Baker, ‘‘An Algorithm for Structuring Flowgraphs’’, *Journal of the ACM* **24**(1), pp. 98-120 (January 1977).
5. Sun Microsystems, in *UNIX Interface Reference Manual, SunOS 4.0*, Mountain View, California (May 1988).
6. S.C.Johnson, ‘‘Postloading for Fun and Profit’’, pp. 325-330 in *Proceedings of the Winter USENIX Conference, Washington D.C.*, USENIX Association (January 1990).

APPENDIX A – FORMAT OF THE .B INTERMEDIATE FILE

The file contains a section for each basic block of object code, of the following form:

address `Block` *bno*`,` `fun` *fno* *source_file*`:`*sline@address* `ref` *nref*`,` `pred` *bno* ...
instructionbest_time`,` *cache_time*`,` *worst_time*
...
`------` `flags` *flags* `successor(s):` *bno* ... `[` `call:`*link_name* `]`
`------` `total:` *best_time*`,` *cache_time*`,` *worst_time*

For the 68020, the timing for an instruction is represented in the form below:

total_cycles `(`*read_cycles*`/`*prefetch_cycles*`/`*write_cycles*`)`

Following the conventions of the processor handbook, the cycle counts assume zero wait states, and the read, prefetch, and write cycle counts are included in the total. When the number of wait states is non-zero, *wet0* adds the appropriate number of extra read and write cycles to the total to form the value passed to later analysis programs.

The other fields are as follows:

address
: The address in the object file of the basic block, in hexadecimal (hence a leading `0x` as in C).

`Block` *bno*
: Each block in the file is numbered, starting at 1. Block 1 is a pseudo-block used internally, and does not appear in the output file.

`fun` *fno*
: Each function (body of something, in Ada terms) is also numbered, starting at 0. A function is a maximal sequence of blocks between *return* instructions.

source_file:*sline@address*
: A reference to the source file and approximate line number of the Ada source that produced this object code. The *address* is the object code address in the line number's symbol table record. It often differs slightly from the block's address.

`ref` *nref*
: The number of references to this block from other block; there is an implicit reference to a block corresponding to an exported symbol.

`pred` *bno* ...
: The current block can be reached from the blocks listed; the list may be empty.

  Note *the list will not include an implicit reference by the textually preceding block if that block ends with an instruction that allows control to flow into the current block (perhaps conditionally).*

`flags` *flags*
: The flags field contains a printable form of some internal flags giving the nature of the last instruction in the block (unconditional branch, return instruction, conditional branch, etc.).

`successor(s):` *bno ...*
: On leaving this block, control can flow to any of the blocks listed; the list can be empty.

`call:` *link_name*
: If the block ends with a call to an external location found in the symbol table, that *link_name* appears here.

`total:` ...
: The sum of the best-, cache-, and worst-case times of all instructions in this block.

An empty block with address `MAXINT` is added to the end of each file; it becomes the target of all jumps that cannot be resolved within the current file, including calls to external routines and indirect jumps.

For example, consider the following subprogram:

```
procedure simple(i: in out integer) is
begin
      if i > 10 then
            i := i - 2;
      else
            i := i - 20;
      end if;
end;
```

The corresponding `.B` file is shown below, without the final empty block:

```
0 Block 2, fun 0 p.A:1@0x0 ref 1, pred
     link &0x0,%fp  3 (0/0/1), 5 (0/0/1), 7 (0/1/1)
     jmp  46(%pc)   2 (0/0/0), 8 (0/0/0), 11 (0/2/0)
     ------ flags UBR  successor(s): 7
     ------ total:  5 (0/0/1), 13 (0/0/1), 18 (0/3/1)
0x8 Block 3, fun 0 p.A:1@0x0 ref 1, pred 9
     move %fp,-16(%fp)   3 (0/0/1), 5 (0/0/1), 7 (0/1/1)
     move 8(%fp),%a0     3 (1/0/0), 7 (1/0/0), 9 (1/2/0)
     cmpi &0xa,(%a0)     3 (1/0/0), 6 (1/0/0), 10 (1/2/0)
     bcc  &0xe 3 (0/0/0), 6 (0/0/0), 9 (0/2/0)
     ------ flags CBR  successor(s): 5 4
     ------ total:  12 (2/0/1), 24 (2/0/1), 35 (2/7/1)
0x18 Block 4, fun 0 p.A:4@0x18 ref 1, pred
     move 8(%fp),%a0     3 (1/0/0), 7 (1/0/0), 9 (1/2/0)
     moveq     &0xfffffffe,%d1     0 (0/0/0), 2 (0/0/0), 3 (0/1/0)
     add  %d1,(%a0) 3 (0/0/1), 4 (0/0/1), 6 (0/1/1)
     trapv          1 (0/0/0), 4 (0/0/0), 5 (0/1/0)
     jmp  12(%pc)   2 (0/0/0), 8 (0/0/0), 11 (0/2/0)
     ------ flags UBR  successor(s): 6
     ------ total:  9 (1/0/1), 25 (1/0/1), 34 (1/7/1)
```

```
0x26 Block 5, fun 0 p.A:6@0x26 ref 1, pred 3
     move 8(%fp),%a0     3 (1/0/0), 7 (1/0/0), 9 (1/2/0)
     moveq     &0xffffffec,%d1     0 (0/0/0), 2 (0/0/0), 3 (0/1/0)
     add  %d1,(%a0) 3 (0/0/1), 4 (0/0/1), 6 (0/1/1)
     trapv          1 (0/0/0), 4 (0/0/0), 5 (0/1/0)
     ------ flags -  successor(s): 6
     ------ total:  7 (1/0/1), 17 (1/0/1), 23 (1/5/1)
0x30 Block 6, fun 0 p.A:8@0x30 ref 1, pred 4
     jmp  32(%pc)   2 (0/0/0), 8 (0/0/0), 11 (0/2/0)
     ------ flags UBR  successor(s): 10
     ------ total:  2 (0/0/0), 8 (0/0/0), 11 (0/2/0)
0x34 Block 7, fun 0 p.A:8@0x30 ref 1, pred 2
     lea  -16(%sp),%a0   5 (1/0/0), 7 (1/0/0), 9 (1/2/0)
     cmpa 0x0.l,%a0 4 (1/0/0), 8 (1/0/0), 11 (1/2/0)
     bcc  &0x6 3 (0/0/0), 6 (0/0/0), 9 (0/2/0)
     ------ flags CBR  successor(s): 9 8
     ------ total:  12 (2/0/0), 21 (2/0/0), 29 (2/6/0)
0x40 Block 8, fun 0 p.A:8@0x30 ref 1, pred
     jsr  0x0.l     3 (0/0/1), 7 (0/0/1), 13 (0/2/1)
     ------ flags CALL  successor(s): 11 9 call:_domemf
     ------ total:  3 (0/0/1), 7 (0/0/1), 13 (0/2/1)
0x46 Block 9, fun 0 p.A:8@0x30 ref 2, pred 7
     clr  -(%sp)    6 (1/0/1), 9 (1/0/1), 11 (1/1/1)
     clr  -(%sp)    6 (1/0/1), 9 (1/0/1), 11 (1/1/1)
     clr  -(%sp)    6 (1/0/1), 9 (1/0/1), 11 (1/1/1)
     clr  -(%sp)    6 (1/0/1), 9 (1/0/1), 11 (1/1/1)
     jmp  -72(%pc)  2 (0/0/0), 8 (0/0/0), 11 (0/2/0)
     ------ flags UBR  successor(s): 3
     ------ total:  26 (4/0/4), 44 (4/0/4), 55 (4/6/4)
0x52 Block 10, fun 0 p.A:8@0x30 ref 1, pred 6
     unlk %fp  5 (1/0/0), 6 (1/0/0), 7 (1/1/0)
     rts       9 (1/0/0), 10 (1/0/0), 12 (1/2/0)
     ------ flags UBR,RET  successor(s):
     ------ total:  14 (2/0/0), 16 (2/0/0), 19 (2/3/0)
```

The successor fields reveal the flow-of-control, which can be represented by the following graph:

When generating code for a procedure on the 68020, the York Ada compiler generates the procedure's initialisation code in two sections, before and after the code for the body, connected by unconditional jumps. Thus, the first block 2 jumps to block 7 to complete the initialisation, and block 9 jumps back to block 3, to execute the code for the procedure body.

Blocks 7 and 8 check for overflow of the run-time stack (the external procedure `_domemf` called by block 8 raises `storage_error`). A call instruction can potentially return, and so block 9 is listed as a successor to block 8, although `_domemf`, if called, does not return.

Block 3 evaluates the Boolean expression of the `if` statement; blocks 4 and 5 are its `then` and `else` clauses. Block 10 returns from the procedure.

APPENDIX B – FORMAT OF THE .W INTERMEDIATE FILE

This file is a sequence of records, one per line. There are two record formats: one summarises a basic block in the object code, the other characterises an Ada construction that typically includes one or more basic blocks. Each record includes a nesting level, which helps *segs* to determine the basic blocks to which an Ada record applies. The level increases (decreases) by one at the start (end) of each body, loop, or clause of a selection statement.

Each record might contain one or two *location* fields. These all have the same format:

source_file:*source_line*@*address*

where the *source_file* and *source_line* refer to the Ada source that produced the code, and the *address* is the code's starting (or ending) address.

The Ada structural records have 6 fields:

nesting block_type loop_count name start_location end_location

The fields are interpreted as follows:

nesting
: Nesting level of code described by this record.

block_type
: A printable representation of the internal Ada block type; see below.

loop_count
: If the construction appears in, or begins a loop, the *loop_count* gives the LOOP-COUNT given by the programmer or determined by the compiler.

name
: If the *block_type* indicates a call to a subprogram or entry, this field contains the full Ada name of the called entity; if the *block_type* marks the start of a body of code, this field gives the code's full Ada name, if known; otherwise, the field contains the space-filler, `-`.

start_location
: Source and object code reference to the start of the construction.

end_location
: Source and object code reference to the end of the construction.

The following block types are currently used:

`block`	range of `declare` block
`call`	location of Ada subprogram call
`case`	range of `case` statement
`ecall`	location of entry call
`elif`	range of `elif` clause

`else`	range of `else` clause
`except`	range of exception handler
`file`	first record of `.W` file
`gproc`	start of object code for subprogram/task/package body
`if`	range of `if` statement
`loop`	range of `loop` statement
`loopbody`	range of loop's body
`pcall`	location of protected entry call
`pkg`	range of package body
`proc`	range of subprogram body
`task`	range of task body
`then`	range of `then` clause

Although any of these can appear in a `.W` file, not all are used by *segs*. For instance, *segs* takes the `gproc` record to mark the start and end of bodies of code in the file, and ignores the `proc`, `task` and `pkg` records, since the distinctions between subprogram, task, and package bodies are not relevant to it. Furthermore, the latter record types include just the code for the corresponding Ada bodies, but *segs* must include any initialisation code, which is covered by the `gproc` record.

Each basic block of code (ie, each section in the `.B` file) has one summary record, containing the following 13 fields:

nesting `bb` block function start_pc end_pc loop_count best cache worst called_name source_location line

The contents of fields not previously explained are as follows:

block
function
: Block and function numbers of this basic block (cf. the `.B` file).

start_pc
end_pc
: Starting and ending locations of the code of this basic block, relative to the start of the object file. This interval is half-open (the *end_pc* is not included in the block).

best, cache, worst
: Total cycle time for this block in each case (best, worst, cache), including any wait states.

called_name
: If the basic block does not end with a call instruction, the field contains the space-filler `-`. Otherwise, this field contains either the link name of the called code, or `#` followed by the basic block number of the called code (in the current file). If the called routine corresponds to some Ada entity (eg, subprogram or entry), the link name reference is followed by `,`*Ada_name*`,`*call_type* . The *Ada_name* is the fully qualified Ada name of the called entity; the *call_type* is `call`, `entry`, or `pentry` (protected entry).

line
: The line number in the `.B` file of the section corresponding to this basic block.

For example, take the subprogram that produced the `.B` file in Appendix A:

```
procedure simple(i: in out integer) is
begin
      if i > 10 then
            i := i - 2;
      else
            i := i - 20;
      end if;
end;
```

The corresponding `.W` file is shown below:

```
1 file 1 - ?:0@0x0 ?:0@0x7fffffff
2 gproc 1 simple p.A:1@0x0 p.A:1@0x0
2 bb 2 0 0x0 0x8 1 7 15 26 - p.A:1@0x0 1
2 bb 3 0 0x8 0x18 1 18 30 55 - p.A:1@0x0 6
2 proc 1 simple p.A:1@0xc p.A:7@0x30
3 if 1 - p.A:3@0xc p.A:7@0x30
4 then 1 - p.A:3@0x18 p.A:4@0x22
4 bb 4 0 0x18 0x26 1 13 29 52 - p.A:4@0x18 13
4 else 1 - p.A:4@0x26 p.A:6@0x30
4 bb 5 0 0x26 0x30 1 11 21 37 - p.A:6@0x26 21
1 bb 6 0 0x30 0x34 1 2 8 15 - p.A:8@0x30 28
1 bb 7 0 0x34 0x40 1 16 25 45 - p.A:8@0x30 32
1 bb 8 0 0x40 0x46 1 5 9 19 _domemf p.A:8@0x30 38
1 bb 9 0 0x46 0x52 1 42 60 83 - p.A:8@0x30 42
1 bb 10 0 0x52 0x7fffffff 1 18 20 29 - p.A:8@0x30 50
```

APPENDIX C – FORMAT OF THE .S INTERMEDIATE FILE

The `.S` file contains printed representations of segment trees, each labelled by the name of its corresponding Ada entity. Each segment of a tree takes one line; when the segment is a root of a subtree, the subtree is printed on following lines, indented.

The general form of a segment line is:

type loop_count [cycles millisec called [sb sl eb el]]

The *loop_count* is the appropriate LOOPCOUNT value for segments representing code in the body of a loop; segments not in a loop, and segments in an infinite loop (cycle) have a *loop_count* of 1.

The segment types correspond to those listed in §5 (‘‘Segments and Segment Trees’’) of the Implementation Guide to *wcet*, except that ‘selector segments’ are represented internally by two segment types, `alt` and `oneof`. The types that can appear in the `.S` file are shown below in the following table:

Printed Name	**Interpretation**
`basic`	simple segment (basic block or resolved compound)
`call`	internal call (call to another segment tree in this file)
`excall`	external call (unresolved external call)
`loop`	loop over segment appearing as subtree
`alt`	root of segment tree for one alternative of selector segment
`oneof`	selector segment: optional `basic` segment, then sequence of `alt`

The optional `basic` segment of a logical selector segment corresponds to the basic block(s) of the code that makes the selection. (The Ada compiler can drop the ‘selector’ marker either before or after this code; this determines whether a `oneof` segment has the `basic` subtree or not.)

The format of segment lines by type is shown below:

`alt` loop_count
`basic` loop_count cycles millisec `-`