Contents

List of figures

List of tables

CHAPTER 1

Introduction

1.1 Abstract

The most significant difference distinguishing real-time systems from other computer systems is the importance of correct timing behaviour. Each hard real-time task has a computation deadline[1] associated with it; the deadline has to be met, otherwise the real-time system fails. This constraint must always hold, even in the worst case, i.e. when the task's execution takes a maximum amount of run time. It is therefore obvious that the maximum execution time, or the maximum duration[2], of a task is of great importance for the construction and validation of real-time systems.

In many classic articles about scheduling in real-time systems, the maximum execution time is assumed to be known; unfortunately this is often not the case. The deadline is something an application programmer can easily specify, because it is usually part of the real-time implementation, but the duration is very hard to compute: it is often guessed, with the aid of experience, and then adjusted according to the results of various tests.

A standard, empiric method, consists in actually running a program on representative test data, and measure its execution time. While this approach is clearly useful, it has the same flaws as debugging: the test set may not cover the whole input-domain, maybe leaving the one yielding to the worst execution time untested.

1. The deadline is the point in time where the task has to be completed, from the programmer's point of view.
2. The duration of a task represents the amount of foreground time that the processor needs to completely execute it.

The ideal solution would be a profiler/analyser that could automatically determine the maximum execution time of a given program at compile time, but unfortunately this will remain a chimera. The modern operating systems' and processors' complexity, and theoretical limitations, prevents us from computing the exact and deterministic maximum duration of a given process.

The goal of this work, is to empower the user with an automatic tool that computes a good approximation of the maximum execution time of a given task. The profiler/analyser should automatize and speed-up one of the most error-prone developing phase of a real-time application, thus reducing the probability of faults.

We strongly believe, that the user interaction, in tools for predicting the timing behaviour of programs coded in high level languages, should be minimized. Real-time systems are used in research and industry fields, where noncomputer scientists are the vast majority of the users and programmers. To expect from the user a complete knowledge of the underlying system and hardware is not compatible with the goal of XOberon, which is about providing a framework for implementators looking for a rapid application development tool.

The work is divided in two major and distinct parts. First, we perform a syntactical analysis of the program source in order to retrieve its structure, using compiler optimization techniques. Particular attention is paid to the automatic computing of the number of iterations within a loop, allowing the transformation of the program's data flow into an acyclic graph. In a second phase, the processor architecture is analysed for computing the length of a code block. The instruction length is refined with run-time statistical information, to bring the worst-case results to values that one can expect when actually running the program.

1.2 XOberon

We integrated our tool in the XOberon hard-real–time operating system and compiler [1, 2]. XOberon is a hard real-time operating system developed at the Institute of Robotics (IfR), Swiss Federal Institute of Technology in Zurich, for the control of high-end mechatronic products. It is loosely based on the Oberon System [3]. Oberon refers simultaneously to a modular, extensible operating system and to an object-oriented programming language. The most recent version of XOberon is written in Oberon-2 [4], an improved revision of the

Oberon language, and takes advantage of the PowerPC processor architecture.

The system is particularly suited for the modelling of complex real-time applications, given its modularity, clean interface definitions and the presence of a dynamic loader, which checks for interface compatibility. The very fast compiler, along with the dynamic loader allows for short edit–compile–run cycles.

The operating system presents a clear, object-oriented interface to the programmer. The framework provides high level abstractions for most of the real-time programming problems. XOberon solves the majority of the usual real-time issues by implementing a deadline-driven schedule with admission testing. The user must provide the duration and the deadline of a submitted task. The real-time scheduler preallocates processor time as specified by the duration/deadline ratio. If the sum of all these ratios remains under 1.0, the scheduler accepts the task, otherwise it will be rejected.

The compiler, where the tool was integrated, is a slightly modified version of the PowerPC MacOberon Compiler (Oberon-2), which finds its roots in the original Ceres Oberon Compiler [3].

1.3 PowerPC 604e overview

This section describes the Motorola PowerPC 604e [12] used for this work. This small overview presents the processor characteristic concerned by this work, such as the processor and pipeline structure.

The 604e is an implementation of the PowerPC[1] family of reduced instruction set computer (RISC) microprocessors. It implements the PowerPC architecture as it is specified for 32-bit addressing, providing 32-bit effective (logical) addresses, integer data types of 8, 16, and 32 bits, and floating-point data types of 32 and 64 bits (single- and double-precision, respectively).

The 604e is a superscalar processor capable of issuing four instructions simultaneously. As many as seven instructions can finish execution in parallel, because the 604e has seven execution units that can operate concurrently. These units are:

- Floating-point unit (FPU)
- Branch processing unit (BPU)

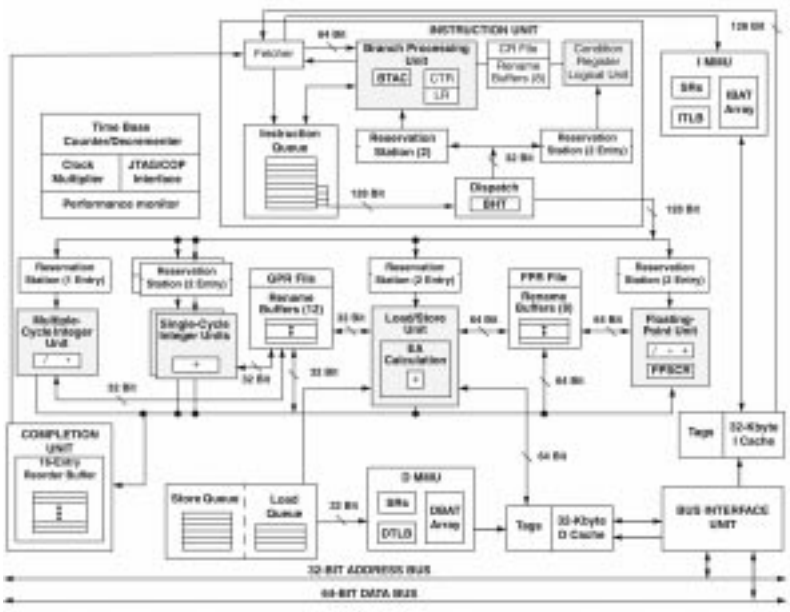
1. Performance Optimized With Enhanced RISC

- Condition register unit (CRU)
- Load/store unit (LSU)
- Two single-cycle integer units (SCIUs)
- One multiple-cycle integer unit (MCIU)

Instructions can execute out of order, and execution results can be made immediately available to subsequent instructions through register renaming. However, the completion unit retires (i.e. it commits results to architectured registers such as FPRs and GPRs) as many as four instructions per clock cycle in order, ensuring a precise exception model. To support out-of-order execution, registers are renamed to prevent write-after-read, and write-after-write conflicts. This renaming is accomplished by the mapping of architectural registers into physical ones.

The PowerPC 604e microprocessor uses dynamic branch prediction to improve the accuracy of instruction prefetching, and can speculatively execute through two unresolved branches.

FIGURE 1. 604e block diagram



The 604e has separate memory management units (MMUs) and separate 32-Kbyte on-chip caches for instructions and data. The 604e implements two 128-entry, two-way set associative translation lookaside buffers (TLBs), one for instructions and one for data, and provides support for demand-paged virtual memory address translation

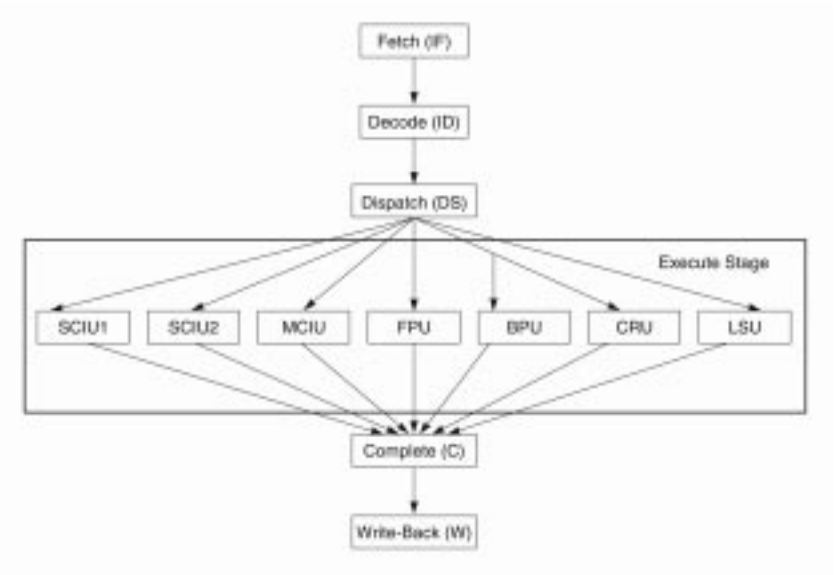
and variable-sized block translation. The TLBs and the caches use the least-recently used (LRU) replacement algorithm.

The 604e has a 64-bit external data bus and a 32-bit address bus. The 604e supports single-beat and burst data transfers for memory accesses and memory-mapped I/O accesses.

The master instruction pipeline of the 604e has six stages. Some instructions combine the completion and write-back stages into a single cycle. Some instructions (load, store, and floating-point instructions) flow through additional execution pipeline stages. The six basic stages of the master instruction pipeline are as follows:

- Fetch (IF)
- Decode (ID)
- Dispatch (DS)
- Execute (E)
- Completion (C)
- Write-back (W)

FIGURE 2. Pipeline diagram



Because the CRU shares the dispatch bus with the BPU, only one condition register or branch instruction can be issued per clock cycle. Both units (CRU and BPU) are treated as a single one by the performance monitor.

CHAPTER 2

Problem statement

The goal of the work is to build an automatic tool for predicting an approximation of the maximum duration of a task. The tool has to be able to analyse the task's source code, retrieve its structure and determine, when possible, the number of simple loop iterations. It should compute the number of repetitions for all the loops that are reconducible to a FOR construct, with constant starting, ending point and increment. This class of loops represent the vast majority of the cycles used in real-time programs.

Once the structure of the program is known and all the loops are bounded (automatically or by the user), the tool should compute the longest path in the program's data flow graph, representing the maximum duration.

To achieve the longest path computation, the tool must be able to track the length of the single instructions generated by the Oberon-2 compiler. It should use the static processor characteristic described by its architecture, in conjunction with statistical data about the task's behaviour in order to approximate the instruction length, to a meaningful value. When computing the maximal program duration, the worst case instruction length cannot be considered, because it has no practical meaning. Statistical data used for this approximation, include the instructions' stall cycles, the instruction parallelism, and the memory effects. The gathered information should summarize how the processor behaves when executing a given task.

Because these data vary in a significant way among different tasks, they must be collected separately for each profiled program. For this information to be collected, a performance monitor should be integrated in the XOberon System, capable of sampling data on a per-task base. The performance monitor should take advantage of the PowerPC 604e special monitoring hardware.

The computed approximation should have a practical meaning, describing the duration of the tasks to be submitted to the scheduler. The various approximations should be refined with an experimental strategy. The error result of various tests should be minimized, trying to avoid predictions falling under the measured run-time duration.

CHAPTER 3

Source-code analysis

In this chapter we describe the first phase of the work: the analysis of the program source, performed to extract its cycle-free data flow, by use of compiler optimization techniques. The longest path of the its acyclic data flow, weighted with lengths of each instruction in basic blocks, is equivalent to the maximal duration of the program.

3.1 Preconditions

In order to correctly compute the maximum execution time of a task, or maximal duration, some strong preconditions have to be met. These affect the program structure and the underlying hardware.

Definition 1 (Application Specific Maximum Execution Time $MAXT_A$) *The Application Specific Maximum Execution Time of a program is the maximum time needed to execute this program in the given application context.*

Note that the application-specific maximum execution time is the maximum CPU time that the task can actually consume, i.e. its duration. When trying to compute the timing behaviour of a task by means of the sole source code analysis, one can only derive the upper bound for its maximal time consumption. Hence the need for defining the calculated maximum execution time.

Definition 2 (Calculated Maximum Execution Time $MAXT_C$) *The Calculated Maximum Execution Time of a task is the least upper bound for the $MAXT_A$ of this task that can be derived from the program code, considering the worst timing behaviour of the underlying hardware.*

$MAXT_C$, as can be seen, is too high to have a practical meaning. The Approximated Maximum Execution Time is therefore defined as follows.

Definition 3 (Approximated Maximum Execution Time MAXT) *The Approximated Maximum Execution Time of a Tasks is an approximation of the $MAXT_C$ value with a given processor behaviour.*

The approximations are needed to determine the length of instructions, this being not deterministic in a pipelined, superscalar processor with deep memory hierarchies.

The control flow of a program obviously depends on the input data and global variable settings, determining the theoretical impossibility to compute the $MAXT_C$ for any program. This is a corollary of the termination problem, which states that it is impossible to compute whether a program will terminate in a finite time with a given input or not [5].

Theorem 1 (Termination problem) The termination problem (H := {<M> ω | M terminates on ω}) is not recursive.

If it is impossible to compute if a program will terminate, it is evident, that its length is not always determinable.

The problems that prevent us from computing the maximum length are the following:

- The number of loop iterations is not known.
- The depth of recursion is not known.
- Procedure variables instances are not known.

3.2 Changes in the Oberon language

We introduced some additions and limitations in the Oberon language to facilitate the source code analysis. The changes are effective only in real-time tasks, specified as compiler parameters. It is therefore allowed to have mixed (real-time and non-real–time) modules.

The changes, in detail, are:

- Recursive procedures, direct or indirect, are not allowed because it is generally impossible—or only with a great effort—to extract the recursion depth.

- No NEW is allowed, because this primitive is not bound neither in time nor in space.
- The loops that are not reconducible to a FOR construct have to be bounded—that is, the programmer must explicitly specify the maximal number of iterations.
- Procedure variables are not allowed, with the exception of Oberon-2 methods.
- The user has the possibility to specify the duration of a single statement.

We added two new Oberon constructs, the first for loop bounding (BOUND), the second to specify the length of a particular statement (LENGTH). Both new keywords are enclosed in comments, so that a modified real-time program, can also be parsed by a standard Oberon-2 compiler. The changes to the Oberon-2 syntax [4] are the following.

Changes in the Oberon syntax

```
WhileStatement = WHILE Expression DO [ (*BOUND
    Number*) ] StatSequence END.
RepeatStatement = REPEAT [ (*BOUND Number*) ]
    StatSequence UNTIL Expression.
LoopStatement = LOOP  [ (*BOUND Number*) ]
    StatSequence END.
ForStatement = FOR ident ":=" Expression TO Expression [BY
    Expression] DO [ (*BOUND Number*) ] StatSequence
    END.
statement = [ [ (*LENGTH Number*) ] assignment | [
    (*LENGTH Number*) ] ProcedureCall | IfStatement |
    CaseStatement | WhileStatement | RepeatStatement
    | LoopStatement | ForStatement | WithStatement | EXIT
    | RETURN [expression] ].
```

3.3 BOUND

Our analyser is able to compute the number of loop iterations for the majority of cycle structures transformable in FOR constructs. For all of the other loops, the programmer has to specify the maximum number of iterations, as in the following example:

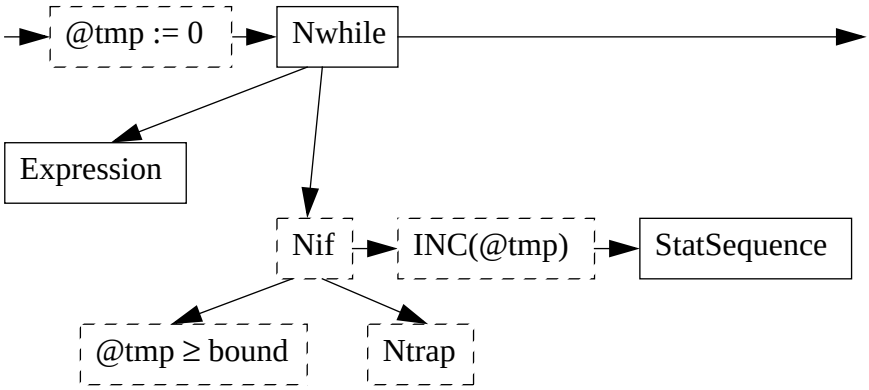
EXAMPLE 1. BOUND

```
WHILE ~done DO (*BOUND 1000*)
   GetData()
END;
```

This must be done, for every loop whose termination is determined by an input value, or a device signal, or the termination condition is too complex to be automatically analysed with a reasonable effort.

The BOUND construct does not only act as a compile-time hint; indeed code is emitted for a run-time check ensuring that the specified value will not be exceeded during execution. A new variable is introduced, and incremented at each loop iteration. In the termination condition a new check is inserted, and if the variable is greater than the bound value a run-time exception is generated (Oberon trap). This allows the user to detect an incorrectly specified value.

EXAMPLE 2. BOUND code generation[1]



3.4 LENGTH

The LENGTH construct is indispensable to profile method calls. The type of the object is not known at run time, and consequently it is impossible to compute the length of its methods. With this new statement it is possible to specify the maximum method duration in processor cycles.

EXAMPLE 3. LENGTH use

```
FOR i := 0 TO 1000 DO
   (*LENGTH 400*) portObj.GetData()
END;
```

1. This scheme represents a snapshot of the compiler's parse tree for a WHILE loop. Nwhile, Nif and Ntrap represent the internal structures corresponding to the Oberon constructs WHILE, IF … THEN … ELSE … END, and HALT. The dotted squares are newly inserted when a BOUND is specified.

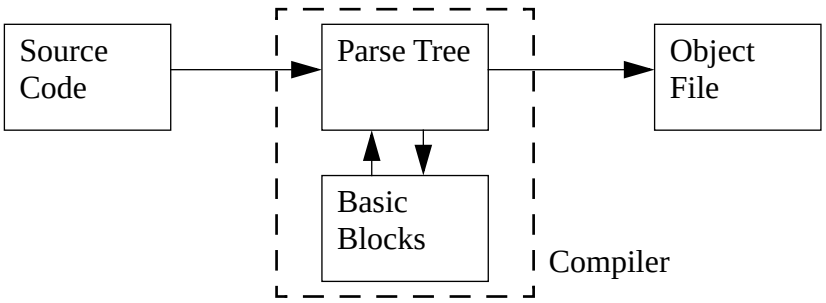
The length value cannot be checked at run time, because there is no way to implement a cycle counter and checker after each instruction. Particular attention is therefore required.

The use of object orientation in XOberon real-time tasks is usually moderate; these tasks are programmed in a procedural style with the exception of drivers software, dramatically reducing the need of the LENGHT keyword. A similar approach is not suited for plain object-oriented programming such as Smalltalk or Java, where the procedural programming is not allowed, and were each procedure (method) call should be prefixed by the length specification.

3.5 Intermediate representation

The XOberon compiler has a parse tree used as an intermediate representation between its front end and its back end. This structure is well suited for the existing Oberon compiler, which generates native code for the PowerPC and OMI object files (a portable Oberon binary file [6]). The parse tree, however, is not suited for data flow analysis, and therefore we introduced a new intermediate representation with basic blocks. The old intermediate representation was not substituted for reasons of practicality and time (a complete compiler rewrite were necessary) but is used in conjunction with the new one.

FIGURE 3. Compiler structure



The basic block intermediate representation contains information about the program structure but is not complete and can not be used for the code generation (for simplicity reasons). A basic block only contains links to the parse tree node, reducing its complexity and the time needed for the implementation.

The new dual structure is practical and useful for data flow analysis, but has the drawback that code changes in the parse tree are generally very difficult if not impossible.

3.6 Exceptions

In a MAXT analysis the time for exception handling should be inserted at each statement where an exception may occur. This is true for exceptional conditions that do not cause a program termination. If a trap is generated, the program is halted, and the maximal duration computation makes no sense any more. It would be useless and wrong to add the time for a divide by zero trap at each division.

Each XOberon task, starts up with no FPU processor support. The first floating point operation fires an exception. The system enables the FPU and the control is returned to the task. The time for this exception is added at the beginning of each procedure using floating point arithmetics.

This is obviously a worst case analysis, since the exception is fired only once. Unfortunately there is not the possibility to know in advance, if a task using the procedure has already generated the exception. Anyway the number of instruction used to enable the FPU is very small resulting in a minimal overhead.

3.7 Procedure calls

The registers that are used in given procedure are saved and restored in its prolog and epilog code respectively, in order to preserve the caller procedure state.

The code for saving, and restoring the floating-point registers is not generated for each procedure but is part of the system. When a procedure frame is created, a branch to a special memory area is performed, the registers are saved and the program branches back to the original place. Although these instructions are not generated by the compiler, they must be computed in the longest path analysis.

3.8 Inline procedures

The Oberon compiler allows the use of assembled code procedures (inline procedures). They are used in the system's and driver's programming, because they enable the generation of special instructions not supported by the compiler. If the structure of the used inline procedures is sequential, i.e. there are no branches, they are decoded, and the instructions are added to the corresponding basic block.

The presence of branches would require decoding with data flow analysis of the machine code, which would fall beyond the goal of

this work. This is not a severe limitation, because these procedure are generally made up of a small member of sequential instructions, which cannot be emitted by the compiler.

3.9 Imported procedures

The length of imported procedures is added to the corresponding block with the aid of a second symbol file containing their predicted duration. The use of a second file maintains the symbol file compatibility.

The structure of the additional symbol file is very simple and no consistency checks are performed.

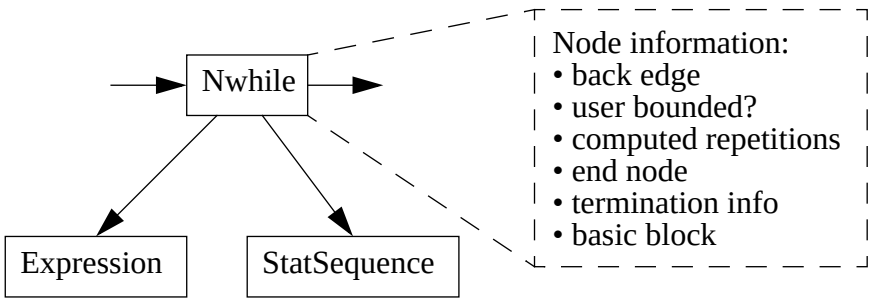
3.10 Loop detection

One of the biggest issue of source code analysis is the elimination of loops to transform the data flow in an acyclic graph. Due to the theoretical impossibility to compute the number of iterations for all the possible loops, this value, in previous works [7, 8], was asked to programmer, and included in program source with special keywords.

The simplicity of real-time tasks, implies that the majority of the loops has a FOR-like structure that permits an automatic computation of the number of iterations.

The parse tree simplifies the task of loop detection, because the whole information on the program structure is preserved. Loops are identified by special nodes (Nwhile, and Nrepeat), with information about their back edge. This avoids the construction of the blocks' dominator tree.

FIGURE 4. Parse tree structure



With the data stored in the parse tree nodes, loops are easily built using the following algorithm.

ALGORITHM 1. Loop detection

```
in:   head of the loop and back edge
out:  all the blocks in the loop

stack := ∅;
L := head(back edge) ∪ tail(back edge);
push(tail(back edge));
WHILE stack ≠ ∅ DO
    bb := pop();
    FOREACH p := pred(bb) DO
        IF p ∉ L THEN
            L := L ∪ p;
            push(p)
        ENDIF
    ENDFOR
ENDWHILE
```

All the predecessors of the block, where the back edge starts, are recursively added to the loop, until the block at the head of the back edge has been reached. This works only because the Oberon language guarantees that the program's flow graph is always reducible.

3.11 Loop termination

Once the loop has been detected, the profiler tries to determine how it is terminated. We restrict the analysis to loops terminated by a single variable, because they represent the majority[1] of loops used in real-time tasks. To reduce the set of different termination conditions, constant propagation and constant folding are applied to the code.

Constant propagation is a well-known global flow analysis problem. The goal of constant propagation is to discover values that are constant on all possible execution paths of a program, and to propagate these constant values through the program code as far as possible. Expressions whose operands are all constants, can be evaluated at compile time (constant folding) and the results propagated further. The constant expressions found are then substituted in the parse tree performing a real optimization pass.

After this important first simplification step, the relational expressions are simplified by reducing the operators and eliminating unnecessary boolean expressions. The set of commonly used expressions is

1. Generally more than the 95% but for more precise statistics see "Oberon language changes" on page 49

now reduced to a few patterns, which can be analysed by the compiler. In our analysis we considered the following structures:

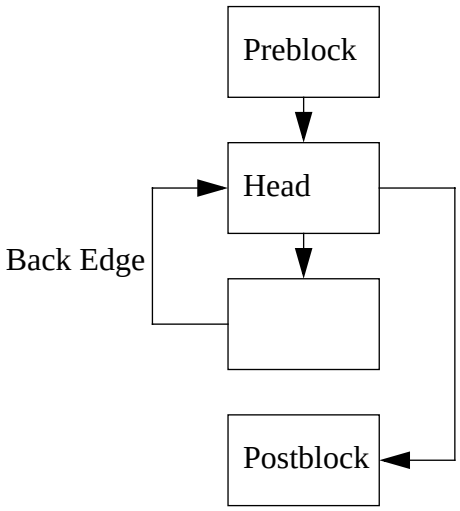
Termination conditions

```
(j relop const)
boolconst
(j relop const) boolop boolconst
~(j relop const)
```

Note that *j* is, in this example and in the following occurrences, the variable responsible for loop termination.

The next step is to determine the starting value of *j*, which corresponds to its value before the loop's head—that is in the loop's preblock. This value, if existing, was already computed, by the constant propagation pass.

FIGURE 5. Loop structure



The value of *j* at the end of the loop is included in the terminating condition.

The last step is to compute how *j* changes within the loop. To simplify the analysis, only one assignment to *j* within the loop is allowed, and this assignment can only be an increment or a decrement by an integer constant. The changes to *j* can be used for the computation of the loop's iterations.

Loops are analysed starting from the innermost one. This ensures that when the increment is computed, the number of block repetitions is known.

To test that the assignment is unique, we perform a reaching-definitions analysis, checking that all the definitions of *j* reaching the statement are not in the loop.

A definition of a variable *x*, is a statement that assigns, or may assign, a value to *x*. We say that a definition *d* reaches a point *p*, if there is a path from the point immediately following *d* to *p*, such that *d* is not invalidated along the path.

We have now all the elements to compute the number of iterations using the following rules:

- REPEATs are transformed in WHILEs inverting the relational operator and adding the increment to the starting value.
- The *greater or equal than* and *the less or equal than* operators are transformed in *greater than* and *less than*.
- The *greater* operator is transformed in the *less* operator.
- If the loop is a REPEAT an iteration is added, since REPEATs are executed at least once.

TABLE 1. Iterations computing rules

Operator	Iterations	Condition
=	∞	start = end $\wedge$ inc = 0
=	1	start = end $\wedge$ inc $\neq$ 0
=	0	start $\neq$ end
$\neq$	∞	start < end $\wedge$ (inc < 0) $\vee$ (inc > 0 $\wedge$ (end-start) mod inc $\neq$ 0)
$\neq$	0	start $\geq$ end
$\neq$	$\frac{\text{end} - \text{start}}{\text{inc}}$	start < end $\wedge$ (end-start) mod inc = 0
<	∞	start < end $\wedge$ inc < 0
<	0	start $\geq$ end
<	$\left\lceil \frac{\text{end} - \text{start}}{\text{inc}} \right\rceil$	start < end $\wedge$ inc > 0

EXAMPLE 4. Simplification of the termination condition

```
i := 200;
C := 100;
inc := -2;
Debug := FALSE;
…

REPEAT
    …
    i := inc + i
UNTIL (i ≥ C+1) & ~Debug
```

TABLE 2. Simplification example

Step	Start	End	Condition	Increment
-	?	?	(i ≥ C+1) & ~Debug	?
constant propagation	200	?	(i ≥ 100+1) & ~FALSE	?
reaching definitions	200	?	(i ≥ 100+1) & ~FALSE	-2
constant folding	200	101	i ≥ 101	-2
≥, ≤ removal	200	100	i > 100	-2
The REPEAT loop becomes a WHILE loop	198	100	i < 100	-2

3.12 User feedback

The user feedback about the bounding analysis, is an important aspect, because the tool can help the user to eliminate some common programming faults. Debugging infinite loops due to wrong termination conditions or missing variable increments is greatly simplified. User feedback is also very useful to help the user to adapt old code to the changes introduced by this work.

The following Oberon errors have been introduced.

Errors related to the new Oberon syntax:

- 601 illegal type of BOUND limit.
- 602 illegal value of BOUND limit.
- 607 LENGTH must be followed by a simple statement.

Errors related to the new real-time restrictions:

- 603 LOOPs are not permitted in real-time procedures.
- 604 NEW is not allowed in real-time procedures.
- 605 cannot profile inline procedure with branches.

Errors related to the loop's iterations computations:

- 606 computed number of loops is different from the specific value (BOUND).
- 620 infinite loop.
- 621 cannot compute the number of loop iterations.

The tool, thanks to the constant propagation analysis, is able to detect dead code, but because of the difficulties making changes in the parse tree, dead code is not automatically removed.

When the iterations are successfully computed and a BOUND value was specified for the loop, the tool checks the bounding validity. If the user specified value is different from the correct computed one, an error message is generated (606).

If the user has not bounded the loop, and the analyser is unable to compute the number of repetitions, an error is shown (621). This generic error includes the impossibility to compute the starting value of *j*, the ending value of *j*, or its increment.

Run-time errors

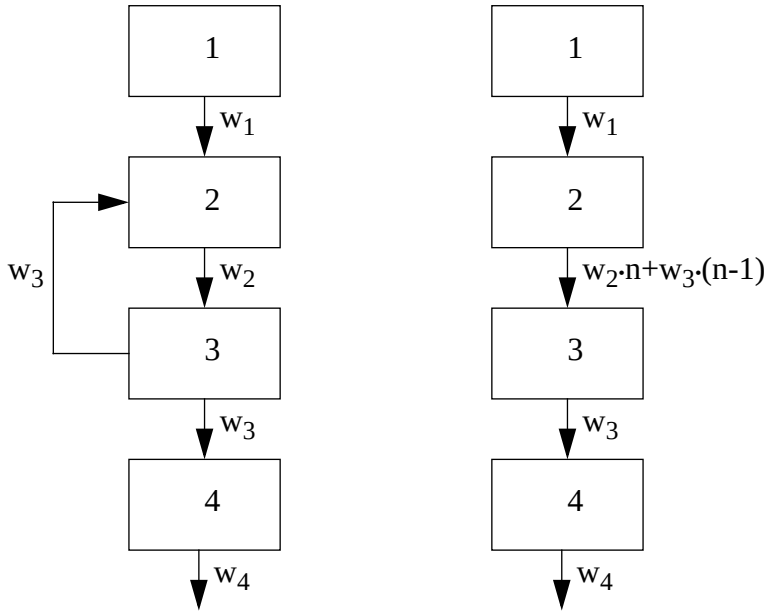
When a loop's number of iterations exceeds the BOUND specified value a run-time error (Oberon trap) is generated ("Number of loop iterations exceeds specified BOUND value").

3.13 Loop elimination

We can now remove the loops from the data-flow graph. Note that the loops are removed only from the basic blocks' graph, no loop unrolling is performed on the generated code.

At each basic block is assigned a field to store the number of time it is repeated. The loop elimination consists in the removal of its back edge, and multiplication of this field, by the computed or specified loop's iterations number for each basic block it contains. Loop heads are treated differently depending of the loop's type (WHILE or REPEAT).

FIGURE 6. Loop elimination



With this process we transformed the graph in a directed acyclic graph (DAG) allowing the computation of the longest path by means of an adaptation of the Dijkstra algorithm for single-source longest path [10]. The algorithm solves the single-source longest-path problem on a weighted, directed graph G=(V, E)[1] for the case where all the weights of the edges are non-negative ($w(u, v)>0 \mid (u,v) \in E$).

ALGORITHM 2. Longest path

```
S := ∅;
Q := V[G];
WHILE Q ≠ 0 DO
    u := min(Q); Q := Q \ u;
    S := S U {u};
    FOREACH v ∈ Adj[u] DO
        IF d[v] < d[u] + w(u, v) THEN   (* relaxation step *)
            d[v] := d[u] + w(u, v);
            pred[v] := u
        ENDIF
    ENDFOR
ENDWHILE
```

pred[] specifies the previous node on the longest path; d[] maintains the longest path from the start to the current node; Adj[] is an adja-

1. V = vertices of the graph; E = edges of the graph.

cency matrix describing the graph; Q is the queue of node to visit; S stores the longest path.

CHAPTER 4

A fine-grained approach to the duration computation

In the second phase of this work, the profiler analyses the generated code to determine the length of the basic blocks. The cycles needed for every instruction are approximated with the aid of run-time information, and added to the corresponding block during the code generation.

4.1 Hardware and system preconditions

To compute the exact length of an instruction, the underlying hardware and operating system must have a deterministic behaviour. The preconditions can be stated as follows:

- The effects of caching, pipelining and DMA performance on tasks performance are predictable.
- The operating system must only provide static memory management.
- Asynchronous interrupts are not present.
- Tasks synchronization is provided by a pre run-time scheduler and thus produces no overhead at run time.
- All resources are always available.
- The task is not interrupted.

Modern operating systems do not comply with these requirements, which are too strong to be considered. XOberon, as the majority of systems, provides run-time scheduling, pre-emptive multitasking, and dynamic memory management.

The dynamic memory management, and therefore the use of the MMUs (two in the case of the 604e), can cause different length by the address computation. The use of multitasking introduces context switches, where

the cache and pipeline stati are changed in an unknown way, destroying the predicted single task behaviour.

Timing indeterminism is a common characteristic of modern processor and systems, where several different user processes run concurrently, and where the processor performance can be strongly affected by the program code.

These restrictions do not prevent us to compute the maximum execution time for a given task; we could consider that all the memory accesses are always cache misses, that the branch prediction is always wrong, and that the pipeline is always flushed. A prediction, using this worst case assumptions ($MAXT_C$), would be unfortunately useless, because the task's duration would be too high to have a practical meaning[1].

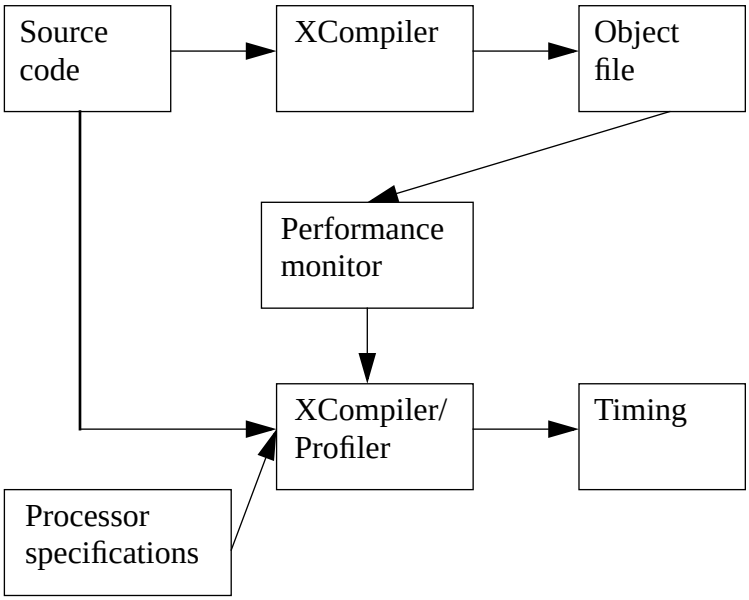
The analyser and profiler adapts the values specified by the architecture with the aid of statistical data as average stall cycles, average instruction parallelism, and memory effects to approximate instruction length with a normal processor behaviour.

Several XOberon tasks were tested to determine if the processor usage was homogeneous among the different processes. Unluckily we observed the impossibility to set standard data describing the task behaviour with a general validity. The presence or absence of floating-point computations, or different data dependencies in the code, result in very dissimilar behaviours.

Our approach consists in the run-time analysis of the task's processor use, monitoring it with the PowerPC 604e built-in Performance Monitor [12, 14].

1. As an example, the mean cache miss penalty for a Motorola MVME 2300 board is between 20 and 30 cycles, compared to 1 cycle for a L1-cache hit.

FIGURE 7. PowerPC 604e Performance Monitor implementation



The task is first compiled, and run-time monitored on the XOberon system. The performance monitor automatically generates a tasks' description file that is used by the compiler in conjunction with the static processor specifications to compute the instruction length.

4.2 PowerPC 604e Performance Monitor

The PowerPC 604e microprocessor provides a performance monitor facility to monitor and count predefined events such as processor clocks, misses in either the instruction or data cache, instructions dispatched to a particular execution unit, mispredicted branches, and many other occurrences. The count of such events (which may be an approximation) can be used to trigger a performance monitor exception, which generates a processor interrupt. The performance monitor facility is not defined by the PowerPC ISA architecture.

Because a software task scheduler may switch a processor's execution among multiple processes, and because statistics on only a particular process may be of interest, a process can be marked for profiling. The marking is done in the machine status register (MSR), which is part of a task's context, and is therefore saved and restored at each context switch. This feature is very useful to monitor only the to-be-profiled task, without performance losses. The marking is very simple and avoids any overhead; it consists in three instructions in the process initialization.

ALGORITHM 3. Process marking.

```
mfmsr[1]  R3        get the current value of MSR
ori [2]   R3, PM    set the PM bit
mtmsr[3]  R3        writes back the MSR
```

The performance monitor uses the following 604e-specific special-purpose registers: four performance monitor counters used to store the number of times a certain event has been detected, and two monitor mode control registers, which establish the function of the counters. Although the 604e supports a performance monitor interrupt that is caused by a counter becoming negative, we inserted the code for gathering statistics in the system's scheduler, avoiding additional overhead.

The overhead of the performance monitor routine in the scheduler is about the 21% (approximately 174 cycles against the 143 cycles of the original scheduler). Note that the monitoring is user activated and does not normally affect the system performance.

The performance monitor events are not precisely reported at the right time; instead they are signalled some cycles after the event has taken place. A DEC study [15] demonstrates that event counting does not accurately attribute events to instructions. They based their considerations on the Alpha 21164, the Pentium Pro, and the MIPS R10000 processors, but due to structural similarities, the consideration can be extended to the Motorola PowerPC 604 and 604e. Out-of-order speculative execution amplify the problem. This brings several difficulties when using the performance monitor counter negative interrupt mechanism. The user-chosen interval between the interrupts is not precisely respected, and an additional counter is needed to compute its length. Two of the four counters are thus unusable, because one is reserved for the interrupt triggering, and the second for the cycle counting. The integration of the performance monitor routines in the scheduler avoids the waste of the interrupt-triggering counter.

The system collects statistics for three events—one counter is always used to gather the number of cycles—in every scheduler cycle alternately for a total of 35 events. The length of a scheduler period reduces the inaccuracies of the event reporting, but eliminates the possibility to analyse the single instructions characteristics.

1. move from MSR
2. or immediate
3. move to MSR

To achieve faster context switching by an interrupt handler call, the XOberon system saves only the general-purpose register, deliberately ignoring the floating-point ones. Floating-point computations are not used in interrupts, because of an important speed improvement. This has however forced us to compute all the performance-monitor data with fixed-point arithmetic, which cause a precision loss, and a smaller maximum value for the used variables. Because of the order of magnitude of the values returned only their incremental mean value is stored.

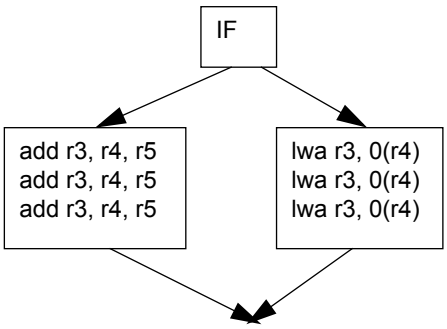
One of the new features introduced in the PowerPC 604e over the 604 is the condition register unit (CRU). The CRU executes all the condition register logical and flow control instructions freeing the branch prediction unit (BPU). Unfortunately the dispatch bus is shared between the two, so that only one condition register or branch instruction can be issued per clock cycle. The performance monitor was not updated and continues to consider both units as a single one. Consequently the profiler/analyser ignores the presence of the CRU; its statistics are included in branch prediction unit ones.

4.3 Cycles per instruction

The simplest way to compute an instruction's time is to use the popular *instruction per cycle* (IPC) metric. This is a poor metric to be used when discussing processor performance and the instructions' timing behaviour, because it does not lend itself to intuition about what the components of that performance are. The differentiation among the different instruction types is lost.

With a given IPC an integer load will be treated as an integer addition, dividing the memory effects and integer pipeline stalls to both of them. In the following example, the left path (three additions) would be erroneously considered equivalent to the right path (three loads), but they obviously have very different timing characteristics.

EXAMPLE 5. CPI for add[1] and load paths



The IPC approximation is unacceptable for our longest path computation because it could easily bring to the choice of the wrong path. The inverse of IPC is the *cycle per instruction* (CPI) metric, i.e. the mean instruction length. The advantage of the CPI metric over IPC relies in the fact that the first can be divided into its major components to achieve a better granularity.

$$\mathrm{CPI} = (\text{cycles per event}) \cdot (\text{events per intsruction}) \quad \textbf{(EQ 1)}$$

$$\text{time} = \frac{\sum \text{length}_{\text{static}}}{\mathrm{CPI}} \quad \textbf{(EQ 2)}$$

The total CPI for a given processor architecture is the sum of an infinite-cache performance and a finite-cache effect (FCE). The infinite-cache performance is the CPI for the core processor under the assumption that there are no cache misses, on the other hand, the FCE accounts for the effects of the memory hierarchy.

$$\mathrm{FCE} = (\text{cycles per miss}) \cdot (\text{misses per instruction}) \quad \textbf{(EQ 3)}$$

The misses-per-instruction component is commonly called the miss rate, and the cycle-per-miss component is called the miss penalty.

The FCE separation is not enough to have a good grasping of the processor usage by the different instructions. A natural and obvious additional classification is based on the different execution units. All the instructions dispatched to the same unit belong to the same group with common characteristic. The PowerPC performance monitor provides very fine-grained information about the different units, resulting in the perfect integration of the preceding taxonomy.

1. add = add between registers; lwa = load word algebric.

The tool can distinguish the mean instruction length, mean cycles, and idle time on a per-unit base, achieving a better granularity. To compute the instruction length: however, the mean instruction parallelism (p) has to be introduced, specifying how many instructions are executing concurrently,

The instructions length can now be expressed with the following equation:

$$CPI = \frac{length + stall_{unit}}{p} + stall_{dispatch} + FCE \quad \textbf{(EQ 4)}$$

The stall cycles in the dispatch unit ($stall_{dispatch}$) represent the time, measured in cycles, the instruction is blocked waiting for being dispatched to the execution units.

4.4 Instruction length computation

The static length of instructions is specified by the processor architecture, and is summarized in the following table.

TABLE 3. Execution latencies and throughput

Instruction	Latency	Throughput
Most integer instructions	1	1
Integer multiply (32x32)	4	2
Integer multiply (others)	3	1
Integer divide	20	19
Integer load	2	1
Integer store	3	1
Floating-point store	3	1
Double-precision floating-point multiply/add	3	1
Single-precision floating-point divide	18	18
Double-precision floating-point divide	31	31

The multiple cycle units (FPU, MCIU, and LSU) have internal pipelines, causing different throughput values depending on the instruction type. The consequence is that we are not able to know the length of a given instruction, even if we do not consider memory effects and pipeline stalls. The lack of the longest path trace, hinders the simulation of the pipeline, forcing the tool to find an approximation indicating whether a given instruction is pipelined or not.

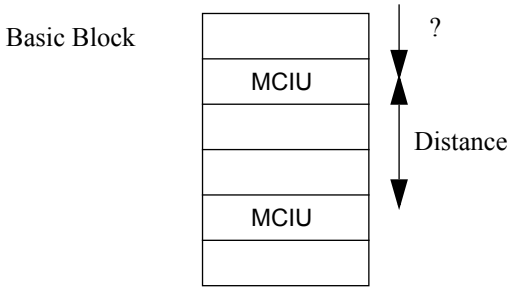
The compiler computes the distance (d) between the current instruction and the last one executed by the same unit. If the distance is less

than a given constant ($dist_{unit}$) the instruction is considered to be pipelined with its length corresponding to the specified throughput, otherwise the length is considered equal to the instruction's latency. The constant representing the maximal distance, differs among the three multiple cycle units, because they have different mean instruction length. Values between four and eight produced the best results in the performed tests. Thus equation 4 becomes:

$$CPI = \begin{cases} \frac{latency + stall_{unit}}{p} + stall_{dispatch} + FCE & d > dist_{unit} \\ \frac{throughput + stall_{unit}}{p} + stall_{dispatch} + FCE & d \le dist_{unit} \end{cases} \quad \textbf{(EQ 5)}$$

The problem with this approximation is that the distance is accounted only inside basic blocks. The first instruction of a given type in a block is considered always nonpipelined. This can result in big inaccuracies when the average length of the basic block is short (less than 10 instructions) since the percentage of wrong instructions length becomes significant.

FIGURE 8. Instruction pipelining



To overcome the issue, the following strategy has been implemented: since the compiler has no information about the preceding block, it can only assume that the structure will be similar for the whole task. Using the global loads of the execution units (gathered with the performance monitor), it computes the probability that a given instruction type will be present in the last d_{unit} instructions. If the probability is greater than a given threshold value we consider the length equal to its throughput.

$$prob = 1 - (1 - load_{unit})^{dist_{unit}} \quad \textbf{(EQ 6)}$$

To check how this approximation works, we confronted known mean instruction lengths (retrieved with the performance monitor) with the predicted ones, confirming the soundness of the method.

By separating the CPI in its major components the memory effects can be accounted to the concerned instructions (loads and stores), transforming equation 5 in its final form:

$$\begin{cases} FCE_{unit} = FCE & unit \in LSU \\ FCE_{unit} = 0 & unit \notin LSU \end{cases} \quad \text{(EQ 7)}$$

$$CPI = \begin{cases} \dfrac{latency + stall_{unit}}{p} + stall_{dispatch} + FCE_{unit} & d > dist_{unit} \\ \dfrac{throughput + stall_{unit}}{p} + stall_{dispatch} + FCE_{unit} & d \le dist_{unit} \end{cases} \quad \text{(EQ 8)}$$

4.5 Finite Cache Effect

Equation 3 describes the theoretical computation of the finite cache effect; unfortunately we cannot directly retrieve the two needed components from the performance monitor. The equation, separating the memory accesses in load and stores, can be expanded to:

$$FCE = miss_{load} \cdot penalty_{load} + miss_{store} \cdot penalty_{store} \quad \text{(EQ 9)}$$

Unluckily the performance monitor is unable to return the store miss penalty; we consider it equal to the load miss one, because of the great similarity of the two operations.

$$FCE = (miss_{load} \cdot miss_{store}) \cdot penalty_{load} \quad \text{(EQ 10)}$$

4.6 Dispatch stalls

The stall cycles in the dispatch unit are an important component of the CPI computation, describing processor hardware limitations, such as the lack of units and the lack of registers. The stall cycles in these units include even the instruction cache misses that are reported as a lack of fetched instructions.

In detail, the performance monitor gives us the following information about the dispatch stalls:

- Number of cycles the dispatch unit stalls, waiting for instructions.
- Number of cycles the dispatch unit stalls due to unavailability of reorder buffer entry.
- Number of cycles the dispatch unit stalls due to no floating-point register rename buffer available.

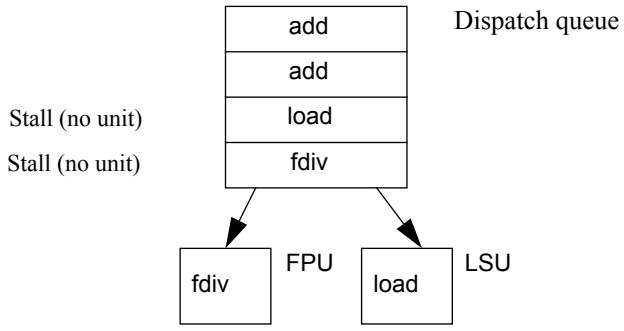
- Number of cycles the dispatch unit stalls due to no unit available.
- Number of cycles the dispatch unit stalls due to unavailability of general purpose register rename buffer.
- Number of cycles the dispatch unit stalls due to no condition register rename buffer available.
- Number of cycles the dispatch unit stalls due to CTR/LR[1] interlock.

The dispatch unit data are very fine grained, but we need a general criteria for the dispatch unit stall cycles, which corresponds to the logical union of all the above events.

There is no information about the superposition of the different events, and therefore there is no way to correctly compute a good melting of these events.

The dispatch unit has a four instruction length queue but the mean occupation is not monitored. Even worse is the fact that a reported stall could be caused by one, two, three or four instructions simultaneously, without noticing the difference. A stall caused by every instruction in the queue simultaneously will be reported as a single stall cycle. On the other hand, some stalls could be reported too early in the queue and be resolved at the time of the dispatch.

EXAMPLE 6. Dispatch unit queue instruction[2] example



In the preceding example two instructions stalls due to unit unavailability, but a single stall cycle is reported.

1. CTR = Count register
   LR = Link register.
2. add = integer addition (SCIU)
   load = data load (LSU)
   fdiv = floating-point division (FPU)

Several tests with different approximation strategies for the mean number of cycles an instruction stalls in the dispatch unit, leaded to the following computation:

$$p_{stall} = 1 - \prod_{event} (1 - p_{event}) \quad \textbf{(EQ 11)}$$

$$stall_{dispatch} = p_{stall} \cdot \frac{CPI}{4} \quad \textbf{(EQ 12)}$$

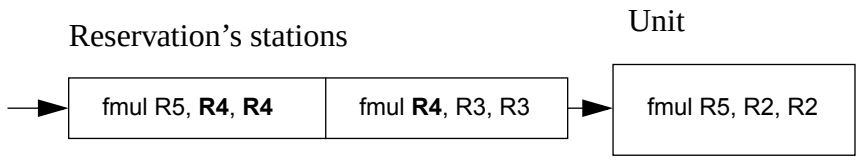
p_{event} is the probability a given event will happen. This value is reported by the performance monitor, since it corresponds to the percentage of stall cycles for each event. p_{stall} is the probability that a stall in the dispatch unit will take place, in other words, the inverse of the union of the probability that there are no stalls for each event.

With this approximation, we assume that no more that one instruction will stall at the same time in the dispatch unit. This is obviously not true, but gives good results.

4.7 Execution units stall cycles

The performance monitor specifications state that there is the possibility to gather the number of stall cycles for each unit. Unfortunately this does not hold. Each unit has two reservation stations where the dispatched instructions wait for execution. The performance monitor signals the number of data dependencies of the instructions in the unit's reservation stations. The number of returned stall cycles is thus higher than the right value (i.e. the stall cycles in the execution unit), because there is no guarantee that a dependence encountered in a reservation station will not be resolved when the instruction is executed.

FIGURE 9. Dependencies in the reservation stations, code example[1]



In this example a stall is reported because the two instructions waiting in the reservation stations depend on the result of the instruction exe-

1. fmul = floating-point multiplication

cuting in unit. However, no real stall will occur because when the instructions will reach the unit all of the operands will be available.

Stalls in single cycle units

The length of the instructions dispatched to this class of units is obviously known (one cycle). This allows us to compute the number of stall cycles in the following way:

$$\text{stall} = \frac{(1 - \text{idle})}{\text{load}} - 1 \qquad \textbf{(EQ 13)}$$

The time that an instruction remains in a given unit is the inverse of the unit's load, multiplied by the nonidle time, and indicates the sum of the instruction length (one) with the number of stall cycles. By reversing this formula equation 13 can be obtained, which returns the stall cycles of the target unit.

Since small errors can occur in the load and idle time monitoring, the computed stall cycles value is checked against negative values.

The computed, and correct, number of stall cycles greatly differs from the one returned from the performance monitor, whose value is often out of its validity range. In some tests, with tasks containing a lot of data dependencies, both single cycle integer units reported that the whole processing time was spent with stall cycles; the result was obviously wrong since the tasks terminated correctly.

Stalls in multiple cycles units

The mean instruction length in multiple cycle execution units (MCIU, FPU, and LSU) in not constant. This hinders the use of the preceding formula (equation 13) for the computation of the stall cycle.

The only information that the performance monitor provides, is the number of dependencies in the reservation stations (wrongly identified as number of stalls). Unfortunately the value is too distant from the correct one to be used without massive refinements.

Another problem in the stall cycle computations, is that the performance monitor does not return the total number of LSU stalls, or more precisely the number of dependencies in the reservation stations, but various subsets of the value:

- Number of cycles the LSU stalls due to BIU[1] or cache busy.

- Number of cycles the LSU stalls due to a full store-queue.
- Number of cycles the LSU stalls due to operands not available in the reservation station.
- Number of cycles the LSU stalls due to busy MMU.
- Number of cycles the LSU stalls due to full load-queue.
- Number of cycles the LSU stalls due to address collision.

As with the dispatch unit stalls we do not have the possibility to deduce the union of the different events, and an approximation must be computed.

Investigating these values with experimental methods, the events resulted generally disjoint. The LSU stall probability is therefore computed as the sum of the different events probability.

$$stall_{LSU} = \sum_{event} stall_{event} \quad \textbf{(EQ 14)}$$

To adjust the stalls returned by the performance monitor we first compute the mean occupation (in instructions) of the reservation stations considering them as an M/M/1 queue, and the execution unit as server. The unit's (server) utilization factor is:

$$\rho = (1 - idle) \quad \textbf{(EQ 15)}$$

With the utilization factor (equation 15) it is now possible to compute the average number of instructions in the reservation station (queue length):

$$N_Q = \frac{\rho^2}{1-\rho} \quad \textbf{(EQ 16)}$$

Since our queue is finite, values of N_Q are rounded down to a maximum of two.

The probability that an instruction will stall in a reservation station corresponds to the inverse of the probability that no instruction will stall in the queue. By reversing the equation the probability that an instruction will stall in a given unit can be obtained.

$$p_{stall} = 1 - (1 - p_{stall_{instruction}})^{N_Q} \quad \textbf{(EQ 17)}$$

1. Bus Interface Unit

$$p_{stall_{instruction}} = 1 - \sqrt[N_Q]{1 - p_{stall}} \quad \textbf{(EQ 18)}$$

The results obtained during several tests confirm that the stall cycles computed with equation 18 are closer to the reality than the monitored values, since the stalls are no more considered globally on all of the instructions present in the queue. However the approximation is not able to consider that some dependencies will be resolved before the instruction really enters the unit.

4.8 Instruction parallelism

The mean instruction parallelism is computed using global values gathered on a per-task basis. The parallelism can be determined by reversing equation 4; the overall CPI is provided by the performance monitor and the mean stall values are already approximated, but the mean instruction length needs to be computed.

To compute the mean instruction length (in the multiple cycle execution units) the mean stall value precedently computed can be used, subtracting it from the average time an instruction remains in the unit —in the same way as the stalls were approximated for the single cycle units.

$$\begin{cases} length_{unit} = 1 & \text{single cycle} \\ length_{unit} = \frac{1}{load} - \frac{idle}{load} - stall & \text{multiple cycle} \end{cases} \quad \textbf{(EQ 19)}$$

$$\overline{length} = \frac{\sum_{unit} load_{unit} \cdot length_{unit}}{load_{total}} \quad \textbf{(EQ 20)}$$

The total mean instruction length is the weighted mean of all the instruction classes.

The elements are now available for computing the mean instruction parallelism, which indicates how many instructions are executing concurrently.

$$p = \frac{\overline{length} + \overline{stall_{unit}}}{CPI - stall_{disp} - (load_{LSU} \cdot FCE)} \quad \textbf{(EQ 21)}$$

To include the finite cache effect we need to multiply it by the percentage of load/store instructions, limiting its effects to this class of instructions.

4.9 Some remarks on the instruction length computation

This chapter describes the approximations used to compute the instruction length, in cycles, of the generated code. We used several probabilistic formulae and queuing theory increasing the possibility of errors. The above computations were built and refined with an experimental method based on several tests. We were forced to this approach by the hardware used, which does not give us the possibility to monitor the single instructions and does not provide all the needed data.

CHAPTER 5

Results

This section presents the test strategy used for the tool evaluation, and the results obtained. The effects of the language and system changes are also evaluated.

5.1 Test strategy

There are several ways to judge the correctness of a profiler/analyser. First of all, the tool has to meet the requirements, in other words it must return values that have a practical meaning and can be submitted to the scheduler as the maximal duration.

This can be checked by letting the task run and measuring its length, which must always fall under the predicted limit. This kind of test is not very useful in the developing phase, since it is very difficult to state the correctness of the results. Moreover for large tasks we are not able to find the MAXT by hand, and therefore we are not able to check for the soundness of the predictor.

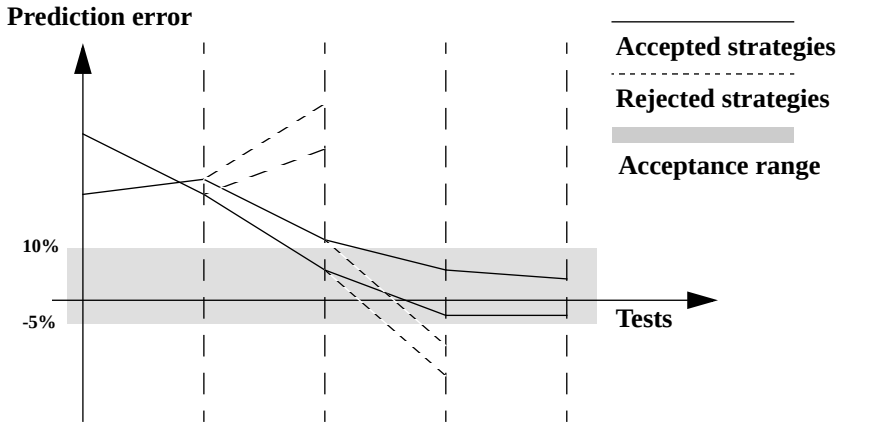
In order to have a better understanding of the produced results, a set of short typical tasks, where the longest path is a priori known, was tested. This gave us the opportunity to see the errors that the predictor makes when computing the instruction length, and eventually compare the different prediction strategies.

The drawback of the small-tests strategy is the homogeneity of the produced code. On the other hand, if the code exhibits strong variations, we are not able any more to understand the real meaning of the results, and, possibly, the causes of the errors.

Each new approximation strategy was tested on all of our examples—even the simplest ones—gathering accuracies of the various results. The results, or more appropriately, the errors were then analysed to check the correctness of the new strategy. The criteria used for accepting a new strategy can be summarized as follows:

- No predicted time must fall under the 95% of the measured (run-time) value.
- The average distance between the actual run-time and the predicted time should be significantly smaller than the one returned by the old strategy.

FIGURE 10. Example of test-strategy validation



The use of an experimental step-wise refinement technique was necessary due to the lack of information. The performance monitor is not powerful enough to allow a precise strategy development, and on the other hand, the system behaviour is not known.

5.2 Timing correctness when the longest-path trace is known

As precedently stated the set of the test samples with a known longest path trace was chosen, heading for simplicity, in order to maintain the homogeneity of the code behaviour. After the very first tests, the assortment of examples was concentrated on a handful of tasks including:

- Integer matrices multiplications (MatMul)
- Floating point matrices multiplications (MatMulFP)
- Searching the maximum in an integer array (Max)
- Searching the maximum in a floating point array (MaxFP)

- A modified Whetstone benchmark [7]
- Runge-Kutta method
- Polynomial evaluation
- Distribution Counting

The following table summarize the errors produced by the predictor with eight base tests.

TABLE 4. Test results

Test	Execution time	Predicted time	Error
MatMul	279.577 ms	310.553 ms	+11.08%
MatMulFP	333.145 ms	351.753 ms	+5.59%
Max	520.094 ms	555.015 ms	+6.71%
MaxFP	854.930 ms	814.516 ms	-4.73%
Whetstone	1958.308 ms	4248.548 ms	+116.95%
Runge-Kutta	79.052 ms	108.648 ms	+37.44
Polynomial evaluation	1251.194 ms	1187.591 ms	-5.08%
Distribution counting	2388.816 ms	2579.051 ms	+7.96%

The tests show good results, with the majority of the error rates in the range of -5% to +10%. The following sections explain in detail the causes of the pessimistic behaviour of the prediction.

5.3 Matrix multiplications and array maximum

The code of the first four tests is simple, nevertheless the examples include a good mixture of integer and floating-point computations, branches, condition checking, and address computations. The load-store unit is heavily stressed in the Max-tests where the memory access are also tested.

ALGORITHM 4. Matrix multiplications

```
FOR l := 1 TO loop DO
    FOR i := 0 TO dim-1 DO
        FOR j := 0 TO dim - 1 DO
            val := 0;
            FOR k := 0 TO dim - 1 DO
                val := val + m1[i, k] * m2[k, j]
            END;
            m3[i, j] := val
        END
    END
END
```

m1, m2 are the two matrices to multiply; m3 is the result; dim is the dimension of the matrices.

ALGORITHM 5. Array maximum

```
FOR j := 0 TO len-1 DO
    a[j] := 0
END;
FOR j := 0 TO loop DO
    max := MIN(TYPE);
    FOR i := 0 TO len-1 DO
        IF max ≤ a[i] THEN
            max := a[i]
        END
    END
END
```

a is the array to analyse; max the array maximum; len the legth of the array.

In the preceding code-snippet, the array maximum computation is not efficiently coded, since a max-value is unnecessarily substituted by an equal one (the array is always initialized with a series of zeros). This ensures that the longest path is always executed. The same effect can be obtained by using an array composed by an increasing strictly-monotonic series.

The prediction with these four tests produced very small errors (Table 4, “Test results,” on page 41). This confirms that when the profiled code presents a certain homogeneity the predictor is able to profile the processor usage with a good precision.

Although the four tests have similar prediction error results, they show a different processor behaviour. In the following table some of the process characteristics are shown.

TABLE 5. Test data[a]

	MatMul	**MatMulFP**	**Max**	**MaxFP**
FCE	0.0017	0.0014	0.1022	0.1145
p	3.66	3.12	4.09	4.17
IPC (mean)	1.8839	1.5804	1.1727	0.7103
$stall_{unit}$	0.51	0.26	1.66	2.66
length	1.10	1.32	1.00	1.77
$stall_{dispatch}$	0.09	0.13	0.19	0.33

a. The FCE, $stall_{unit}$, and $stall_{dispatch}$ are expressed in cycles per instruction.

Note that instructions parallelism greater than four are possible although the processor fetches only four instruction per cycle, because the processor parallelism is defined as the average number of instructions concurrently in-flight on the seven execution units.

The errors are due for the greatest part to the lack of an exact monitoring of the pipeline stalls, especially the number of stalls in the execution units, that reached 12 cycles per instruction for the FPU in the MaxFP test. The cause of this value (clearly too high) is that all the FPU computations are dependent on the operands' address computations. Most of the dependencies are considered as stalls, although they are resolved before leaving the reservations stations.

Another part of the error is surely to attribute to the instruction length approximation that, in the case of small basic blocks, (as it is in the tests) can produce some imprecisions.

5.4 Whetstone results

The Whetstone test is the major synthetic benchmark program, intended to be representative for numerical (floating-point intensive) programming. Based on statistics gathered at the National Physical Lab in England, using an Algol 60 compiler, which translated Algol into instructions for the imaginary Whetstone machine. The compilation system was named after the small town of Whetstone, outside the city of Leicester, England, where it was designed [16].

Synthetic benchmarks try to match the average frequency of operations and operands of a large set of programs, but are often not representative of the reality[1].

The benchmark is a collection of eight (in the original version, eleven) different small tests:

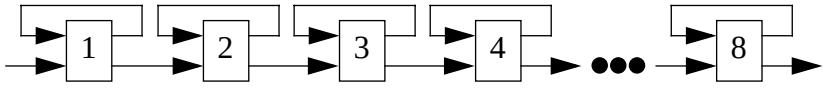
- Simple identifiers (floating-point arithmetic)
- Array elements (floating-point arithmetic with array elements)
- Array as parameter (the above test embedded in a procedure, the array is passed as a reference parameter)
- Conditional jumps (if-then-else statements)

1. "No user runs synthetic benchmarks, because they don't compute anything a user could want. Synthetic benchmarks are, in fact, even further removed from reality because kernel code is extracted from real programs, while synthetic code is created artificially to match an average execution profile. Synthetic benchmarks are not even *pieces* of real programs, while kernels might be."
—Hennessy and Patterson [17, p. 21]

- Integer arithmetic
- Procedure calls (floating-point arithmetic embedded in a procedure)
- Array references (assignments between array elements)
- Integer arithmetic (additions and subtractions)

The test parts are embedded in completely disjoint cycles, repeated many times.

FIGURE 11. The structure of the Whetstone benchmark



The benchmark can be considered as a collection of small separate code patterns. The different fragments are not similar and generate very different processor utilisation data; moreover they exhibit very different instruction parallelism. With our approach, statistics are computed as mean values, and they do not apply well to the very different code parts of Whetstone, leading to prediction errors. By analysing the different benchmark sections separately we found errors in the range -5% – +10%, as in the other four simple tests.

This test can be seen as a good method to observe the predictor's behaviour, when the heterogeneity of the code is maximally stressed. Although the difficulties presented by the code pattern hinder a precise prediction, the results are acceptable within a factor-2 of the correct value.

When developing the different approximations, attention was paid to the consequence of errors, trying to prefer higher values over lower ones. This test is compliant to this technique, since the predicted duration is higher than the worst case.

Notice that, if the specified duration for a task is too high, only the system performance is affected (in term of utilization); on the contrary a value under the effective run time would undermine the system's stability.

5.5 Runge–Kutta method

The Runge-Kutta method numerically computes a solution for differential equations, approximating the solution of the initial value prob-

lem y'=f(t, y) with y(a) = y_0 over [a,b] with an error control and variable step-size method.

ALGORITHM 6. Runge-Kutta Method[1]

```
h := 0.1; x0 := x0; y0 := y0;
FOR k := 0 TO len - 2 DO
    xk := x0 + k*h;
    k1 := f(xk, yk);
    k2 := f(xk + 1/2*h, yk + 1/2*h*k1);
    k3 := f(xk + 1/2*h, yk + 1/2*h*k2);
    k4 := f(xk + h, yk + h * k3);
    yk+1 := yk + h/6 * (k1 + 2*k2 +2*k3 + k4)
END
```

The high number of function calls, present in this test, cause a prediction higher then the correct value. The procedure calls, in fact, reduce the code locality. As in the preceding examples the number of stalls in FPU is wrongly reported.

5.6 Polynomial evaluation

This small test combines memory accesses and floating point computations evaluating a polynomial. We used the well-known Horner's rule to compute the value of the polynomial p at the point X.

ALGORITHM 7. Polynomial evaluation

```
FOR k := 0 TO n-1 DO
    x[k] := 1.5;
    c[k] := 0.4
END;

FOR i := 0 TO len - 1 DO
    p := c[n-1];
    FOR k := n-2 TO 0 BY -1 DO
        p := c[k] + (X + x[k])*p
    END
END
```

The code uniformity of this test produce good results, near to the correct value. The small error is probably caused, as in many other cases, by the wrong number of stalls in the FPU that reached 14 stall cycles per instruction. The code produced does never use the MCIU reduc-

1. $y(x)=e^x$, f(x, y)=y

ing the number of used units, and provoking a higher stall value in the dispatch unit due do unit's unavailability.

5.7 Distribution counting

The distribution counting algorithm sorts a file of N records whose keys are integers between 0 and M. The idea is to count the number of keys with each value and them use the counts to move the records into position on a second pass through the file.

ALGORITHM 8. Distribution Counting

```
FOR j := 0 TO M-1 DO
    count[j] := 0
END;
FOR i := 0 TO N-1 DO
    count[a[i]] := count[a[i]] + 1
END;
FOR j := 1 TO M-1 DO
    count[j] := count[j-1]+count[j]
END;
FOR i := N-1 TO 0 BY -1 DO
    b[count[a[i]]-1] := a[i];
    count[a[i]] := count[a[i]]-1
END;
FOR i := 0 TO N-1 DO
    a[i] := b[i]
END
```

This test presents a very low error rate (8%) since it does not use floating-point arithmetic, and uses only sparingly the MCIU, avoiding the stall imprecisions. This demonstrate the soundness of the predictor when the performance monitor data are attendible.

5.8 Drivers timing

The only obstacle to the MAXT computation for actual real-time applications, is the XOberon drivers' structure. The drivers are objects retrieved from a database, by querying their names. It is obviously not possible to know the duration of a driver method at compile time, but the LENGTH construct helps us to specify its length.

We only need to compute the different methods' length for the drivers used by the profiled task. A theoretical length measurement is very difficult because of the many hardware dependencies, but fortunately the drivers normally have a linear structure, resulting in very uniform

execution times—they normally perform a simple read or write access. This allows an experimental measurement of their duration over a big set of samples. The next example presents the code pattern used for the drivers timing.

EXAMPLE 7. Driver method length measurement

```
GetObj("name", obj");
PerformanceMonitor.Start();
FOR i := 0 TO Iterations-1 DO
    monitorOn();
    obj(type).method;
    monitorOff()
END;
PerfomanceMonitor.Stop();
length := PM.cycles/Iterations - length_monitorOff
```

The test module retrieves the object from the system database (GetObj) and repeatedly invokes the driver's method a given number of times. The performance monitor is switched on and off, exactly before and after the call to avoid the loop interferences.

With this simple scheme we are able to compute with a minimal effort the driver method's lengths. Note that this measurements must be done only once for a given driver on a given machine.

5.9 LaserPointer

The first test on a real application was done on the LaserPointer. The LaserPointer is a test and example machine, which moves a laser pen applied on the tool-centre-point (TCP) of a 2-joints (2 DOF) manipulator.

The three real-time tasks of the robot-control application were profiled, with the following results.

TABLE 6. LaserPointer results

Task	Measured value	MAXT
WatchDogHandler	869 cycles	860 cycles
PlannerHandler	891 cycles	894 cycles
ControllerHandler	2189 cycles	3214 cycles

The WatchDogHandler is a very small task with a linear code, and therefore the longest path (i.e. the only existing path) is always taken.

The result for this test (-0.99%) is to be compared to the first four simple tests of the preceding section.

The other two values are more difficult to check, since we are not able to force the longest path execution at run-time, but the results are coherent: they are greater or equal than the measured value, and they have a plausible value.

The application required very little adaptation to be profiled: we only inserted five LENGTH constructs for the drivers calls, and changed the procedure MathL.exp, which was recursive.

5.10 Hexaglide

The Hexaglide is a project developed at the Institute of Machine Tools of the Swiss Federal Institute of Technology Zurich [18]. The machine is a parallel manipulator with 6-DOF intended to be used as a high speed milling machine.

TABLE 7. Hexaglide results

Task	Max run-time	MAXT
TrajectoryHandler	0.058 ms	0.052 ms
DynamicsHandler	0.286 ms	0.451 ms
LearnHandler	0.065 ms	0.065 ms

The table presents the results obtained by profiling three real-time tasks of the Hexaglide software. The first column contains the maximal duration noticed at run time and the second one contains the predicted MAXT. We achieved very good results very near to the measured maximal value.

5.11 Related work

The source level analysis approach was already used in several works [7, 8], with a large use of bounding constraints, using deterministic CISC machines. These works compute good predictions, but the approach does not have a practical meaning when applied to modern processor architectures where the instruction length is not simply deducible.

P. Pushner and Ch. Koza [8] presented an interesting refinement to the longest path approach, adding knowledge to analysed programs. The user inserts information in the source code, since he knows the program characteristics, to help reducing the worst case.

There are several run-time monitoring projects, but because of the higher precision requirements they are usually deployed on custom architecture with heavy hardware support, such as the ProfileMe DIGITAL project [15]. Our profiling approach is too imprecise for discussing processor performance or evaluating produced code, but it adapts well to the charter of this work. As indicated the other actual performance monitor tools achieve much better accuracies but require hardware support or breakpoints in the profiled task.

5.12 Oberon language changes

The influence on existing code brought by the new Oberon constructs is very small, as we noticed in the performed tests. The changes needed for profiling a big application with the new tool are negligible, with average rates of 3–4 BOUND introductions every 100-Kbytes of code. The system software needed only a small change in the exponential function computation (MathL module), which was programmed recursively. Small applications like the LaserPointer were profiled without any adjustment (excluding the drivers' length specifications). The number of different driver's calls in an application, i.e. different LENGTHs to specify, is normally small, reducing the user's effort. In the following table the number of changes in the code for some applications are shown.

TABLE 8. Changes in the source code of existing applications

Module	Length	BOUND	LENGTH	Other
LaserPointer	25 Kbytes	-	5	-
MathL (FP library)	14 Kbytes	1	-	1 recursive procedure
Hexaglide [18]	54 Kbytes	2	4	-
Performance Monitor and scheduler	20 Kbytes	1	-	-

5.13 Optimization performance

The constant propagation optimization performed on the code does not normally improve—or only in minimal part—the tasks performance, because of its simplicity and little degree of utilization. A complete common subexpression elimination would for sure be of greater benefit, especially in array indices computations.

5.14 Penalties when using the performance monitor.

The code to enable the performance monitoring is, has seen, very simple and does not affect the system performance. The additions to the scheduler to gather the monitored data cause, indeed, an overhead. The insertion of code, for approximately 31 processor cycles, in the scheduler produces a very small performance loss, i.e. about 0.10%.

5.15 Real-time Oberon programs

The work gave us the possibility to analyse the processor behaviour with the XOberon system and several application. This data, although not directly relevant to the problem, are very interesting and deserve to be mentioned.

The following table shows typical values gathered by monitoring the XOberon system code alone, and with some real-time tasks running.

TABLE 9. Performance monitor data

Event	XOberon	Applications
IPC	0.59	1.10
CPI	1.68	0.91
p	1.86	2.88
Loads (misses)	25.8% (0.66%)	12.41% (0.05%)
Stores (misses)	13.22% (0.00%)	4.93% (0.12%)
FCE	0.04	0.00
Miss penalty	25.62	18.53
Stall cycles in the execution units	1.32	0.87
BPU load	19.2%	13.6%
SCIU load	20.0%	61.4%

Although L1-cache misses on the used boards have high penalties—between 20 and 30 cycles—the FCE is always small. This is a direct consequence of the real-time program's simplicity, leading to a big code locality. The compactness of the XOberon kernel contributes to the good utilisation of the on-board caches, usually fitting in 4–8-Kbytes data–cache for the scheduler's basic operations. The reason of the reduced memory accesses is the small use of dynamic structures by the real-time processes.

The average IPC is usually small, compared to the processor maximum of four. This is a direct consequence of the smallness of the

basic blocks, and the high number of branches. Data dependencies worsen the situation further by lowering the code parallelisation. The XOberon system code has very few mathematical computations, and contains a lot of branches (20%) resulting in a very low IPC (0.6).

CHAPTER 6

Conclusions and future directions

6.1 Conclusions

This works shows the feasibility of a source code timing prediction tool for real-time tasks and its limits. The simplicity of the programs allows for an automated analysis, but the modern hardware hinders a precise prediction. The performance of modern processors is becoming increasingly difficult to understand: the dynamic nature of out-of-order issue and completion, coupled with dynamic branch prediction, speculative execution paths, and with the complexity of deep memory hierarchies, makes it impossible to predict program behaviour solely through static code analysis.

The run-time monitoring approach is very helpful when, as in this work, the tasks are simple and relatively homogeneous, but shows its limits when the application complexity grows. The hardware performance counters found in existing processors, which cannot accurately attribute events to the single instruction types, do not allow a precise prediction.

The instruction length computations showed the soundness of the cycle per instruction metric separation suggested by P.G. Emma in timings prediction as in the field of performance evaluation [11].

The obtained results encourage the use of the tool in the real world, producing meaningful values and avoiding their guessing by the programmer.

The clean integration of the performance monitoring tool in the XOberon system provides the user with a transparent infrastructure with a low overhead making it practical for continuous profiling, although the retrieved data does not guarantee a big precision.

A wealth of additional information was also collected, providing interesting data about the interactions between instructions, including concurrency levels, and pipeline utilization, in either the XOberon system or user applications. This helps the programmers and compiler-writers to better understand the processor behaviour, and to improve the code quality.

6.2 Future directions

The source code analysis has place for several improvements, especially in the terminating conditions where more complex structures could be analysed. Recursive depth computation could also be implemented.

An important progress could be achieved by the introduction of the programmers' knowledge in the source code analysis. The user could help the profiler to eliminate excluding source paths.

EXAMPLE 8. Lack of knowledge

```
cur := first;
WHILE cur # NIL DO   (*BOUND 10000*)
    IF cur.name = "Dilbert" THEN
        DoSomething(cur)
    END;
    cur := cur.next
END;
```

In this short example a long list of people is scanned searching for the ones named Dilbert, and if the element is found some operation is done on the person's record. The user could know that the number of individuals with the searched name is very small, but the profiler adds the time for the IF construct to each iteration. The introduction of knowledge could allow the user to specify a maximal number of executions for a given code snippet.

The instruction length computation with the performance monitoring approach has also room for several improvements. A monitoring tool giving information on an instruction basis could be used to increase the accuracies, or the measurements could be integrated with pipeline simulation in order to better understand some phenomena as the pipelining in the single execution units.

APPENDIX A

File formats

The following section provides the format of the processor description file, and the performance monitor information file, described in EBNF syntax.

A.1 Processor description file

The processor description file contains static information about the PowerPC architecture. It includes the processor clock and a list with all the instructions indicating their timing characteristics.

```
File = NAME String FREQUENCY number InstructionList.
Serialization = “-” | “Execute” | “I/O” | “FPempty” |“Postdispatch” |
    “String/multiple” | “Dispatch/execute” | “Complete”.
Unit = “MCIU” | “SCIU” | “FPU” | “BPU” | “CRU” | “LSU” |
    “Completion”.
Bus = “1” | “0”.
Early = “1” | “0”.
InstructionList = INSTRUCTION { Name Unit Length Throughput
    Bus Regs Early Serialization }.
```

Early exit indicates instructions that can complete earlier than specified in case of special conditions. *Bus* indicates a bus access by the instruction. *Regs* indicates the number of register involved (as in string manipulation operations).

EXAMPLE 9. Processor description file

```
NAME PPC604e
FREQUENCY 300
INSTRUCTIONS
add SCIU 1 1 0 0 0 -
```

```
adde SCIU 1 1 0 0 0 Execute
stmw LSU 2 2 0 1 0 String/multiple
fres FPU 18 18 0 0 0 FPempty
eciwx LSU 2 2 1 0 0 Execute
```

A.2 Performance monitor information file

PMInfoTag = 07X.
File = PMInfoTag CPI FCE p $stall_{dispatch}$ $stall_{SCIU}$ $stall_{MCIU}$ $stall_{FPU}$ $stall_{BPU}$ $stall_{LSU}$ $load_{SCIU}$ $load_{MCIU}$ $load_{FPU}$ $load_{BPU}$ $load_{LSU}$.

All the values are IEEE 754 single precision values in little endian mode.

APPENDIX B

Implementation problems

B.1 Compiler integration

The structural complexity of the Oberon compiler has grown in the last years. The original Ceres Oberon compiler was adapted to the various architectures (Motorola 680x0, Intel 80x86, MIPS, POWER, and PowerPC) and to the language changes (Oberon-2). The parse tree as intermediate representation was added to support the code generation for different processors. The lack of a compiler rewrite with a clean intermediate representation, hinders the addition of optimization techniques.

The profiler/analyser generates useful data, which could be used to improve the code quality, but unfortunately the work needed to introduce the changes is too big, nearly approximating a complete rewrite.

B.2 Constant propagation and reaching definitions

To compute the number of loop iterations, we performed some data flow analysis: constant propagation and folding, and reaching definitions.

The introduction of the additional intermediate representation and the large use of *use-def* chains and *bit-vectors* caused several memory problems, because there is no automatic memory reclamation during an Oberon command execution. The MacOberon garbage collector is called only after a constant number of command's calls. We were thus obliged to maintain an alternative memory management with a pool of disposed memory blocks to recycle.

B.3 Performance monitor

The biggest problem with the performance monitor implementation was the faulty documentation. The PowerPC 604e User Manual [12] is full of typographical and structural errors.

A wealth of small errors are present in the specification of the performance monitor events (e.g. unit misleading, wrong bit numbering).

As precedently seen by the computation of the instruction stalls, some events are even wrongly specified. By the multiple cycle units the number of missing operand (unresolved dependencies) in the reservation stations is stated as the number of stall cycles.

APPENDIX C

User interface

C.1 Oberon Compiler

In this small section the new compiler options needed for the profiling tool are presented.

We added the possibility to tell the compiler to ignore the module SYSTEM during the cycle length computation (\S option). SYSTEM provides the user the possibility to write data at a given memory address without any checks. This feature is very useful and indispensable when writing drivers, but insane for code optimization. A generic memory write invalidates everything that was precedently defined since it could destroy any stored data structure.

The programmer, which specifies to ignore SYSTEM, asserts that memory accesses are only used to access peripherals, and that no program variable is touched by SYSTEM calls.

When not specified, every SYSTEM memory access invalidates all the constant definitions resulting in very poor constant propagation results. Since the constant propagation, and reaching definitions analysis are essential to the automatic loop bounding, we chose to allow this compromise, although the use of SYSTEM could be inherently dangerous.

The procedures to be profiled are specified between the compiler options and the module name in the following way:

```
[ "[" procname [ "+" ] { "," procname [ "+" ] } "]" ]
```

The “+” sign after the procedure name indicates that the tasks or procedure must be run-time monitored. This cause the compiler to set a bit in

the machine status register of the task, so that the PowerPC performance monitor will consider it during the measurements.

In order to profile a procedure the performance monitor information file and the processor description file must be in a Oberon System readable directory (Oberon path).

C.2 PowerPC Performance Monitor

The PerformanceMonitor module acts as the user interface for the performance monitor. It provides the following commands:

- PerformanceMonitor.Start

  Starts the tasks' monitoring.
- PerformanceMonitor.Stop

  Stops the tasks' monitoring. No data is cleared.
- PerformanceMonitor.Info

  Prints general information about the tasks processor utilization, including the units loads, stalls, and idle cycles, the memory accesses, and cache misses.
- PerformanceMonitor.Info taskId ~

  Same as above, but prints information about the specified real-time periodic task. The performance monitor in not able to distinguish the different run cycles of a periodic task, thus requiring a special treatment.
- PerformanceMonitor.Restart

  Restarts the performance monitor, resetting all of the collected data.
- PerformanceMonitor.LSUInfo

  Prints detailed information about the LSU stalls.
- PerformanceMonitor.WriteInfoFile

  Creates the performance monitor information file.

Additional commands have been implemented for the monitoring of the XOberon system scheduler.

- PerformanceMonitor.MonitorSchedulerStart

  Starts the system's scheduler monitoring. The gathered data can be retrieved with the usual commands (Info, LSUInfo and WriteInfoFile).
- PerformanceMonitor.MonitorSchedulerStop

  Stops the system's scheduler monitoring.

APPENDIX D

Profiler/Analyser Structure

This section presents the changes operated on the MacOberon Compiler, including the new modules added.

D.1 List of compiler changes by module

The following list shows the different modules, a small description, and the modifications brought to the source. The new modules introduced by this work have their name in boldface.

PPCOOPM: Low level file support, and error managing.
The support for the additional symbol file with procedure length was added.

PPCOOPS: Scanner.
The module was modified for the recognition of the new constructs: LENGTH and BOUND.

PPCOOPD: Special profiler files interface.
This module acts as an interface to the processor description file and the performance monitor data file. It provides a scanner and a parser to unserialize the information contained in the files.

PPCOOPT: Type definitions and symbol file generation.
The code for the generation of the additional symbol file with the procedure length was added to module. It also contains all the definitions of the objects needed for the data flow analysis as *basic block* and *bit vectors*.

PPCOOBV: Bit vectors.
In addition to the implementation of the bit vectors methods, the module

also contains the special memory management needed for the *use-def* chains. The disposed chain elements are stored to be recycled.

PPCOOBB: Basic blocks.
This important module, contains all the routines needed for the management and building of the additional intermediate representation. This includes computation of instruction length with the data provided by the performance monitor data file.

PPCOOPL, and PPCOOPC: Code generation.
All the generated instructions are signaled to the PPCOOBB module, so that their length can be computed and added to the corresponding basic block.

PPCOOD: Decoder.
Decodes all the instructions contained in the inline procedures, and reports them to the PPCOOBB module.

PPCOOPV, PPCOOPB, and PPCOOPP: Parser modules.
The handling of the new restrictions and constructs (BOUND and LENGTH) was added, including the generation of the appropriate parse tree nodes for the run-time bound value check. This module also reports the calling of procedures to PPCOOBB so that their length can be added to the appropriate block.

PPCOOCP: Constant propagation.
Includes all the data flow computations for the copy propagation problem.

PPCOORD: Reaching definitions.
Includes all the data flow computations for the reaching definition problem.

PPCOOLA: Loop analysis.
Contains all the automatic bounding routines including the termination condition analysis.

PPCOODAG: Directed acyclic graph management.
Contains the longest path and loop unrolling computations.

Compiler: Interface module.
Contains the support for the new compiler options and coordinates all the compile and profile phases.

Reference list

1] R. Brega.
A real-time operating system designed of predictability and run time safety.
Proceedings of The Fourth International Conference on Motion and Vibration Control (MOVIC), pp 379-384.
Institute of Robotics, ETH, Zürich 1998.

2] R. Brega, and S.J. Vestli.
A hard real-time operating system for mechatronics.
Not published, ETH Zürich, June 1998.
http://www.ifr.mavt.ethz.ch/

3] N. Wirth, and J. Gutknecht.
Project Oberon.
Addison-Wesley, 1992.

4] H. Mössenböck.
Object-Oriented Programming in Oberon-2.
Springer-Verlag, 1993.

5] Ingo Wegener.
Theoretische Informatik.
B.G. Teubner, Stuttgart, 1993.

6] M. Franz and T. Kistler.
Slim Binaries
Communications of the ACM, XL(12):87–94, 1997.

7] C.Y. Park, and A.C. Shaw.
Experiments with a program timing tool based on source-level timing schema.
Computer, Vol. 24, No. 5, IEEE, May 1991, pp. 48-57.

8] P. Puschner, and Ch. Koza.
Calculating the maximum execution time of real-time programs.
The Journal of Real-Time Systems (1):159-176, 1989.
Kluwer Academic Publishers, The Netherlands, 1989.

9] A.V. Aho, R. Sethi and J.D. Ullman.
Compilers: Principles, Techniques and Tools.
Addison-Wesley, 1983

10] E. W. Dijkstra.
A note on two problems in connection with graphs.
Numerische Mathematik, 1:269-271, 1959

11] P.G. Emma.
Understanding some simple processor-performance limits.
IBM Journal of Research & Development XLI(3), 1997.

12] IBM Microelectronics Division and Motorola Inc.
PowerPC 604/604e RISC Microprocessor User's Manual.
Motorola, 1998.
Also available in electronic form at: http://www.mot.com/SPS/PowerPC/teksupport/teklibrary/manuals/604UM.pdf

13] IBM Microelectronics Division and Motorola Inc.
PowerPC Microprocessor Family: The Programming Environments.
Motorola, 1994.
Also available in electronic form at: http://www.mot.com/SPS/PowerPC/teksupport/teklibrary/manuals/pem32b.pdf

14] F.E. Levine, and C.P. Roth.
A programmer's view of performance monitoring in the PowerPC microprocessor.
IBM Journal of Research & Development XLI(3), 1997.

15] J. Dean et al.
ProfileMe: Hardware support for instruction-level profiling on out-of-order processors.
Proceedings of Micro-30, IEEE, 1997.

16] H.J. Curnow, and B.A. Wichmann.
A synthetic benchmark.
The Computer Journal IXX(1):43-79, 1976.
Oxford University Press.

17] J.L. Hennessy, and D.A. Patterson.
Computer Architecture a Quantitative Approach.
Morgan Kaufmann, San Francisco, second edition 1996.

18] M. Honegger, A. Codourey, and E. Burdet.
Adaptive control of the Hexaglide, a 6 dof parallel manipulator.
IEEE International Conference on Robotics and Automation, Albuquerque, USA, April 1997.

D. Bertsekas, and R. Gallager.
Data Networks.
Prentice-Hall, New Jersey, second edition 1992.

G. Kackmarcik.
Optimizing PowerPC Code.
Addison-Wesley, 1995.

N.P. Jouppi and D.W. Wall.
Available instruction-level parallelism for superscalar and superpipelined machines.
Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems.
Boston, Massachusettes, April 1989

N. Wirth.
Compiler Construction.
Addison-Wesley, 1996.

Acknowledgments

I would like to thank the following persons.

Roberto Brega, my supervisor assistant, for his willingness to actively support me during my work, for his invaluable hints, and for reviewing this report's draft.

Prof. Thomas Gross, for having accepted to be my supervising professor.

Mr. Charles P. Roth, at the IBM Somerset Design Center, for the very helpful explanations and suggestions about the documentation faults and performance monitor architecture.

Mr. Michael Naef for the useful information on the DIGITAL Continuous Profiling Infrastructure Project.

My family that morally and financially supported me during these years, and allowed me to accomplish my computer science studies in Zurich.

All my friends, especially Gabriele, who carefully read the draft of this thesis, and Andrea and Luca who gave me valuable suggestions about the work.