PIPELINED PROCESSORS AND WORST CASE EXECUTION TIMES

N. Zhang
A. Burns
M. Nicholson

Department of Computer Science,
University of York, UK

ABSTRACT

The calculation of worst case execution time (WCET) is a fundamental requirement of almost all scheduling approaches for hard real-time systems. Due to their unpredicatability, hardware enhancements such as cache and pipelining are often ignored in attempts to find WCET of programs. This results in estimations that are excessively pessimistic. In this paper a simple instruction pipeline is modeled so that more accurate estimations are obtained. The model presented can be used with any schedulability analysis that allows sections of non-preemptable code to be included. Our results indicate that WCET over-estimates at basic block level can be reduced from over 20% to less than 2%, and that the over-estimates for typical structured real-time programs can be reduced by 17%-40%.

1. Introduction

In real-time systems, predictable temporal behaviour is an essential requirement. To be able to predict, and therefore guarantee, the timing behaviour of a real-time system two issues need to be addressed. First, the worst case execution time of software should be known before run time. Secondly, the system's end-to-end response time must be predicted by taking into account scheduling and communication overheads, etc. In this paper we address the issue of estimating worst case execution time, but do so within a general scheduling approach.

Although almost all methods of guaranteeing timing behaviour require knowledge of worst case execution times there has been little published work in this area (compared with, say, scheduling models). Notable exceptions are the work of Puschner within the context of the MARS project[17, 24], Park and Shaw[15, 14, 20, 16], Mok[12], Woodbury[25] Sarkar[18], and the language specific reports, e.g. Pearl[4], and Real-Time Euclid[23, 21, 22, 5]. The general approach adopted in this work is to split up the computations of a task into *basic blocks*. A basic block has the property that if the first statement in the block is executed then all statements of the block are executed. There are then two independent issues:

- what is the worst case execution time of each basic block — on a particular hardware platform;
- what is the longest (i.e. worst case) path through the basic blocks of the task.

To get the worst case execution time of each basic block, hardware enhancements such as cache and pipelining are usually ignored. The result is that estimations of execution times are overly pessimistic. Although some work has been done on predictable caches[7, 8] useful models for pipelined processors have not been reported.

This paper attempts to rectify this by presenting such a model. *Non-pessimistic analysis is possible if preemption is controlled.* We do this in the context of a scheduling approach that already takes into account intervals of non-preemption. In the next section this general scheduling approach is outlined. Section 3 then presents the mathematical model, which, by utilising hardware implementation and software timing information in a table-driven fashion, produces good execution time estimation of assembler basic blocks. A simple processor pipeline (that used

on the Intel 80C188 architecture) is analysed in detail to describe the methodology. In section 4 the contribution of this accurate model of the processor's pipelined architecture is examined with respect to other sources of pessimism such as loop constructs executing at a less-than-maximum iteration number, and alternative constructs. In Section 5 we present our conclusions.

2. Scheduling Analysis

The most common form of task based scheduling uses a preemptive priority based dispatcher on each processor[2]. At all times the task with the highest priority (of those wishing to execute on that processor) is executing. Priorities are assigned according to some scheduling theory and the temporal behaviour of the system is obtained by applying a schedulability test. For example the *rate monotonic scheme*[11] is applicable if all tasks are periodic, independent and have deadline equal to period. When deadline is less than period the *deadline monotonic scheme* is optimal[10]. Both of these schemes can be extended to cater for sporadic activities.

Task interactions present more complex problems as a static priority scheme can lead to priority inversion. All safe task interactions require some form of synchronisation; it is therefore possible for a "high" priority task to be suspended waiting for a "low" priority task to complete some computation. For example consider an inter-task communication model based on shared data areas that require mutual exclusion for safe usage. While one task is accessing a data area no other task that needs to access the same area can be allowed to do so. When a task is delayed waiting for a lower priority task it is said to be *blocked*. If it is preempted by a higher priority task it is suffering *interference*.

In order to limit the blocking time on tasks some form of priority inheritance is needed[19]. With the shared data model, each area can be given a ceiling priority that reflects the maximum priority of any task that uses it. At run-time whenever a task accesses a shared data area its priority is immediately raised to the ceiling level. As a result, blocking times are minimised (and deadlock is prevented). Moreover the tasks suffer their blocks at the beginning of their execution (i.e. immediately they are released). Once they start executing they will continue and will only experience interference.

Schedulability tests incorporate a factor for blocking (B). For example with the rate monotonic scheme task τ_k is guaranteed to meet its deadline if

$$\frac{C_1}{T_1} + \frac{C_2}{T_2} + \cdots + \frac{C_k}{T_k} + \frac{B_k}{T_k} \leq k\left(2^{\frac{1}{k}} - 1\right) \tag{2.1}$$

where C represents worst case execution time, and T the period of the task.

With this model task τ_1 has the top priority. In the more general deadline monotonic formulation the schedulability test has the following form

$$C_k + I_k + B_k \leq D_k \tag{2.2}$$

where D is the task deadline and I is a measure of higher priority tasks' interference. Various formula are available for calculating I_i (see for example Audsley et al[1]).

2.1. Non-preemptive Scheduling

The implication of introducing blocking into equations (2.1) and (2.2) is that a pure preemptive model is no longer employed. There are situations in which a "high" priority task must wait for a "low" priority task. For task τ_k to be guaranteed it must be able to cope with a block of duration B_k. This blocking can be experienced in two ways; either the task starts its execution immediately it is the highest priority runnable task (and then can suffer a block during its execution), or its

release is delayed by the blocking time (and it then suffers no further block). The run-time behaviour that delays the start of a task by a known bounded time is termed *deferred preemption*. Whilst the high priority task has its execution deferred the lower priority task is benefiting from a period of non-preempted execution.

Different scheduling schemes use priority in different ways. For example earliest deadline scheduling always runs the task with the closest deadline (i.e. priority is dynamic). In all of these schemes it is possible to allow tasks access to non-preemptable sections of code. The result of this blocking is just a reduction in the schedulability of the task set.

With pipelined processors and preemptive scheduling it is difficult to model the behaviour of the pipeline because it is not possible to predict when the pipeline needs to be flushed due to preemption. We propose that application code be composed of non-preemptable sections. These sections can then be analysed (for their worst case execution time) assuming all the benefits of the pipeline. Each task will have a maximum non-preemptable section, the duration of which is used to calculate the maximum blocking time for higher priority tasks. In general the size of non-preemptable sections will be commensurate with the blocking times each task experiences due to synchronisation blocking.

To prevent a task suffering a double block (e.g. one from deferred preemption and another from synchronisation) each synchronisation section must start a non-preemptable section. For example with the shared data model described earlier, each time a task enters the shared area (and has its priority raised to the ceiling) it will start a new non-preemptable section. Indeed if the time spent on accessing shared data is short the associated code can be contained entirely within a non-preemptable section. If this is done no further action needs to be taken to ensure mutual exclusion (i.e. priorities need not be explicitly raised as non-preemption, in effect, raises the priority to the maximum possible ceiling).

The advantage of the model presented in this paper is that it uses the knowledge that each task can suffer from blocking to produce more predictable run-time behaviour. Equations (2.1) and (2.2), for example, are known to be pessimistic as most of the time a task will not be blocked. But given that the schedulability test has assumed a block it is profitable to allow the block to happen more often and to gain some advantage.

The optimal size of the maximum non-preemptable section is inevitably application and hardware specific. As indicated above non-preemptable sections reduce schedulability; for example, in equation (2.2) B_k cannot be extended beyond the level that would cause a deadline to be missed. But as B_k increases the worst case execution time (C_k) decreases because of the advantage of the pipeline. In the experiments described below a 20% decrease in computation time was observed. The increase in blocking is unlikely to be anywhere near as large.

3. Analysing Sequential Code On Two-Stage Pipelined Processors

3.1. Processor Architecture

In order to illustrate the pipelining approach, this section describes an example two-stage pipelining processor architecture based on the Intel 80C188 processor[6].

The 80C188 is a 16-bit microprocessor, but designed to work with a 8-bit external bus. It is object code compatible with the well known 8086/8088 processor categories. Internally, the base architecture of the 80C188 has fourteen registers, which are grouped into general, segment, base/index, and status/control registers, a 16-bit ALU, a bus interface unit, as well as three programmable timers.

Similar to many other Intel processors, the 80C188 uses a two-stage pipelining architecture with a prefetch queue (or instruction queue) to improve the speed of the computer[9]. The

processor's CPU function is fulfilled by overlapping instruction opcode fetching and instruction processing. Figure 1 shows the physical architecture. The CPU contains two subprocessors, the execution unit (i.e. EU, the composite of ALU and general registers) and bus interface unit (BIU). The two units work asynchronously and concurrently.

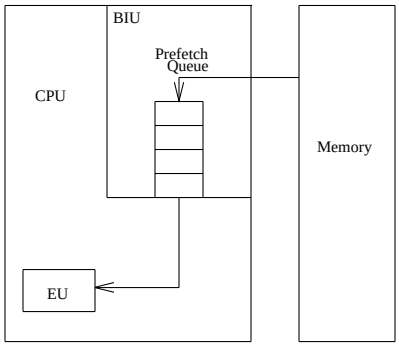


Figure 1: The block diagram of the 80C188 pipelining structure

The EU is in charge of decoding and executing all instructions, whereas the BIU does instruction opcode fetching and data accessing from external memory. The prefetcher inside the BIU fetches successive instructions, addressed by the Instruction Pointer (IP), by executing prefetch bus cycles, and places the acquired instructions in a 4-byte long prefetch queue. This operation is undertaken whenever the prefetch queue has at least 1-byte of free space or after the occurrence of a control transfer. When the EU requires a data memory access, it issues bus cycle requests to interrupt the BIU so that the EU can utilise the local bus.

By means of this design, a new instruction can be fetched by the BIU while old instructions are being executed in the EU. In other words, when the processor is ready for the next instruction, it does not need to fetch many bytes from memory since the entire instruction (or part of it) may have already been prefetched and resides in the prefetch queue. As a result, program execution speed can be improved. However, this speed enhancement is limited by the rate of instruction fetches when a series of simple instructions are executed. When long execution time instructions, which do not require memory data accesses, are executed the prefetch queue has more time to fill, and the program execution times approach closer to the CPU processing time given in the processor's manual.

3.2. The Analysis

In the following r_i is the number of data memory reads (in bytes) required by the ith instruction of some defined sequence of instructions (where $r_i \in \{0, 1, 2, ...\}$), and w_i is the number of data memory writes (in bytes), where $w_i \in \{0, 1, 2, ...\}$. For most of the instructions, the values of r_i and w_i are no more than 2 (bytes). However, they can be more than 2 if string instructions are involved.

We also calculate and present execution times in terms of CPU cycles. A simple conversion, based on the speed of the CPU, enables actual times to be generated.

3.2.1. The Assumptions

In the following, three assumptions are made in order to simplify the model:

(1) During the execution of the ith instruction which requires data memory access (i.e. $r_i + w_i \neq 0$), no instruction prefetches are conducted.

During the execution times of these instructions, the BIU may have a certain amount of time for prefetching. Whether it does or not (and if it can, the exact number of prefetches the BIU can do during the time period) varies with individual instructions. This complexity cannot be dealt with in a systematical manner, and hence the number of prefetches in these time periods is taken as zero.

(2) Instruction execution can only start when all the opcode of an instruction is fetched and resides in the prefetch queue.

(3) No opcode of an instruction is removed from the prefetch queue until the execution of the instruction is completed.

Assumptions (2-3) are made based on the consideration that although the instruction execution may start before all the opcode bytes of an instruction are placed in the prefetch queue, and similarly, partial opcode for an instruction may be removed from the queue prior to its execution completion the time saved is minimal. Hence, since it is difficult to express in a systematic way the exact timing instance when an instruction byte is actually removed from the queue with regard to various types of instructions, we again make the worst case approach to this complexity.

3.2.2. No Pipelining Consideration

The easist way to estimate the worst case timing bound for a basic block is to sum up the worst case execution times of the included instructions, regardless of the processor pipelining implementation. That is,

$$WCET_{NP}\ (basic\ \ block) = \sum_{i=1}^{N_{bb}} WCIET_{NPi} \tag{3.1}$$

where,

$WCET_{NP}$ is the worst case execution time when the processor pipelining architecture is not taken into account.

N_{bb} is the number of instructions in the sequential program block.

$WCIET_{NPi}$ is the worst case instruction execution time of the i th instruction in the stream. Its value can be obtained by the following equation:

$$WCIET_{NPi} = \begin{cases} NIFPI_i \times NCPA + E_i & (r_i + w_i = 0) \\ NIFPI_i \times NCPA + E_i + 1 & (r_i + w_i \neq 0) \wedge (r_i \times w_i = 0) \\ NIFPI_i \times NCPA + E_i + 2 & (r_i \times w_i \neq 0) \end{cases} \tag{3.2}$$

here, E_i is the execution time of the i th instruction, which excludes the fetch time of its opcode, i.e.

$$E_i = MINIET_i + (r_i + w_i) \times NCPA \qquad (CPU\ \ clock\ \ cycles) \tag{3.3}$$

also,

$NIFPI_i$ is the number of opcode fetches (i.e. the number of bytes an opcode consists of)

required by the i th instruction.

$NCPA$ is the number of CPU clock cycles required for a memory access; four is a typical value.

$MINIET_i$ is the minimum instruction computational time in the EU (in CPU clock cycles), which can be read from the processor manual. It excludes the time spent on opcode fetches and data memory accesses[6].

As can be seen from Equation (3.2), the worst case instruction execution time is composed of three elements: the time for opcode fetches, $NIFPI_i \times NCPA$; the time for instruction processing (i.e. decoding and execution), E_i; and finally the delay as the result of asynchronous handshake between the BIU and EU[6]. The equation addresses the difference in handshake delays with regard to different instruction categories. That is, the instructions, which requires no data memory access during its execution (i.e. $r_i + w_i = 0$), suffer no handshake delay, while those with one way data access (i.e. $(r_i + w_i \neq 0) \wedge (r_i \times w_i = 0)$), and those with two way data access (i.e. $r_i \times w_i \neq 0$), require, respectively, 1 and 2 additional CPU clocks above the processing time, E_i.

3.2.3. Pipelining Consideration

Now we present the new approach, which takes into account the processor pipelined architecture. We denote $WCET_P$ as the worst case execution time of a sequential assembly code stream calculated by this pipelined approach:

$$WCET_P(basic\ \ block) = \sum_{i=1}^{N_{bb}} t_i \tag{3.4}$$

where, t_i is the i th instruction worst case execution time by the pipelining analysis, which includes the necessary asynchronous handshake overheads between the BIU and EU.

Taking away handshake delays from t_i, we have b_i. That is, for $i = 1, \cdots, N_{bb}$,

$$t_i = \begin{cases} b_i & (r_i + w_i = 0) \\ b_i + 1 & (r_i \times w_i = 0) \wedge (r_i + w_i \neq 0) \\ b_i + 2 & (r_i \times w_i \neq 0) \end{cases} \tag{3.5}$$

where, $r_i \in \{0, 1, 2, ...\}$, and $w_i \in \{0, 1, 2, ...\}$.

The value b_i consists of opcode fetching time and instruction computational time. However, unlike non-pipelining analysis expressed in equations (3.1-2), function overlapping between opcode fetch and instruction computation should be considered when calculating b_i. We therefore present the following equation:

$$b_i = \begin{cases} E_i & (NIFPI_i \le f_{i-1}) \wedge \left[(r_i + w_i = 0) \vee (h_{i-1} = 0) \right] \\ E_i + (NCPA - h_{i-1}) & (NIFPI_i \le f_{i-1}) \wedge (r_i + w_i \ne 0) \wedge (h_{i-1} \ne 0) \\ E_i + (NIFPI_i - f_{i-1}) \times NCPA - h_{i-1} & (NIFPI_i > f_{i-1}) \end{cases} \quad (3.6)$$

where, f_{i-1} represents the number of opcode bytes reside in the prefetch queue at the time instant the $(i-1)$th instruction execution is completed and its opcode is removed from the queue; and h_{i-1} defines the time interval, which is shorter than that for fetching one byte, between the finishing point of the last prefetch prior to the $(i-1)$th instruction execution completion and the point where instruction $(i-1)$th finishes. Both of these values are derived below.

Figure 2 illustrates the time overlapping relationship between instruction fetching and processing (the instructions used in the figure require no data memory accesses during their executions). Note that the BIU cannot load the instructions for E_{i+1} until E_{i-1} has completed.

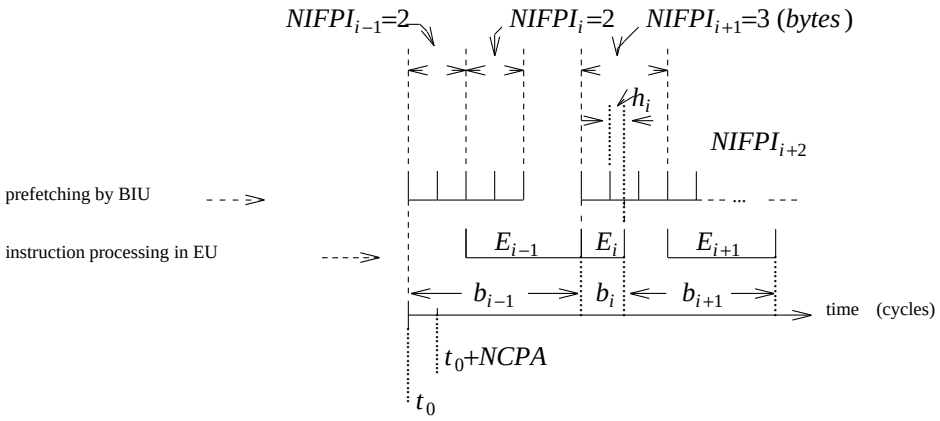


Figure 2: To illustrate pipelining operation

Before discussing equation (3.6), let us first explain the meaning of the conditions involved. Firstly, condition $NIFPI_i \le f_{i-1}$ means that the number of opcode bytes in the prefetch queue at the time when the $(i-1)$th instruction execution is completed (note: f_{i-1} excludes the $(i-1)$th instruction's opcode since it is removed upon its completion) is greater than or equal to the length of the ith instruction opcode (in bytes). This implies that, by the time the $(i-1)$th instruction execution is completed, and ith instruction is to be processed by the EU, all the opcode of the ith instruction have already been prefetched and reside in the prefetch queue. On the other hand, condition $NIFPI_i > f_{i-1}$ means that not all of the ith instruction opcode bytes have been prefetched by the time when they are needed. Secondly, h_{i-1}, as shown in Figure 3, represents the time interval, during which a prefech is being conducted but not completed, between the last prefech finishing point prior to the completion of execution of instruction $(i-1)$ and the $(i-1)th$ instruction execution finishing point. This time interval may affect the value of instruction i's execution time, depending on whether other conditions such as $(NIFPI_i \le f_{i-1})$ and $(r_i+w_i \ne 0)$ are met. We will discuss this issue further shortly.

As can be seen from the equation (3.6) the value of b_i is dependent on, firstly, whether the opcode bytes have been prefetched at the time when they are needed; secondly, whether the timing effect on the current instruction execution due to uncompleted prefetching exists; and thirdly, whether the current instruction needs data memory accesses during its execution. The first case (i.e. the first expression in the equation) says that, if the entire opcode of instruction i has already been prefetched (i.e. when the condition $NIFPI_i \le f_{i-1}$ is met) by the time the $(i-1)$th instruction execution is completed, and also if either the current instruction, i, does not require data memory accesses (i.e. $r_i + w_i = 0$), or alternatively the timing element, h_{i-1} is zero, then its execution time equals the minimum EU processing time, E_i.

However, the second case (shown by the second expression in the equation), states that if the entire opcode of instruction i has already been prefetched by the time the $(i-1)$th instruction is completed, but both instruction i requires data memory accesses during its execution (i.e. $r_i + w_i \neq 0$) and $h_{i-1} \neq 0$, then the timing element, $(NCPA - h_{i-1})$, caused by uncompleted prefetching at the time when the instruction is due for execution should be included in its execution time. Figure 3 illustrates this case (in the figure, we have assumed that $r_{i-1} + w_{i-1} = 0$ and $r_i + w_i \neq 0$).

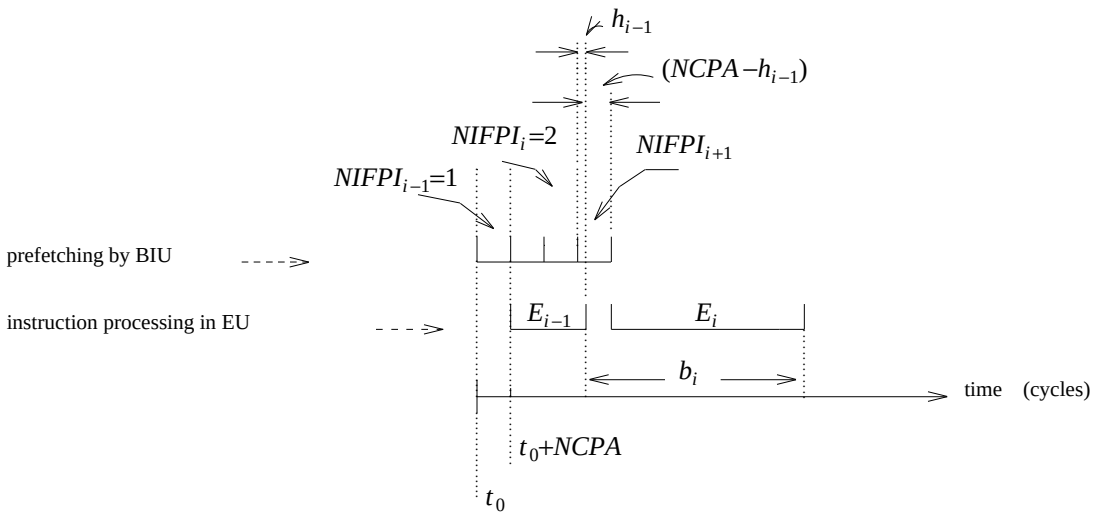


Figure 3: To illustrate the effect of h_{i-1}

As can be seen from Figure 3, when the execution of instruction $i-1$ is completed the BIU has already started fetching the $(i+1)$th instruction's opcode. Since the ith instruction's opcode is already in the prefetch queue, the execution of instruction i should proceed. However, because instruction i needs data memory access, it has to wait until the current prefetch is completed before its execution can go ahead. Thus, there is a time overhead, $(NCPA - h_{i-1})$, introduced to its execution time, b_i.

As the final case, we consider how to calculate b_i if the opcode of instruction i is not fully fetched by the time instruction $i-1$ completes its execution (i.e. when condition $(NIFPI_i > f_{i-1})$ is met). Obviously, in this case, b_i should include the time which the BIU takes to fetch the remaining opcode of instruction i after the completion of the $(i-1)$th instruction. For this, we derive the number of opcode bytes of instruction i, which are fetched before the $(i-1)$th instruction completion. As mentioned above, at the time when instruction $(i-1)$ completes its execution and its opcode bytes are removed from the queue, the remaining bytes in the prefetch queue do not include opcode bytes of instruction $(i-1)$'s predecessors. Additionally, because condition $(NIFPI_i > f_{i-1})$ is met, the opcode bytes of instruction $(i+1)$ and its successors are definitely not in the queue. As a result, the bytes in the queue are actually the prefetched opcode of instruction i,

and the number equals f_{i-1}. Thus, by referring to the third expression in equation (3.6), the second item, $(NIFPI_i - f_{i-1}) \times NCPA$ is the opcode fetching time not overlapped by $(i-1)$th instruction execution. The third item, h_{i-1} should not be included in b_i, since this time interval is overlapped by the $(i-1)$th instruction execution.

In order to calculate b_i, four intermediate variables need to be found. These are g_i, s_i, f_i, and h_i.

Firstly, g_i represents the number of opcode bytes prefetched during the time interval when instruction i is being processed in the EU, which are not subject to the limitation of a 4-byte long prefetch queue (note: the effect of 4 byte long prefetch queue is to be counted for when estimating f_i, shortly). In other words, g_i determines the amount of instruction fetching time which is overlapped by instruction processing time, E_i. Its value is related to two considerations: first, while instruction i is executing in the EU, whether or not the BIU can perform opcode prefetching, is merely dependent on whether or not instruction i requires data memory accesses during its execution; second, how many opcode bytes can be prefetched if instruction i requires no data memory access, and this is dependent on the instruction's minimum execution time (i.e. EU processing time, E_i). Based on these considerations, we have the following expression:

$$g_i = \begin{cases} 0 & (r_i + w_i \neq 0) \vee \Big((NIFPI_i \leq f_{i-1}) \wedge (h_{i-1} \neq 0) \wedge (E_i < NCPA - h_{i-1}) \Big) \\ K & (r_i + w_i = 0) \wedge (NIFPI_i \leq f_{i-1}) \wedge (h_{i-1} \neq 0) \wedge (E_i \geq NCPA - h_{i-1}) \wedge \\ & (K \times NCPA \leq E_i - (NCPA - h_{i-1}) < (K+1) \times NCPA) \\ M & (r_i + w_i = 0) \wedge \Big((NIFPI_i > f_{i-1}) \vee (h_{i-1} = 0) \Big) \wedge \\ & (M \times NCPA \leq E_i < (M+1) \times NCPA) \end{cases} \quad (3.7)$$

where, K and M are integers.

The equation above clearly indicates (as shown by the first expression in the equation) that there is no opcode prefetching during the execution of instruction i if either its execution needs data memory accesses (i.e. when $r_i + w_i \neq 0$) or its EU processing time is not long enough for a fetch to be completed. Only during the execution of those instructions which do not require data memory accesses, can the opcode fetching function be performed concurrently. The number of the prefetched bytes is then dependent on the length of the EU processing time, as expressed by the second and third expressions in the equation. The difference between the second and third expressions of g_i lies in the fact that, in the second expression, we ought to, but did not consider in g_i the one byte which is being fetched, but not completed toward the end of the execution of instruction $(i-1)$. We shall consider this one byte opcode in s_i, to be discussed next.

We define s_i as the total number of opcode bytes which could reside in the prefetch queue by the time the ith instruction execution is completed if the queue buffer space were infinite and the opcode for the ith instruction had not been removed. Clearly, s_i is the number of opcode bytes in the queue when instruction i is actually to start its execution added to the bytes which are prefetched during the course of its execution had the buffer infinite space. It is known, through the above discussion, that the latter, i.e. the bytes prefetched when instruction i is executing, is g_i. The former, however, depends on whether or not the entire opcode of instruction i has already been prefetched at the time when it is executed. If it is, i.e. the condition $NIFPI_i \leq f_{i-1}$ is met, then the value is f_{i-1}. Otherwise, if by the time instruction i is to be executed its opcode has not been completely prefetched, i.e. when the condition $NIFPI_i > f_{i-1}$ is met, then the entire opcode of instruction

i should be fetched before its execution can go ahead. In this case, the bytes accommodated in the queue prior to its actual execution are the opcode of instruction i itself, i.e. $NIFPI_i$ (since all the other opcode bytes, prefetched prior to the ith instruction execution, would have then been removed from the queue).

The factor 1 in the second expression in equation (3.8) contributes to the extra one byte which, as mentioned above, is fetched during the intervals h_{i-1} and $NCPA - h_{i-1}$. This can be seen more clearly by examining equation (3.8) in conjunction with equations (3.6-7).

$$s_i = \begin{cases} f_{i-1} + g_i & (NIFPI_i \le f_{i-1}) \wedge \\ & \left((h_{i-1} = 0) \vee (r_i + w_i = 0) \wedge (E_i < NCPA - h_{i-1}) \right) \\ f_{i-1} + g_i + 1 & (NIFPI_i \le f_{i-1}) \wedge (h_{i-1} \ne 0) \wedge \\ & \left((r_i + w_i \ne 0) \vee (E_i \ge NCPA - h_{i-1}) \right) \\ NIFPI_i + g_i & (NIFPI_i > f_{i-1}) \end{cases} \tag{3.8}$$

The value f_i is the number of bytes which can actually reside in the prefetch queue prior to the $(i+1)$th instruction execution. This value is subject to the following conditions: firstly, the prefetch queue buffer can at most accommodate 4 bytes of opcode; and secondly, the opcode of the ith instruction is removed from the queue after its completion. Thus, obviously, for $i = 1, \cdots, N_{bb} - 1$, we have

$$f_i = \begin{cases} s_i - NIFPI_i & (s_i \le 4) \\ 4 - NIFPI_i & (s_i > 4) \end{cases} \tag{3.9}$$

and,

$$f_0 = 0$$

Finally we return to h_i. This is the short time interval (i.e. $h_i < NCPA$) between the time when the last prefetch is completed (prior to the completion of the ith instruction execution) and the instant when the ith instruction execution is finished. That is:

$$h_i = \begin{cases} E_i - (NCPA - h_{i-1}) - g_i \times NCPA & (r_i + w_i = 0) \wedge (s_i < 4) \wedge (NIFPI_i \le f_{i-1}) \wedge \\ & (h_{i-1} \ne 0) \wedge (E_i \ge NCPA - h_{i-1}) \\ E_i + h_{i-1} & (r_i + w_i = 0) \wedge (s_i < 4) \wedge (NIFPI_i \le f_{i-1}) \wedge \\ & (h_{i-1} \ne 0) \wedge (E_i < NCPA - h_{i-1}) \\ E_i - g_i \times NCPA & (r_i + w_i = 0) \wedge (s_i < 4) \wedge \\ & \left((NIFPI_i > f_{i-1}) \vee (h_{i-1} = 0) \right) \\ 0 & (r_i + w_i \ne 0) \vee (s_i \ge 4) \end{cases} \tag{3.10}$$

and,

$$h_0 = 0$$

here, E_i, as mentioned earlier, is the execution time of the ith instruction, which excludes the fetch time of the opcode, and can be obtained from Eq. (3.3). All other values such as $NIFPI_i$, which is the opcode length (in bytes); r_i, the number of data memory reads; and w_i, the number of data memory writes of the ith instruction; can be obtained from the processor manual[6].

As indicated by the analysis assumptions, the above methodology for estimating the worst case execution time when processor pipelining architecture is taken into account may still produce pessimistic results in comparison with those obtained from real system measurement. But the degree of pessimism obtained from using the pipelining analysis is far less than using the simple summing up technique, and the results presented in the next section will demonstrate this point.

3.3. The Results

We now present some illustrative results that compare the execution times estimated by our theoretical evaluation and experimental measurements obtained directly from the execution of the code. Three theoretical values are presented in the following, i.e. the best case execution time (i.e. $BCET$), the worst case execution time when no pipelining architecture is considered (i.e. $WCET_{NP}$), and the worst case execution time when the two-stage pipelining structure is taken into account (i.e. $WCET_P$). The values of $BCET$ are (as indicated by the processor's manual[6]) based upon the assumption that the opcode, along with any data or displacement required for execution of a particular instruction, has been prefetched and resides in the prefetch queue at the time when it is needed, but the extra time to put data into the memory are included in the the estimation of $BCET$. However, in processor 80C188, it is noticeably likely that opcode and data bytes are not always prefetched at the time they are needed. Thus, actual program execution time will be substantially greater than the $BCET$. The $BCET$ values are of no interests to real-time system temporal analysis, and its presence here is intended as a reference to the experimental results collected.

In the following, four examples are given, for each of which the instruction code as well as the related timing information is presented in separate tables, Table 1, 2, 3, and 4. Three computed values, i.e. $BCET$, $WCET_{NP}$, and $WCET_P$, together with the experimental collected results, denoted as $Experimental\ values$, are listed for each of the examples in Table 5. To give a quantitative idea about the accuracy of the theoretical approach in which processor pipelining architecture is taken into account in comparison with the case where no pipelining is considered in the calculation, the differences with reference to real system measurements are presented. That is,

$$\begin{cases} Difference_P(\%) = \dfrac{WCET_P - Experimental\ value}{Experimental\ value} \\ Difference_{NP}(\%) = \dfrac{WCET_{NP} - Experimental\ value}{Experimental\ value} \end{cases} \tag{3.11}$$

Due to the timer resolution, the experimental value for a single measure may fluctuate 4 CPU cycles. If this is the case the average experimental value is used.

Example 1, listed in Table 1, illustrates the case in which the efficiency of instruction prefetching is only subjected to the length of instruction prefetch queue and the execution time of the next instruction. Since all the instructions in the example do not require data memory accesses during their execution, the BIU, in reality, can always perform the opcode prefetching for the next instruction as long as both the queue is not full and the computational time of the current

instruction is long enough. However, when simple instructions are executed, the BIU may not have enough time to complete the prefetching before the instructions are needed. In other words, similar to the instruction prefetching, the efficiency of the pipelining analysis is dependent upon the instructions involved as well as the program context.

Codes	$NIFPI_i$(bytes)	r_i/w_i(bytes)	$MINIET_i$(clks)
MOV dx, 003ch	3	0/0	4
ADD cl, 01h	3	0/0	4
OR dl, cl	2	0/0	3
MOV dl, 0ch	2	0/0	3
NEG cx	2	0/0	3
XOR dl, 01h	3	0/0	4
CMP dx, cx	2	0/0	3
DEC dx	2	0/0	3
INC cx	2	0/0	3
SUB dl, cl	2	0/0	3
XCHG dx, cx	2	0/0	4
TEST dl, cl	2	0/0	3
PUSH dx	1	0/0	14
POP dx	1	0/0	14

Table 1: Instruction Timing Information for Example 1

Obviously, according to Table 5, the pipelining approach provides a remarkable improvement in estimation accuracy over the conventional non-pipelining approach with reference to the real system measurement. It can be seen that the difference of the computed execution time by the pipelining analysis is 9.4%, whereas the non-pipelining analysis gives 43.8%. In other words, the pessimism in computed worst case execution time without considering processor pipelining architecture is significantly reduced by the pipelining approach, and good worst case timing bounds can therefore be obtained. The major factor which causes the residual pessimism in the pipelining analysis is that opcodes for an instruction may have been removed from the prefetch queue before the completion of the instruction execution, and thus the BIU can actually prefetch more instruction opcode bytes than the model has assumed. However, due to the complexity of extending the model to deal with these effects no further effort is made to eliminate this pessimistic factor.

Codes	$NIFPI_i$ (bytes)	r_i/w_i (bytes)	$MINIET_i$ (clks)
MOV cx, CalcuV	2	2/0	9
ADD dx, CalcuV	2	2/0	10
CMP CalcuV, dx	2	2/0	10
AND dx, CalcuV	2	2/0	10
SUB dx, CalcuV	2	2/0	10
OR dx, CalcuV	2	2/0	10
ADD TestV, dl	2	1/1	10
NEG TestV	2	1/1	10
AND TestV, dl	2	1/1	10
SUB TestV, dl	2	1/1	10
OR TestV, dl	2	1/1	10
XOR TestV, dl	2	1/1	10

Table 2: Instruction Timing Information for Example 2

Example 2, given in Table 2, illustrates a totally different case from that given in Example 1. Here, all the instructions included require data memory accesses, i.e. memory reads, memory writes, or both. The comparisons between the computed and experimentally measured values shows that, in reality, during the execution of those instructions which require data memory accesses the BIU has little chance to perform prefetching as the memory accesses require the system bus. This is confirmed by the fact that the result obtained by pipelining analysis is the same as that by non-pipelining analysis, and both give very accurate (3.5% of the differences) predictions. However, when more sophisticated instructions, such as *DIV* and *MUL*, are being executed, the BIU may have more time to fill the prefetch queue even though the instructions require data memory accesses. This is illustrated by Example 3 in which two instructions in Example 2, *MOV* and *OR*, are replaced by *DIV* and *MUL*, respectively. Experiments show that *DIV memory−byte* and *MUL memory−byte* take less time than their worst case values, i.e. for *DIV memory−byte*, measured value is 42 (clks) against 48 (clks), and for *MUL memory−byte*, measured value is 40 against 47. Taking this point into consideration, compared with Example 2, the differences between the computed and measured values have increased from 3.5% to 4.1% as the result of Assumption (1).

Codes	$NIFPI_i$ (bytes)	r_i/w_i (bytes)	$MINIET_i$ (clks)
DIV TestV	2	1/0	35
ADD dx, CalcuV	2	2/0	10
CMP CalcuV, dx	2	2/0	10
AND dx, CalcuV	2	2/0	10
SUB dx, CalcuV	2	2/0	10
MUL TestV	2	1/0	34
ADD TestV, dl	2	1/1	10
NEG TestV	2	1/1	10
AND TestV, dl	2	1/1	10
SUB TestV, dl	2	1/1	10
OR TestV, dl	2	1/1	10
XOR TestV, dl	2	1/1	10

Table 3: Instruction Timing Information for Example 3

Example 4 shows the case where both categories of instructions are included. Table 5 shows that the difference of 20.4% by non-pipelining analysis is reduced to 1.8% by the pipelining approach. This further demonstrate the accuracy and importance of the pipelining approach.

Codes	$NIFPI_i$ (bytes)	r_i/w_i (bytes)	$MINIET_i$ (clks)
ADD TestV, dl	2	1/1	10
MOV dx, 003ch	3	0/0	4
ADD cl, 01h	3	0/0	4
NEG TestV	2	1/1	10
MOV dl, 01h	2	0/0	3
AND TestV, dl	2	1/1	10
INC cx	2	0/0	3
MUL dl	2	0/0	28
SUB dx, CalcuV	2	2/0	10
PUSH dx	1	0/0	14
OR cx, CalcuV	2	2/0	10
POP dx	1	0/0	14
AND dx, CalcuV	2	2/0	10
INC dx	2	0/0	3
XCHG dx, cx	2	0/0	4
AND TestV, dl	2	1/1	10

Table 4: Instruction Timing Information for Example 4

Parameters	Example 1	Example 2	Example 3	Example 4
$BCET$ (clks)	68	143	193	163
$WCET_P$ (clks)	140	329	358†	289
$WCET_{NP}$ (clks)	184	329	358†	342
Experimental values (clks)	128	318	344	284
$Difference_P$	9.4%	3.5%	4.1%	1.8%
$Difference_{NP}$	43.8%	3.5%	4.1%	20.4%

Table 5: Theoretical and Experimental Results

4. Program Analysis

In order to examine the contribution of the above analysis to the examination of typical real-time programs we will discuss, in this section, theoretical and real system experimental results of programs which are composed of basic blocks with limited lengths, loop statements and alternative constructs. Before this discussion, we, in the following two subsections, briefly address two issues related to program timing analysis[13]: representing a program as a timing graph (TG), and reducing the program timing graph into a single node to obtain its worst case execution time. Other important issues such as restrictions on program constructs[4, 17], and types of annotations required[17], in

† The sum of instruction times actually comes to 371; inaccuracies in the manual values for the DIV and MUL instructions accounts for this reduction of 13.

order to achieve predictability of real-time software, are not addressed in detail in this paper. Higher level issues, such as how control flow in one part of the program can effect a subsequent path (or iteration rate) are also not considered. These issues are fully covered by the techniques decribed by Park[16].

Here, we assume that program constructs always have predictable timely requirements, i.e. loop constructs must have a finite upper bound on its iterations. The analysis presented below (including the processor pipeline model) have been incorporated into a software tool, ET^2 (Execution Time Estimation Tool). This tool forms part of a workbench for real-time systems engineers.

4.1. Program TG Construction

A program TG is a graph, in which the nodes represent basic blocks and edges represent program control flow. Each node in the TG contains the WCET of the corresponding basic block. Thus, performing temporal analysis for a program, basic blocks should first be identified. Basically, a basic block should be bounded by branch/jump instructions. That is, any instruction or procedure call, which leads to the update of the contents of the *IP* register with the address of an instruction other than the next one, should bound a basic block. In the following, we apply the Extended-Backus-Naur-Form (EBNF) to describe basic block definitions. That is,

(a): ::= means 'is defined as',

(b): | means 'or',

(c): ¬ means 'not',

(d): <> enclosed items.

We first define the branch instructions as:

<branch instruction> ::= *<conditional jump instruction>* | *<unconditional jump instruction>*

where,

<conditional jump instruction> ::= *je* | *jz* | *jl* | *jnge* | *jb* | *jnae* | *jle* | *jng* | *jbe* | *jna* | *jp* | *jpe* | *jo* | *js* | *jne* | *jnz* |
| *jnl* | *jge* | *jnle* | *jg* | *jnb* | *jae* | *jnbe* | *ja* | *jnp* | *jpo* | *jno* | *jns* | *jcxz* | *loop* |
loopz | *loope* | *loopnz* | *loopne*

and

<unconditional jump instruction> ::= *jmp* | *call* | *ret*

Similarly, we have

<sequential instruction> ::= ¬ *<branch instruction>*

<instruction> ::= *<branch instruction>* | *<sequential instruction>*

and

<labelled instruction> ::= *<the instruction headed by label>*

Upon the above definitions, we then further define *leaders* and *tailers* as:

<leader> ::= *the first instruction in a program module* |
<the instruction succeeded the branch instruction> | *<labelled instruction>*

<tailer> ::= *<branch instruction>*

The basic blocks are thus defined as:

<basic block> ::= *leader <sequential instruction> tailer* | *leader <sequential instruction>*

The above method illustrates that a basic block is started by a *leader* instruction, then appended to by all subsequent *sequential* instructions, and finally (or not) concluded by a *tailer* instruction. The software first identifies a set of leaders, which begin basic blocks, and then constructs a block by appending to its leader all subsequent instructions up to, but not including, the next leader.

Once basic blocks are identified, based upon the control flow of the program, a program TG can be constructed, in which the WCETs of basic blocks are obtained by the techniques (pipelining or non-pipelining approaches) described in Section 3.

4.2. Program TG Reduction

The timing behaviour of a given TG can best be estimated by reducing the TG to a minimal size, rather than examining all possible paths through the original TG. In this paper, we only address the TGs which contain no rescheduling points. That is, the graph is always reducible. Before formally describing the graph reduction rules, some notations and definitions to be used are presented[3]. In detail, we use the following notation [3]:

N is a node;

N^i is the node with name i;

N_k is the unique node with ordering index k;

The notation $N^i \longrightarrow N^j$ is used as a name for the edge from N^i to N^j as well as the relation which indicates such an edge does exist.

A path in the graph is a ordered sequence of nodes $N_i, N_{i+1}, \cdots, N_j$ such that $N_i \longrightarrow N_{i+1} \longrightarrow \cdots \longrightarrow N_j$. A *forward edge* is an edge from N_i to N_j, where $i < j$.

The graph is assumed to have one and only one *entry node* N^e, i.e. there is no N^k for which $N^k \longrightarrow N^e$.

Now, we can give the definitions of three potential assembler constructs, as follows:

(1) A **single forward path** from N_i to N_j is a unique sequence of two or more nodes $N_i, N_{i+1}, \cdots, N_{i+n}, N_j$ such that $N_i \longrightarrow N_{i+1} \longrightarrow \cdots \longrightarrow N_{i+n} \longrightarrow N_j$, where $i < i+n < j$. In other words, a single forward path contains neither alternative nor loop constructs.

(2) An **alternative construct** is a region which, originating from node N_i and joining at N_j, contains two single forward paths (i.e. $(N_i \longrightarrow N_{k'} \longrightarrow N_{k'+1} \longrightarrow \cdots \longrightarrow N_{k'+L'-1} \longrightarrow N_j) \cap (N_i \longrightarrow N_{k''} \longrightarrow N_{k''+1} \longrightarrow \cdots \longrightarrow N_{k''+L''-1} \longrightarrow N_j)$), or one forward edge and one single forward path (i.e. $(N_i \longrightarrow N_j) \cap (N_i \longrightarrow N_{k''} \longrightarrow N_{k''+1} \longrightarrow \cdots \longrightarrow N_{k''+L''-1} \longrightarrow N_j)$), where $i < (k'+L'-1) < j$, $i < (k''+L''-1) < j$, and $N_{k'+n}$ (where $n \in \{0, 1, 2, \cdots, L'-1\}$) and $N_{k''+m}$ (where $m \in \{0, 1, 2, \cdots, L''-1\}$) are the nodes covered by two different forward paths, respectively.

(3) A **loop construct** is a set of nodes, $\{N_i, N_{i+1}, \cdots, N_j\}$, where $i \le j$, such that $N_j \longrightarrow N_i$, in which N_i is a *loop head* and N_j is a *latching node.*

Now, we present the subgraph reduction rules for obtaining the WCET bounds for different program constructs[17].

(1): Single forward path subgraph reduction

For a single forward path include nodes $N_i, N_{i+1}, \cdots, N_j$, where $i < j$, we can replace the path with a new node N^{new}, i.e. N^{new} ::= $N_i \longrightarrow N_{i+1} \longrightarrow \cdots \longrightarrow N_j$. The timing bounds for the newly replaced construct can be obtained by summing up the worst case execution times of the involved nodes, respectively. That is,

$$WCET(N^{new}) = WCET(N_i) + WCET(N_{i+1}) + \cdots + WCET(N_j) \qquad (4.1)$$

(2): Alternative subgraph reduction

For an alternative construct, i.e. $(N_i \longrightarrow N^a \longrightarrow N_j) \bigcap (N_i \longrightarrow N^b \longrightarrow N_j)$, where $i < j$, we can replace the construct with a new node N^{new}, with its worst case execution time bound being to sum up the worst case execution times along the longest path (i.e. the path with the highest sum of the execution times of the single nodes). This can be expressed by equations as follows:

$$WCET(N^{new}) = WCET(N_i) + Max\left(WCET(N^a), WCET(N^b) \right) + WCET(N_j) \tag{4.2}$$

where, Max is the function for obtaining the maximum value.

(3): Loop subgraph reduction

For a loop construct, i.e. having a node N_i, the subgraph $N_i \longrightarrow N_i$ exists. Supposing the loop upper bound is *MAX.COUNT,* then we can replace the construct with a new node N^{new}. The WCET bound of N^{new} is determined by the WCET bound of N_i as well as the maximum iteration numbers of the loop. That is,

$$WCET(N^{new}) = WCET(N_i) \times MAX.COUNT \tag{4.3}$$

4.3. The results

Two assembler program macros, of typical constructs, are used to evaluate the degree of pessimism reduction by consideration of the pipelined architecture with respect to the pessimism caused by program constructs. Associated with each program example, three sub-diagrams are presented, each providing program context together with its basic block partitions and worst case execution times, the constructed TG of the program, and theoretical as well as experimental results, respectively.

In the first example (the 5th in the paper), illustrated by Figure 4, we examine the contribution of the pipelined approach with respect to the pessimism caused by loop constructs. The program, shown by the first column in Figure 4.a, contains a loop construct of maximum iteration of 15. Based on the rules of basic block partitioning, presented above, we divide the program into four basic blocks, denoted as N_1, N_2, N_3, and N_4. The WCETs of these basic blocks, shown by the second colum in Figure 4.a, are obtained by the pipelined analysis, denoted as $WCET_P$, as well as simple summing up technique (or non-pipelined analysis), denoted as $WCET_{NP}$, described in Section 3. The graph of the program is constructed and presented in Figure 4.b. Based on the graph reduction rules for calculating WCETs, presented in Section 4.2, i.e. adding up WCETs of the basic blocks along the program longest possible path, we can derive the WCETs of the given program, as follows:

$$\begin{aligned} WCET_P(Example_5) = {} & WCET_P(N_1) + (WCET_P(N_2) + WCET_P(N_3)) \times MAX.COUNT \\ & + WCET_P(N_2) + WCET_P(N_4) \end{aligned} \tag{4.4}$$

and correspondingly,

$$\begin{aligned} WCET_{NP}(Example_5) = {} & WCET_{NP}(N_1) + (WCET_{NP}(N_2) + WCET_{NP}(N_3)) \times MAX.COUNT \\ & + WCET_{NP}(N_2) + WCET_{NP}(N_4) \end{aligned} \tag{4.5}$$

note, since the program has a branch jump outside the loop, the actual iteration number for basic block N_2 is one time more than that for N_3.

For real system experiments, we varied loop iteration number and gathered three sets of actual execution times. That is, we, for the first two trials, set the loop iteration numbers less than, while the third one equal to, the maximum (i.e. $MAX.COUNT$). Comparing the experimental results with the theoretical worst case estimation, as shown in Figure 4.c, we can observe that, for this

particular example, the pipelined analysis reduces the pessimism by nearly two thirds in comparison with the non-pipelined analysis (i.e. 10% against 27%) when setting the loop iteration number equal to the maximum. When the loop iteration number is set less than the maximum during real system experiments, the degree of pessimism reduction enabled by using pipelined analysis is about one third when loop real iteration number is 10, and a quarter when it is set to 5. Obviously, a loop construct, which, at run time, iterates at a much less number than its upper bound (i.e. the maximum value expected before run time), is the main source of pessimism. But, even so, a good over-estimated reduction can be observed.

Program Context	Basic block partitions and their WCETs (cycles)
word dw 0 Example_5 macro	

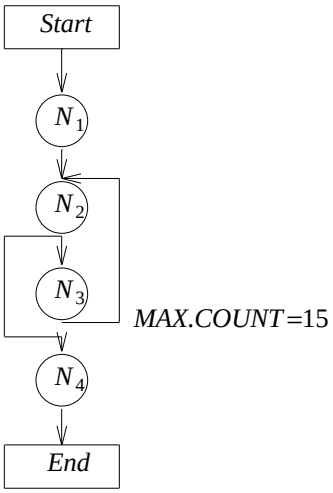
push bx
add bl, 0ah
mov bh, 20
sub dx, word
xor dx, dx
mov cx, 1ah
	$WCET_P(N_1)$=73
	$WCET_{NP}(N_1)$=95

digit:
mov ah, 01h
sbb word, 0
cmp bh, 00h
jz exit
	$WCET_P(N_2)$=69
	$WCET_{NP}(N_2)$=75

and ax, 000fh
xchg ax, dx
mul bl
mov dx, 0001h
add dx, ax
dec bh
jmp digit ;MAX.COUNT=15
	$WCET_P(N_3)$=115
	$WCET_{NP}(N_3)$=137

exit: mov ah, 02h pop bx	$WCET_P(N_4)$=26 $WCET_{NP}(N_4)$=29
endm	

(a): Program and timing information of basic blocks



(b): The graph

Parameters	trial_1 (loop bound=5)	trial_2 (loop bound=10)	trial_3 (loop bound=MAX.COUNT=15)
$WCET_P$ (*Macro*)(clks)	2928	2928	2928
$WCET_{NP}$ (*Macro*)(clks)	3379	3379	3379
Experimental values (clks)	1148	1912	2656
$Difference_P$	155%	53%	10%
$Difference_{NP}$	194%	77%	27%

(c): Theoretical and Experimental results

Figure 4: Information related to example 5

Program Context	Basic block partitions and their WCETs (cycles)
word dw 0 byte db 0 Example_5 macro	

push bx
xor bx, 0001h
mov cx, 10
cmp cx, 0
jnz next1
	$WCET_P(N_1)$=78
	$WCET_{NP}(N_1)$=94

add byte, 0
mul dl
and dx, 000fh
xchg bx, cx
mov word, dx
xor bh, bh
sub bx, 00fah
mov cx, 5
	$WCET_P(N_2)$=171
	$WCET_{NP}(N_2)$=186

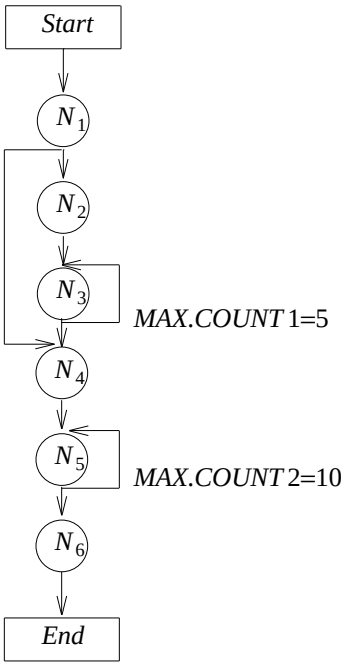
label1:
dec dx
mov dl, 02h
xchg bx, dx
sub dx, 000fh
loop label1 ;MAX.COUNT1=5
	$WCET_P(N_3)$=68
	$WCET_{NP}(N_3)$=78

next1: lea dx, byte mov cx, 10	$WCET_P(N_4)$=24 $WCET_{NP}(N_4)$=30

label2:
dec dx
mov ah, 02h
sub dx, 000fh
inc dx
mov bl, 01h
loop label2 ;MAX.COUNT2=10
	$WCET_P(N_5)$=76
	$WCET_{NP}(N_5)$=88

pop bx	$WCET_P(N_6)$=18 $WCET_{NP}(N_6)$=18
endm	

(a): Program and timing information of basic blocks



(b): The graph

Parameters	trial_1 short path	trial_2 long path
$WCET_P(Macro)$(clks)	1391	1391
$WCET_{NP}(Macro)$(clks)	1598	1598
Experimental values (clks)	918	1227
$Difference_P$	51%	13%
$Difference_{NP}$	74%	30%

(c): Theoretical and Experimental results

Figure 5: Information related to example 6

Figure 5 shows another example, in which an alternative construct is studied. Similar to the example 5, we present the program context and its basic block partitions together with their WCETs in Figure 5.a, program graph in Figure 5.b, and theoretical and experimental results in Figure 5.c. The iteration number of each of the two loop constructs is always set to, during real system experiments, its maximum, i.e. *MAX.COUNT* 1, and *MAX.COUNT* 1, respectively.

The alternative construct, as shown in Figure 5.b, leads to a fact that the program, during its execution, may take either of the two paths, $N_1 \longrightarrow N_2 \longrightarrow \{N_3 \longrightarrow N_3\} \longrightarrow N_4 \longrightarrow \{N_5 \longrightarrow N_5\} \longrightarrow N_6$, and $N_1 \longrightarrow N_4 \longrightarrow \{N_5 \longrightarrow N_5\} \longrightarrow N_6$. However, for WCET estimation, we must take the longest possible path into consideration. Thus, we can present the follow equations for the WCETs of the program by pipelined and non-pipelined approaches:

$$WCET_P(Example_6) = WCET_P(N_1) + WCET_P(N_2) + WCET_P(N_3) \times MAX.COUNT\,1 + WCET_P(N_4) + WCET_P(N_5) \times MAX.COUNT\,2 + WCET_P(N_6) \quad (4.6)$$

$$WCET_{NP}(Example_6) = WCET_{NP}(N_1) + WCET_{NP}(N_2) + WCET_{NP}(N_3) \times MAX.COUNT\,1 \quad (4.7)$$

$$+WCET_{NP}(N_4)+WCET_{NP}(N_5)\times MAX.COUNT\,2+WCET_{NP}(N_6)$$

By using the basic block timing information given in Figure 5.a, we can derive the theoretical WCETs of the program, which are presented in Figure 5.c. Real program execution times along two execution paths were both measured, and the results are also shown in the figure. The results show that, if the program takes the longer path during real system experiments, the WCET over-estimates is reduced from 30% obtained by the non-pipelined analysis to 13% by the pipelined analysis. When the shorter path is taken, the values are, correspondingly, from 74% to 51%.

The above discussions about Examples 5-6 illustrate that, although the lengths of basic blocks are limited and programs are of typical real-time software structure, the proposed pipelined analysis can still make marked contribution on reducing pessimism introduced by WCET analysis in comparison with the simple non-pipelined approach.

5. Conclusion

A mathematical model has been presented, which produces good execution time estimation for sequential assembler code by taking into account the processor pipelined architecture. The model has proved to be much more effective than the conventional instruction counting technique in that it does not ignore the function concurrency mechanism embedded in the processor hardware.

Its application is simple. By decomposing timing related information such as opcode length, memory reads/writes, and CPU execution time and utilising them in a table-driven fashion, the method is able to produce very accurate worst case timing predictions. The comparisons between analysis and real system experiments have shown that the newly proposed technique can offer much less pessimistic worst case execution time estimation than the conventional simple summing up technique.

Although the model proposed is based on a simple two-staged pipelined processor architecture, the methodology can be applied to more complex pipelined architectures. The significancy of the discussions lies in the fact that it proves that good and realistic timing bound estimation can be obtained by taking into account more system details such as hardware configuration. In addition, the study shows that other efforts, such as imposing reasonable or variable upper bounds for loop constructs at run time should also be accomplished.

References

1. N.C. Audsley, A. Burns, M.F. Richardson and A.J. Wellings, ‘‘Hard Real-Time Scheduling: The Deadline Monotonic Approach’’, *Proceedings 8th IEEE Workshop on Real-Time Operating Systems and Software*, Atlanta, GA, USA (15-17 May 1991).
2. A. Burns, ‘‘Scheduling Hard Real-Time Systems: A Review’’, *Software Engineering Journal* **6**(3), pp. 116-128 (1991).
3. C. P. Earnest, K. G. Balke and J. Anderson, ‘‘Analysis of Graphs by Ordering of Nodes’’, *Journal of ICM*, pp. 23-42 (1972).
4. W.A. Halang, ‘‘A priori Execution Time Analysis for Parallel Processes’’, *Proceedings of the Euromicro Workshop on Real-Time*, Washington, IEEE Computer Society Press (1989).
5. W.A. Halang and A.D. Stoyenko, *Constructing Predictable Real-Time Systems*, Kluwer Academic Publishers (1991).
6. Intel, *16/32 Bit Embedded Processors*, 1991.
7. D.B. Kirk, ‘‘SMART (Strategic Memory Allocation for Real-Time) Cache Design.’’, *Proceedings Real-Time Systems Symposium*, pp. 229-237 , IEEE computer society press (1989).

8. D.B. Kirk and J.K. Strosnider, ‘‘SMART (Strategic Memory Allocation for Real-Time) Cache Design Using the MIPS R3000’’, *Proceedings Real-Time Systems Symposium*, pp. 322-330, IEEE computer society press (December 5-7 1990).

9. P. M. Kogge, *The Architecture of Pipelined Computers*, McGraw-Hill (1981).

10. J.Y.T. Leung and J. Whitehead, ‘‘On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks’’, *Performance Evaluation (Netherlands)* **2**(4), pp. 237-250 (December 1982).

11. C.L. Liu and J.W. Layland, ‘‘Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment’’, *JACM* **20**(1), pp. 46-61 (1973).

12. A.K. Mok, ‘‘Evaluating Tight Execution Time Bounds of Programs by Annotations’’, *Proceedings of 6th IEEE Workshop on Real-time operating Systems and Software*, pp. 74-80 (1989).

13. D. Niehaus, ‘‘Program Representation and Translation for Predictable Real-Time Systems’’, *Proceedings of IEEE Real-Time Symposium* (1991).

14. C.Y. Park and A.C. Shaw, ‘‘A Source-Level Tool for Predicting Deterministic Execution Times of Programs’’, #89-09-12, Department of Computer Science and Engineering, University of Washington, Seattle (September 1989).

15. C.Y. Park and A.C. Shaw, ‘‘Experiments with a Program Timing Tool Based on Source-level Timing Schema’’, *Proceedings Real-Time Systems Symposium* , pp. 72-81, IEEE computer society press (December 5-7 1990).

16. C. Y. Park, ‘‘Predicting Program Execution Times by Analyzing Static and Dynamic Program Paths’’, *The International Journal of Real-Time Systems* **5(1)**, pp. 31-62 (Mar. 1993).

17. P. Puschner and C. Koza, ‘‘Calculating The Maximum Execution Time Of Real-Time Programs’’, *The Journal of Real-Time Systems* **1**(2), pp. 159-176 (September 1989).

18. V. Sarkar, ‘‘Determining Average Program Execution Times and Their Variance’’, *ACM SIGPLAN Notices (Proceedings of SIGPLAN ’89 Conference on Programming Language Design and Implementation)* **24**(7), pp. 298-312 (July 1989).

19. L. Sha, R. Rajkumar and J. P. Lehoczky, ‘‘Priority Inheritance Protocols: An Approach to Real-Time Synchronisation’’, *IEEE Transactions on Computers* **39**(9), pp. 1175-1185 (September 1990).

20. A.C. Shaw, ‘‘Reasoning About Time in Higher-level Language Software’’, *IEEE Transactions on Software Engineering* **15 (7)**, pp. 875-889 (1989).

21. A. Stoyenko, ‘‘A Real-time Language with a Schedulability Analyser’’, Phd Thesis University of Toronto (1987).

22. A.D. Stoyenko, ‘‘A Schedulability Analyzer for Real-Time Euclid’’, *Proceedings 8th IEEE Real-Time Systems Symposium*, Fairmont Hotel, San Jose, California, pp. 218-227 (1-3 December 1987).

23. A.D. Stoyenko, V.C. Hamacher and R.C. Holt, ‘‘Analyzing Hard Real-Time Programs for Guaranteed Schedulability’’, *IEEE Transactions on Software Engineering* **17**(8), pp. 737-750 (August 1991).

24. A. Vrchoticky and P. Puschner, *On The Feasibility of Response Time Predictions - An Experimental Evaluation*, PDCS Project (Esprit BRA Project 3092), Second Year Report (May 1991).

25. M.H. Woodbury, ‘‘Analysis of the Execution Time of Real-Time Tasks’’, *Proceedings IEEE Real-Time Systems Symposium*, Fairmont Hotel, New Orleans, Louisiana, pp. 89-96 (December 2-4, 1986).