HARD REAL-TIME OPERATING SYSTEM KERNEL STUDY

TASK 8, Volume D

USING THE WORST-CASE EXECUTION ANALYSER

Charles Forsyth
York Software Engineering Limited, UK
17th May 1993

Abstract

This document describes the use of programs that constitute 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

TABLE OF CONTENTS

1. ADA RULES

To allow the timing analyser to estimate the time spent in sections of Ada code, some restrictions have been placed on the use of Ada. You might also need to tell the analyser, via a new pragma, the maximum number of times a `loop` will loop in any execution.

1.1. Use of the New Pragma LOOPCOUNT

A loop in Ada will have one of the following forms:

1. a cycle (infinite loop) as might be used in the body of a task providing a service:

```
loop
      L;    -- do something ...
end loop;
```

2. a `for` loop with static bounds:

```
for i in 1 .. 10 loop
      L;
end loop;
```

3. a `while` loop, a `for` loop with non-static bounds, or a `loop` containing an `exit` statement:

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

pragma loopcount(250);
loop
      exit when C;
      L;
end loop;
```

In cases (1) and (2), the compiler can deduce the maximum time requirements of the loop without help: the first is an infinite cycle; the second will execute the loop body L at most 10 times (an exception might cause the loop to stop prematurely). For loops of the third type, you must provide a loop limit, using a new pragma:

`pragma LOOPCOUNT(`*static_universal_integer*`);`

The pragma must appear immediately before the loop it limits, as shown above. (If your estimate is exceeded, the current task will exhaust its budgeted time, and an exception will be raised.) In the example, the analyser can for timing purposes allow for 1001 executions of `f(j)` and 1000 executions of the loop body L.

The compiler issues a warning when it encounters an unbounded loop without a preceding `pragma LOOPCOUNT`.

1.2. Restrictions on Ada

When writing an Ada program that will be analysed, you must follow the rules below:

1. Do not use `goto` statements (checked by the compiler).
2. Do not use recursion, including mutual recursion (the timing analyser complains if a program contains a nested sequence of calls more than 50 calls deep; this also detects recursive calls).
3. Do not declare or allocate objects of dynamic size (checked by the compiler).
4. Do not use the optimisation option `-O` of the *Ac* command (for any unit that is to be given to the analyser – note that no check is made).
5. Do not overload subprogram and entry names *in the same package or task* (one name can be used in many packages or tasks but not twice in the same package). This rule does not apply to the pre-supplied library packages.
6. Suppress elaboration checks (they are suppressed by default by the ESA version of *Ac*).
7. Tasks should communicate via entries of protected tasks, not the conventional Ada rendezvous (the analyser will leave normal Ada entry calls unresolved).
8. The only subunits that are guaranteed to be analysable are subprograms that are subunits of a library package. Package subunits must not be used.

The bodies of library units *must* be kept in separate source files from the corresponding specifications. No more than one body may appear in a given source file.

The use of generics and entry families might prevent the analyser from resolving references between packages, especially when objects are renamed; thus generics and entry families are best avoided. (Basically, the problem is that through these features Ada allows several names to refer to the same object, and internally the York compiler does not maintain a canonical name, since that has only been needed for this application!)

2. WCET – A COMMAND TO FIND THE WORST-CASE EXECUTION TIME

A single command script oversees the running of the various programs that do the timing analysis. The command is used as follows:

`wcet` [`-w` *wait_states*] [`-c` *clock_MHz*] [`-m` *prefetch_cycles*] [`-i`] *name* ...

The command defaults to 2 wait states, a 20 MHz clock and 1 prefetch cycle for literal moves.

The `–m` option is included as a problem was discovered with the timings returned by WCET for literal assignment operations, such as:

```
a := 10;
```

The worst-case times returned in such cases were shorter than the actual times as measured on the target hardware. The problem is isolated to constructs which generate move literal instructions at the machine language level. The discrepancy between measured times and the times given in the M68020 processor manual suggest that there may be an error in the manual. The default of one additional prefetch cycle for literal moves will compensate for this.

For each *name* the Ada compiler *Ac* must previously have produced a corresponding *name*`.o` (in other words, *name* is the prefix of the name of the Ada source file *name*`.A`).

Wcet combines the corresponding object files into one temporary file called `Timing.o`, and gives that to the analysis programs. Intermediate data is written to the files `Timing.B`, `Timing.W`, and `Timing.S`. The results appear in a fixed format on the standard output, described below in §2.2. Unless the `−i` option is given, *wcet* discards the other intermediate files. They are intended for use mainly by the developers, not by the end user. The current format of the `.B` and `.S` files is given in Appendices A and B; the format of the `.W` files appears in the Implementation Guide for *wcet*. The *wcet* command invokes several other programs. It finds them in the directory `$WCETDIR/bin`. If the environment variable `WCETDIR` is not set and exported when *wcet* is invoked, it takes the default value `/usr/esa/wcet`, which is where the programs were placed in the original distribution of the software. If they are located elsewhere, `WCETDIR` must name the correct directory before *wcet* is invoked.

The timing analysis programs need measured or declared worst-case times for various run-time procedures, including those in Ada library packages. The programs look for these values in a file in a *wcet* library directory. *Wcet* takes the library name from the environment variable `WCETLIB`. If you do not set this variable, *wcet* uses a default value `$WCETDIR/lib` (and thus `/usr/esa/wcet/lib` if the variable `WCETDIR` has its default setting). If you prefer to keep the library local to the current directory (current project) you might instead set `WCETLIB=lib` (have the shell export it). Currently only one file must appear in that directory; its format is described in the next section. The values of `WCETDIR` and `WCETLIB` may be set independently.

2.1. File of Known Times

Wcet reads known (or declared) worst-case times from the file `$WCETLIB/Knowntimes`. Each (non-comment) line of the file has up to three fields, separated by spaces or tabs, and has one of the two forms below:

[*link_name*`,`] *Ada_name* ● [*cache_case_msec* ●] *worst_case_msec*

link_name ● [*cache_case_msec* ●] *worst_case_msec*

(The symbol ● shows where spaces separate the fields.)

Each line associates a known worst-case time with a subprogram or task entry. On any line, the character sequences `--` (as in Ada) or `#` (as in the UNIX command *awk*) cause *wcet* to ignore the rest of the line. Empty lines are ignored.

Execution time is expressed in (fractions of) milliseconds, represented by a floating-point number. If only one value appears, it is taken to be the worst-case time. If two values appear, the largest is taken to be the worst-case time regardless of the order in which the values appear. The *cache_case_msec* field is optional, since it is not currently used – *wcet's* final result is strictly worst-case – but it is convenient to allow for it in syntax, in case the system is later extended to handle it.

Time values may be zero, denoting a routine that can be ignored for scheduling purposes, either because it will not be called except on a fatal error (eg, certain routines that raise run-time exceptions), or because its effect is accounted for elsewhere.

A subprogram might be a run-time support routine, or a member of an Ada library package, including one of the standard packages. When the analyser must resolve a reference to a subprogram or entry, it will have the item's *link_name* (the name by which it is known to the UNIX linker), or its fully qualified *Ada_name*, or both. The analyser searches the known time file first for an entry matching both names (if it knows them), next for an entry matching the *link_name*, and finally for an entry matching the *Ada_name*. Each *Ada_name* must be written in lower-case letters; operator names can appear but should not be surrounded by quote marks.

This search procedure allows either link names (for low-level run-time support routines) or Ada names to be used as required. It also gives you precise control over the analyser's search for overloaded names, when needed. For example, the package `CALENDAR` declares the following operators:

```
function "+" (left: time;     right: duration) return time;  --(A)
function "+" (left: duration; right: time)     return time;  --(B)
```

The known-time file could have any of the following (unrealistic) entries:

```
calendar.+ 1
_S_calendar02,calendar.+ 2
_S_calendar02 2
_S_calendar0a,calendar.+ 3
```

Each has a different effect. The first entry matches both operators; the second matches only `(A)`; the third entry has the same effect without mentioning the operator name; the fourth entry matches only `(B)`. (The link names can only be discovered by examining the assembly code produced by the compiler; thus, unless you break the earlier prohibition of overloading names, you will only use link names if you are responsible for providing the known times for the run-time system and related packages.)

The *worst_case_msec* field can contain the word `UNKNOWN`; such an entry prevents resolution of references to the associated *link_name*.

2.2. Output

Wcet produces lines of output of two types: summary lines, and a line denoting an unresolved reference. There is exactly one summary line for each subprogram, each package body, each entry of a protected task, and each task body in the program (each *object* that might be of interest when scheduling). A summary line has the following fields:

Field Number	Contents
1	Object name (the fully qualified Ada name). An object that is renamed by one or more `rename` declarations will appear just once under its original name, not once per `rename`.
2	Source line reference: *source_file_name*`:`*line_number@code_address*
3	Basic block number: *#block_number* (Ignore this field.)
4	Line number of basic block in the intermediate file `Timing.B` (Ignore this field.)
5	The word `resolved`, if enough data is available to allow the *object*'s worst case time to be calculated; otherwise the word `unresolved`.
6	Worst case execution time in milliseconds (a floating-point number). This field is present only when field 5 is `resolved`.

For example, the summary lines from one example are shown below; the symbol • marks the space between fields:

```
motor.opcs_set_pump • motorsetpump.A:8 • #528 • 3300 • resolved • 1.01355
motor.opcs_not_safe • motornotsafe.A:4 • #519 • 3247 • resolved • 0.073
handler.opcs_start • handler_start.A:5 • #474 • 2979 • resolved • 8.4534
handler.opcs_sporadic_frequency_overrun • handler.A:12 • #403 • 2524 • resolved • 0.01695
```

When an object is marked `unresolved`, its summary line will be followed by lines naming the object(s) that prevented its resolution. Each such line has six fields:

Field Number	Contents
1	The character `+`.
2 to 4	Reserved (currently each field contains the character `-`).
5	The word `unresolved`
6	The name of the object that prevents resolution of the object named by the preceding summary line.

If a low-level object *X* prevents resolution of another object *Y*, perhaps by calling a run-time support routine that has not been included in the known time file, or because the unit containing the object has not been presented for analysis, all objects that call *Y* will also not be resolved. In other words, the 'prevents' relationship is transitive. The `unresolved` records for an unresolved object name just those unresolved objects that it calls directly.

Note that the structure of the output allows resolved objects to be selected by the *awk* program:

```
$5 == "resolved" {print $1, $6}
```

and unresolved objects to be selected by the *awk* program:

```
$5 == "unresolved" & $1 != "+" {print $1}
```

A list of all unresolved objects will be printed by the following shell pipeline:

```
wcet ... | awk '$5 == "unresolved" & $1 != "+" {print $1}' | sort -u
```

A particular object can be selected by an *awk* expression that includes a suitable pattern match or equality test of field `$1`. For example, the following *awk* program:

```
$1 ~ /^motor./ {print $1 $5 $6}
```

selects all objects declared in the package *motor*.

2.3. Interpretation of Times and Timing a Smaller Scope

An object's time includes the time for the execution of the bodies of any subprograms or protected entries called by the object directly or indirectly, taking into account any loops in which those calls might be nested.

For this purpose, any infinite loop (cycle) is regarded as having its body run *once*. Thus for a cyclical task, which contains some initialisation code followed by an infinite loop (during which it makes protected entry calls to other tasks), the worst-case time includes the initialisation code and one execution of the loop. Depending on the requirements of the Schedulability Analyser, it may be desirable to exclude the initialisation time (eg, if only the loop execution time is critical and the initialisation time is large). This can be done either by putting the initialisation code in a separate procedure, and deducting from the task object's time the time given by the procedure's summary line, or by putting the loop in a separate procedure, and using only the time for the procedure.

In general, any section of the program that contains constructions that cannot be analysed can be split into smaller procedures; the procedure containing such a construction, and any object that invokes that procedure, will remain unresolved, but times can be obtained for the other parts.

3. DIAGNOSTICS

There are four classes of diagnostics:

errors in the application of a command (usage or file access trouble)
warnings
fatal errors
internal errors

The first class of diagnostics should be self-explanatory (e.g. ‘usage’ and ‘cannot open *file*’ messages); file access messages usually include the diagnostic string provided by the operating system.

Otherwise, a diagnostic has the following format:

program_name **:** *diagnostic_class* **:** [*location_of_error* **:**] *diagnostic_text*

The *program_name* is `wcet`, `wet0`, or `segs`. The *diagnostic_class* is `warning`, `fatal error`, or `internal error`. The location of the error is typically given by reference to a file name, file name and line number, or (approximate) object code address depending on the nature of the *diagnostic_text*; the *location_of_error* text might also suggest the processing stage when the error was detected (eg, ‘reading annotations from file.o’). Diagnostics in each class are discussed below.

3.1. Warnings

After issuing a *warning*, the *wcet* programs continue processing.

- "*name* contains code for several Ada units (or a spec. and body)

  When more than one body (or a specification and a body) appears in a single **.A** source file, the compiler produces an object-file archive, not an object file. The archive cannot be handled correctly by *wcet*, because it uses the UNIX linker *ld* in an intermediate step, and *ld*’s linking rules differ for archives and object files. Ensure that specifications and bodies are kept in separate source files (if they are to be analysed). The output will certainly be incomplete, and some references might remain unresolved.

- no debug table entries found (was it compiled without -g?)

  The object file contains no debug symbol table entries, and more specifically, no structural annotations from the Ada compiler. Since the compiler’s **−g** option is on by default for ESA compilations, the most common cause of this complaint is that more than one body (or a specification and a body) appears in a single **.A** source file. This causes the compiler to produce an object-file archive, not an object file, which cannot be handled correctly (see previous entry). The output might be incomplete, and some references might remain unresolved.

- cannot find time for floating-point instruction ‘*name*’
- cannot find time for instruction ‘*name*’
- 68020 outer displacement extension word(s), not implemented

  If any of these diagnostics results from code produced by the Ada compiler, it can be treated as an internal error (see below). The analyser gives only a warning, since until the the program is repaired, it is still possible to make progress with a larger system by providing an estimate for the instruction time manually. The approximate instruction address and the contents of the **.B** file should help to track down the offending instruction. The instruction *name* is a code used internally; it will be similar to a 68020 instruction name.

- N_PC (block type) debug entry outside of block
- N_RBRAC (block end) debug entry without matching N_LBRAC (block start)
- duplicate N_PC (block type) entry
- unusual event: call pc 0x*address* maps to *name1* and to *name2*

  Each of these warnings might indicate an internal error in the analyser, but has been left as a warning since the results should be usable. Please report the occurrence of any of these to YSE.
- no time known for entry call implementation routine *name*

  An entry is required in the `Knowntime` file for the run-time support routine *name* (typically `_protenter` for protected entry calls and `_enter` for ordinary entry calls). All (protected) entry calls will remain unresolved until the time is provided.
- null unit entry for *name*

  No code segments for unit *name* were found in the `.S` file; check the source to ensure that this is plausible.
- unusual characters in identifier

  An identifier in the `Knowntime` file has non-Ada characters in its name. Check for unprintable characters on the indicated line of the file.

3.2. Fatal Errors

A fatal diagnostic indicates that processing stopped when the diagnostic was issued, and the *wcet* output may be missing or incomplete.

- *file* is not a recognised object file

  The given *file* does not have the magic number that marks it as a UNIX object file.
- blocks nested too deeply? limit: *n*

  If there is no instance in the source program of Ada blocks being nested more than 50 deep, treat the diagnostic as an internal error (see below).
- error reading *something*. *UNIX diagnostic*.

  Either an IO error or an end-of-file was detected when trying to read something (eg, code, symbols) from the object file, but the object file's control information said it was there. Check for object file corruption (eg, by file system running out of space).
- no string table found

  An object file was missing the table of identifiers specified by the UNIX object file format.
- out of memory ...

  A program tried to allocate memory dynamically using the C function *malloc*, but the space was refused. The diagnostic sometimes says what the program was attempting to allocate.
- unexpected reserved addressing mode

  The object file was corrupt, or could not be disassembled correctly: a text segment word that ought to have contained an instruction had an addressing mode that is not defined (reserved) on the 68020.

- unknown pc addressing mode 0x*address*

  The object file was corrupt, or could not be disassembled correctly: a text segment word that ought to have contained an instruction had an addressing mode that is *either* not defined (reserved) on the 68020 *or* is not implemented by the disassembler.

- bad format: require ‘identifier [cache-time] worst-time’

- illegal time value

  The indicated line in the `Knowntime` file does not use the syntax specified in the User’s Guide for the analyser. If it looks plausible, check for unprintable characters.

- time value too large for 32-bit integer once converted to CPU cycles

  Internally the analyser uses unsigned long (32-bit) integers to hold CPU cycle values. The indicated entry in the `Knowntime` file has a (floating-point) time in milliseconds that is too large when converted to represent a CPU cycle count.

- recursive or mutually-recursive sequence of calls

  The analyser cannot handle a program that contains such calls. The diagnostic is also produced for any chain of calls more than 50 levels deep.

- unit contains Ada code that cannot be analysed

  If this diagnostic appears, the compiler *Ac* ought to have produced a diagnostic warning about some construction that cannot be analysed. The construction must either be removed, or isolated in a source (object) file that will not be analysed.

3.3. Internal Errors

Unless an object file is corrupt, *internal errors* denote a logic error or shortcoming in the *program*. They should be reported to YSE.

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).

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 .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 `-` sb sl eb el
`call` loop_count cycles millisec called
`excall` loop_count cycles millisec called
`loop` loop_count
`oneof` loop_count

Cycles gives the worst-case execution time in CPU cycles; *millisec* is the same converted to milliseconds using the clock time given to *segs* by the user.

The `call` segment line:

`call` loop_count #block_number

gives the block number in the `.B` file of the called routine (this allows the corresponding segment to be located once it has been built).

The `excall` segment line has several variants:

`excall` loop_count #block_number,Ada_name
`excall` loop_count link_name[,Ada_name]
`excall` loop_count `P:`Ada_name
`excall` loop_count `E:`Ada_name
`excall` loop_count Ada_name

The first form is used when a segment is first built for a call that refers to a block in the current file; it gives the block number of the called segment. The second form denotes an external subprogram with the given *link_name* and *Ada_name*; such a segment might later be replaced by a `basic` segment if a known time has been provided for the called routine. The last three forms represent unresolved references to a protected entry, entry, or subprogram, respectively. The Ada name is always the fully qualified name of the called entity's definition.

The `basic` segment line:

`basic` loop_count cycles millisec `-` sb sl eb el

uses `-` as a place holder for the 'called name'. The values *sb* and *eb* give the starting and ending block numbers of the blocks included in this segment. The values *sl* and *el* give the starting line numbers of those blocks in the `.B` file.