Predicting Worst Case Execution Times on a Pipelined RISC Processor

Shaun J. Bharrat Kevin Jeffay
University of North Carolina at Chapel Hill
Department of Computer Science
Chapel Hill, NC 27599-3175 USA
{*bharrat,jeffay*}*@cs.unc.edu*

Abstract: A key step in analyzing and reasoning about the performance of real-time systems is the derivation of the worst case execution time of a program or program fragment. Modern computer systems with pipelined processors, caches, DMA, *etc*., can complicate this process. We demonstrate that pipelining need not be considered to be a barrier to the computation of useful worst case execution time bounds of programs by developing a simple method for accounting for the speed-up due to pipelining in an implementation of the Sparc RISC processor architecture. The method is applied to several non-trivial program fragments and is capable of accurately measuring worst case execution time even when programs are delayed by interrupt processing.

1. Introduction

The correctness of real-time systems depends on the satisfaction of both logical and temporal constraints. An important component of evaluating whether a program meets its temporal constraints is prediction of the execution time of the entire program or some component of the program. In particular, for the class of hard real-time systems where stochastic measures are unacceptable, the determination of worst case execution time is an important problem. Knowledge of worst case execution times are required to apply the feasibility tests associated with real-time scheduling algorithms and models (*e.g.*, [7, 8, 9]), to effectively use prototyping and system analysis systems (*e.g.*, [10]), and to use program logics to reason about real-time programs (*e.g.*, [2]).

Several features of current computing environments conspire to make the determination of *useful* (*i.e.*, non-overly pessimistic) worst case execution times a difficult problem. Most programs are written in a higher level language, but at the source code level, identical statements will often be translated into very different machine instructions leading to wide variability of execution time. A worst case analysis must necessarily select the worst. Even at the machine instruction level, there may be variability in the execution time of particular instructions. Hardware architectures add to the non-determinism. Virtual memory, caches,

and pipelining all compound the problem of determining tight worst case execution time bounds. Again, the worst case analysis must by definition ignore the benefits offered by each of these since, in the worst case, they may not be used. Some of the architectural sources of non-determinism, such as the effects of caching, may be eliminated by design (or re-design) as in [11]. Other sources can dealt with through careful examination of the architecture and instances of its implementation. We found this to be the case with piplining on a modern RISC processor that is an implementation of the Sparc processor architecture.

This paper examines the analysis of worst case execution time for a pipelined Sparc processor. The focus of the work is on assessing the complexity of accounting for the effects of pipelining. In particular, our study was carried out in an environment where most other sources of non-determinism in instruction execution time were absent. In our system there is no virtual memory, DMA, memory refresh, or caching. In this environment we demonstrate empirically that by using a rather simple and straightforward methodology, one can develop bounds on the worst case execution times of programs that are tight enough to be useful in practice (*e.g.*, within a few percentile of the observed worse case). Moreover, we show that the effect of activities such as certain classes of interrupt handling and other exceptional processing can be quantified and incorporated into the computation of execution times. These results argue that the use of pipelining in the implementation of an architecture such as the Sparc, in principle, need not be considered a formidable barrier to predicting the execution time of programs.

The more specific focus of this work is on the computation of worst case execution times for basic blocks — the lowest level and most processor dependent component of timing analysis. The higher-level goal, that of computing the worst case execution time of a program given timing information on basic blocks, has been considered in [3]. In this paper we use the methodology reported in [2, 3] (augmented with our techniques for computing execution times of basic blocks) to compute worst case execution times of programs. We demonstrate our method by analyzing several small programs and compare the analysis to the measured performance in a live system.

Our work is most closely related to that of Park and Shaw [3], and Zhang, Burns, and Nicholson [4]. In [2] a tool for computing execution time bounds by analysis of the (high-level) source code is presented. For a subset of the C programming language and programmer specifications of bounds on the number of iterations of each loop in the program, the tool first predicts the code to be generated and then determines the execution time bounds using that predicted code. In [4], a method is presented for computing worst

case execution time taking into account the effects of the pipeline in the Intel 80C188 processor. That processor has a rather simple pipeline consisting of just two stages, an instruction prefetch stage and an execution stage. (The execution stage is rather complex, however). They apply the model to several code fragments to quantitatively demonstrate the advantage of including the pipeline speedup.

We follow the same basic approach, however, our analysis is done at the assembler code level. While it is true that programs are indeed written in higher-level languages, determining execution times requires detailed knowledge of the implementation of the program (*i.e.*, the compiler and code generator). Park and Shaw's solution to this dilemma is to predict the code generated and then do the analysis. This is not conceptually very different from just compiling the code to assembler and then doing the analysis at the assembler level (after modifying the compiler to annotate the assembler code so that necessary markers and programmer specifications are passed through). Second, this is an analysis of code for a complex pipelined processor. This processor is representative of the current generation of RISC processors. Under certain restrictions, we believe the work will more easily generalize to other RISC processors.

The following section describes the hardware and software system used for the project. Section 3 describes our methodology in more detail. Section 4 applies the method to several example programs and validates against actual executions. Section 5 concludes and suggests extensions to the work.

2. Overview of Hardware/Software System

2.1. The Sparc Processor Architecture

The processor studied in this work is an implementation of the Sparc processor architecture. Sparc is an open RISC architecture originally introduced by Sun Microsystems. The architecture defines a basic load/store processor that is representative of the current generation of RISC processors. Since it delineates the instruction set and the instruction set alone (the instruction set implicitly prescribes certain architectural features but never their implementation), the architecture is very flexible, and a number of different chips from different producers using different technologies are currently available. Consequently, the "same" processor is used in applications with wide variability in power — the entire line of Sun 4 workstations, several compatibles, and numerous embedded processor applications all use the Sparc processor — without sacrificing binary code compatibility.

Among current RISC implementations, the use of register windows is unique and requires some explanation. The register file on the Sparc architecture is divided into a number of windows, arranged as a circular buffer, of which only one is active at any time. Each window consists of 32 logical registers which are further subdivided into four groups: eight globals, eight locals, eight “ins”, and eight “outs”. The eight global registers are shared among all windows while the eight locals are private for each window. Each window overlaps its two adjacent windows by eight registers. For window i, the eight registers shared with window $i - 1$ are called the “outs” whereas the eight registers shared with window $i + 1$ are the “ins”. The reason for the names is evident when one considers how parameter passing is achieved. The architecture provides two instruction SAVE and RESTORE which decrement and increment the current window number respectively. Consequently, to pass parameters to a procedure, the values are placed into the out registers and, as part of the callee’s procedure entry protocol, a SAVE is executed. Since the outs of the caller’s window overlaps the ins of the callee, the parameter values are now available in the “in” registers. This allows for very fast function calls since no stack operations are necessary when the number of passed parameters is less than seven (the eighth out register is used for the return address). Values can be returned in a similar fashion.

The Sparc architecture uses a delayed control transfer. When a control transfer instruction is executed, the next instruction, called the delay slot instruction, will also be executed before control is transferred to the target of the control instruction. To assist the compiler in filling the delay slot, the architecture provides an annul version for all control instructions. If one of these versions is used, and the transfer does *not* take place (because the conditional fails), then the execution of the delay slot instruction is annulled, *i.e.* not allowed to change the state of the processor. Note, however, that the delay slot instruction is *always* fetched and executed and takes at least one cycle in the execute stage.

2.2. The Network Interface Unit

The measurements we report in Section 4 were taken from a computer built at the University of North Carolina called the Network Interface Unit (NIU). The NIU is a host interface for the graphics multicomputer Pixel Planes 5 [5]. Built as part of the VISTAnet gigabit network testbed [6], the NIU is an embedded processor board consisting of a Sparc architecture processor and various subsystems for transmitting and receiving packets over a HIPPI network. The board has 1 MByte of processor main memory, all of it static RAM with a deterministic access time of exactly one clock cycle and with no refresh requirements. No cache is available, or needed, and there is of course no virtual memory. Data movement

between I/O subsystem and processor main memory is infrequent and is under total processor control: there is no DMA. Timing is achieved with an on-board counter which increments once every clock cycle; overflow of the counter triggers a clock overflow interrupt. The other features of the NIU are not applicable to the work on this project.

The NIU uses the Cypress CY7C611 implementation of the Sparc architecture [1]. This version, a 25MHZ custom CMOS implementation, is an adaptation of the more common CY7C601 for embedded processor applications. (The differences are important only at a hardware design level.) The CY7C611 uses a four stage pipeline. During the fetch stage, the instruction is fetched from memory. Since the NIU memory system is deterministic and always provides data within one clock cycle, this stage is single-cycle. During the decode stage, the instruction is decoded and the register operands are retrieved. This stage is also single-cycle. The instruction is executed during the execute stage. For most instructions (all but seven), this stage takes a single clock cycle. Some instructions, however, can take up to five clock cycles. Note that if the instruction is a load or store, the data memory access occurs in this stage. Finally, during the writeback stage, the results are written to the register file.

2.3. Compiler

Code fragments will be compared from the C source level. Using the GNU gcc compiler, version 2.2.2 running under SunOs Version 4.1.2, the code is compiled to assembler to be used in predicting execution time. Next, it is annotated with timing code and assembled using the GNU gas assembler, version 3.4, and linked with execution libraries using GNU gld, version 5.6.

3. Methodology

This section outlines a method for bounding the worst case execution time of a program. The method is straightforward: starting with a C program, a control-flow graph is constructed. The program is next compiled into Sparc assembler and worst case execution time of each basic block is computed. These time are then aggregated to give an overall bound. If interrupts are expected to occur , the overall worst case time is then adjusted to reflect the interrupt handling cost. The novel part of the method is the analysis of effects of pipelining and interrupts (in the context of a pipelined processor). The rest of the method is borrows from [2, 3].

3.1. Static, Non-Pipelined Analysis

Without suitable restrictions on program behavior, prediction of worst case execution time must ignore the potential speed-up offered by pipelining. The term "non-pipelined analysis," as used in this paper, refers to an execution time analysis that ignores *any* overlap in the execution of instructions. Clearly this approach will lead to overly pessimistic bound. However, it is necessary since the overlapping of instructions is a stochastic phenomenon and, in the worst case, the execution time for an instruction is indeed equal to its latency since the pipeline may be flushed between instructions (such as during a context-switch).

With the potential overlap ignored, the computation of worst case execution time for a block of code is greatly simplified. The worst case execution time for each instruction equals the maximum number of cycles spent in the execution stage plus three cycles, one each for instruction fetch, decode, and writeback. (Because the NIU has deterministic memory access, the instruction fetch is guaranteed to complete in a single cycle. This is not true in systems with memory caches). Since there is no interplay among adjacent instructions, the overall bound for the block is then simply the sum of the worst case execution times for each instruction.

The maximum number of cycles in the execution stage for each instruction is listed in Appendix A. Add three cycles to obtain the worst case execution time (ignoring pipelining).

3.2. Pipelined Analysis

By imposing restrictions on the program behavior, some of the effects of pipelining can be incorporated into the analysis to produce tighter bounds. To start, consider the model presented by Shaw [2]. There is a single task executing on a single processor. There are no preemptions and no interrupts (added later). Necessary ancillary processing such as the clock maintenance is done on a separate dedicated processor. Under these constraints, the state of the pipeline at any point is deterministic and, therefore, the pipeline speedup can be incorporated.

Define a *basic block* as a sequence of assembler instructions starting at the beginning of a procedure or at a label and continuing to another label or to a delay slot instruction (*i.e.* the instruction following a control transfer instruction). The worst case execution time for a basic block will be computed in three steps. First, the worst case execution time for each instruction, under the assumption of no pipeline hazards, is determined. Next, the instruction execution times are adjusted to account for each of the three possible pipeline

hazards. Finally, the adjusted individual execution times for each instruction in the block are summed.

The starting point is the worst case execution time of each instruction. Providing that there are no pipeline stalls, the instruction fetch, decode, and writeback for instruction *i* are done in parallel with the execution phase for $i - 2$, $i - 1$, and $i + 1$ respectively and they, therefore, do not contribute to the cost of instruction *i* (an exception is discussed later). Hence, the initial worst case execution time of each instruction is simply the maximum time spent in the execution stage.

For the hardware considered here, there are no additional stalls due to structural hazards. A structural hazard occurs when a new instruction cannot be issued because the necessary hardware is not available. First, we are not considering code with floating point instructions. Consequently, the most common structural stalls, those due to a non-pipelined or not fully pipelined floating-point unit, do not occur. Second, while differences in the time spent in the execution stage by separate instructions can result in structural stalls, it turns out that the cost of the stall is already counted as part of the execution cost of a prior instruction. This phenomenon is clearly shown in Figure 3. Consider two sequential instructions, instructions *i* and $i + 1$, in the instruction stream. Instruction *i* is a multi-cycle instruction with the execution phase taking 2 cycles.

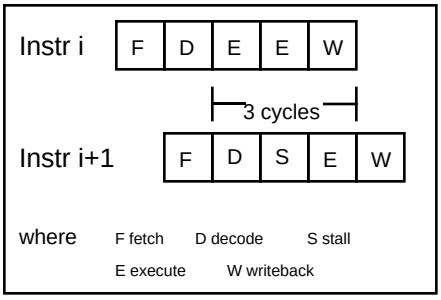

Figure 3

Clearly the aggregate execution stage time is 2+1 and not 2+1+1, *i.e.*, the cost of the stall is not counted. In any event, it is more logical to include the cost of the stall as part of the instruction *i* versus instruction $i + 1$ since it is instruction *i* that "causes" the stall.

It turns out that control hazards are not possible either. Control hazards normally occur because a transfer of control is not determined until after the succeeding instruction has been fetched. Correcting the flow then requires several stalls while the intended instruction is fetched. The Sparc architecture precludes the problem by using two techniques. First, as previously mentioned, a delayed control transfer is used. This effectively gives the

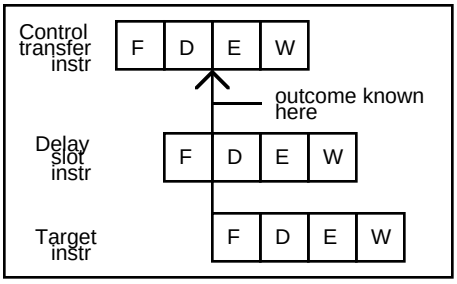

Figure 4

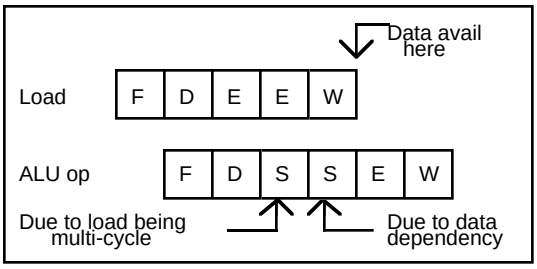

Figure 5

hardware an extra cycle of "decision" time and reduces any pipeline stall by one cycle. Second, extra hardware is used to determine both the branch target and the truth value of the condition by the end of the decode stage. The timing is shown in Figure 4. Obviously no additional control stalls are necessary.

Data hazards are possible. A data hazard results from a dependency between instructions that prevents an instruction from being issued because one or more of its operands is not yet available. The Sparc architecture uses data forwarding to eliminate all data hazards between arithmetic/logical instructions. Some stalls due to data hazards are unavoidable, however. Use of a register immediately following a load to that register must be stalled by a clock cycle. The reason can be seen in the timing diagram shown in Figure 5.

Similarly, a call instruction immediately followed by an instruction that uses register R15, or a jump-and-link instruction immediately followed by an instruction that uses the jump-and-link instruction's destination both invoke a one-cycle interlock. The conditions for stalls are summarized in Table 1.

# of Stalls	Previous instr / Current instr	Condition
1	load / any instr	Destination register of load instruction is a source operand of current instruction.
1	call / any instr	Register 15 (out register 7) is a source operand of current instruction
1	load-link / any instr	Destination register of load-and-link instruction is a source operand of current instruction

Table 1

The procedure for computing worst case execution time of an atomic block can now be delineated. For every instruction in the block, find the initial worst case execution time in

Appendix A. Next, compare the instruction and the immediately preceding instruction to see if the instruction couple meets any of the conditions outlined in Table 1. If they do, adjust the bound by the indicated number of stalls. Note that for the first instruction in the block, this requires examination of the last instruction of all blocks that can directly precede this block. The worst case is at most one stall. (A worst case of one *must* be used if all possible preceding blocks cannot be determined, such as, for example, when the current block starts a procedure.) Finally, sum these adjusted instruction bounds to form the overall bounds for the block. These bounds are in turn aggregated using the methods of Section 3.1.

There is one final overhead that must be included when analyzing an entire program. The reason is that the assumption that the fetch and decode stages of instruction i are overlapped with the execution of instructions $i - 2$ and $i - 1$ does not hold during program startup, and the assumption that the writeback stage is overlapped with the execution stage of instruction $i + 1$ does not hold during program termination. Therefore, for a pipeline with length s cycles before and t cycles after the execution stage, respectively, the entire program incurs a startup cost of s cycles and a shutdown cost of t cycles. For the Sparc architecture, this amounts to a total extra overhead of four clock cycles. Note that this penalty applies to a program as a whole; if one is interested in a code fragment rather than an entire program, this overhead does not apply.

3.3. Pipelined Analysis with Interrupts

This section describes how some interrupts may be incorporated into the analysis of worst case execution time for programs on the Sparc processor. Specifically, for a particular interrupt, if upper bounds on the exact number of said interrupt during the execution of the program are known a priori, then the cumulative cost can be incorporated into the overall bound for the program. Note that all strictly periodic interrupts, including the clock interrupt, fall into this class. Also included are most synchronous interrupts, *i.e.*, interrupts caused by execution of instructions in the program.

Some background on how the Sparc handles an interrupt is helpful. The Sparc architecture uses a vectored interrupt system and defines two broad classes of interrupts, asynchronous interrupts and synchronous interrupts or traps. The process of handling an asynchronous interrupt is described; the differences for a synchronous interrupt are mentioned later. When an asynchronous interrupt occurs, the instruction in the execution stage is allowed to complete but subsequent instructions in the pipeline (two in all) are annulled, *i.e.*, not allowed to change the state of the processor, and the pipeline drained. The level of the

recognized interrupt is then combined with the starting address of the interrupt table and this address placed into the program counter. At the second clock cycle following the interrupt recognition, the first instruction of the trap handler enters the pipeline. The timing is shown in Figure 6.

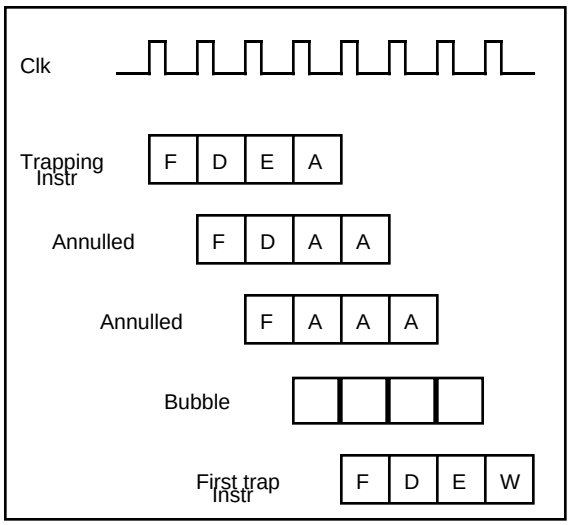

Figure 6

The process for a synchronous trap is identical except that the instruction that causes the trap is also annulled thereby requiring a restart of three instructions (versus two for the asynchronous).

In summary, the startup cost for invoking the handler is four clock cycles, and the penalty for interrupting the instruction stream is a restart of two instructions for an asynchronous interrupt and a restart of three instructions for a synchronous interrupt. The maximal worst case execution time of any instruction is four clock cycles; the two restarted instructions for the asynchronous interrupt and the three restarted instructions for the synchronous interrupt can also have one and two associated stalls respectively. Hence, the worst case interrupt overhead is 4+2*4+1=13 clock cycles for an asynchronous interrupt and 4+3*4+2=18 clock cycles for a synchronous interrupt.

The first step in incorporating interrupts into the model is determining the cost of each occurrence of the interrupt. Each cost is computed as the sum of the interrupt handling overhead and the execution time for the interrupt handler (a sequential). The interrupt overhead was described above. The handler execution time is computed by delineating the entire interrupt handler code and applying the methods of Section 3.3. The Sparc uses a vectored interrupt handler mechanism where each entry in the interrupt table has space for four instructions; if the handler is more than four instructions, the first instruction in the

entry is a branch to the real handler. In the former case, the code for the interrupt handler consists just of the code in the interrupt table, whereas in the latter, it consists of the branch (and delay slot) instruction in the interrupt table plus the code in the real handler.

The next step is introducing the cost of interrupt handling as many times as necessary. For synchronous interrupts, if an instruction *may* cause an interrupt, its cost is added to the worst case execution time. For strictly periodic interrupts, the total cost of the maximal number of interruptions must be determined and added in. Suppose that the interrupt occurs with maximum frequency F, the worst case cost of the interrupt handler is H, and that the worst case execution time, without interrupts, computed by the methods of the previous sections is T. Let T' represent the desired worst case time with interrupt handling. Then by Shaw's [2] method,

$$T' = T + \lceil T' \cdot F \rceil \cdot H < T + (T' \cdot F + 1) \cdot H$$

hence $T' < \dfrac{T + H}{1 - F \cdot H}$.

3.4. Measuring Actual Execution Times

Actual execution time of a program on a Sparc processor can be accurately measured using the NIU's on-board synchronous counter. As described before, this counter is a 16-bit counter that is incremented once every clock cycle. This counter is in memory-mapped I/O space and be read with a normal load instruction. Since all programs will have an execution time that is an exact multiple of the clock cycle time, this counter, by itself, can give the exact timing of any program execution of less than 65555 cycles. For longer program executions, the clock interrupt must be enabled so that overflows of the counter are not lost. Of course, this now entails that interrupt handling costs be incorporated into the analysis.

The cost of time-keeping must not be included in the execution time for the program under test. The normal procedure is to run the time-keeping code with a null program to determine the cost of time-keeping. This cost is then subtracted from subsequent runs. The approach is still valid but some subtleties must be understood if the measured time is to accurately reflect the true execution time of the code under test. First, the timing code cannot be inserted at the C code level. Otherwise, the compilation could change the (generated) code under test. Instead, we compile the code under test to assembler and *then* insert the timing code. Second, injudicious placement or improper design of the timing code could alter the timing characteristics of the code under test. In some cases, it could speed up the code (by

removing a stall condition) while in others it could increase the execution time (by introducing stall conditions). For each of the examples in Section 4, we ensure that the timing code does not affect the execution of the code under test.

The code for time-keeping is straightforward. Essentially, a timestamp is obtained and saved to a global variable immediately before and after the code under test is executed. After the ending timestamp is saved, a core dump trap is generated; this trap saves sufficient state for post-mortem debugging. Using a modified version of the GNU gdb debugger, the values of the timestamps can then be ascertained and the execution time of the code under test computed. For normalization, the time-keeping code with a null program takes thirteen clock cycles to execute.

3.5. Computing Worse Case Program Execution

To evaluate our method of accounting for pipelining effects, we use the techniques outlined above to compute the worst case execution time of basic blocks and then employ methodology outlined in [2, 3] for computing the worst case execution time of programs.

For example, the code generated for an if-then-else clause. The assembler code generated will have an execution subgraph of the form shown below.

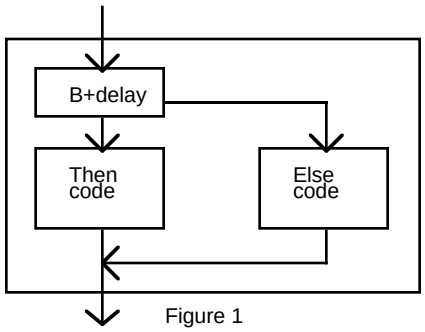

Figure 1

The worst case execution time for the composite block is simply

$$\{B + delay\} + \text{MAX}(\{then\}, \{else\})$$

where $\{x\}$ is the worse case execution time of the basic block x.[1]

As another example, consider the form of the code generated for a for-loop (assuming that loop unrolling is *not* done). Given an upper bound N on the number of executions of the

---

[1] To be fair, we are abusing the notation in [2, 3], as there the authors operate with intervals representing best and worst case execution times. Here we are only interested in the worst case execution time.

original loop (again, annotation would be helpful), then the worst case execution time of the composite block is simply

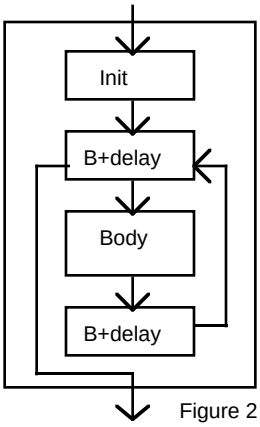

Figure 2

$$\{Init\} + (N+1)\cdot\{B + delay\} + N\cdot\{Body\} + N\cdot\{B + delay\}$$

Analysis of the forms resulting from other constructs is similar.

4. Empirical Results

This section applies the methods of the previous section to some specific code fragments written in C. The code fragments are compiled with the gnu gcc compiler to assembler. The assembler code is then analyzed to determine the worst case execution time, first ignoring the pipelining and then accounting for it. Then, timing code is added to the assembler and the result assembled, linked with the necessary libraries, and executed. The empirical values are compared with the predicted.

The code fragments selected are not meant to be representative of the code found in any particular real-time application. Instead, they were chosen to exercise the different parts of the analysis and thereby serve to test the method. Since all instructions of the Sparc architecture are covered by the method, however, any program compiled for that architecture is amenable to the analysis.

4.1. Example 1: Checksums

Consider the following fragment which computes the one's complement sum of a network packet. Such computations are common in software implementing reliable network protocols. The exact length of the packet is 100 32-bit words, and the contents of the packet are in an array called *data*.

```
sum = 0;
for ( i=0; i < 100 ++i ) {
   sum += (data[i]&0xffff);
   sum += (data[i]>>16);
}

sum = (sum>>16) + (sum&0xffff);
sum += (sum>>16);
```

The assembler code with its associated basic blocks, as defined in Section 3.1, is shown in Figure 7.

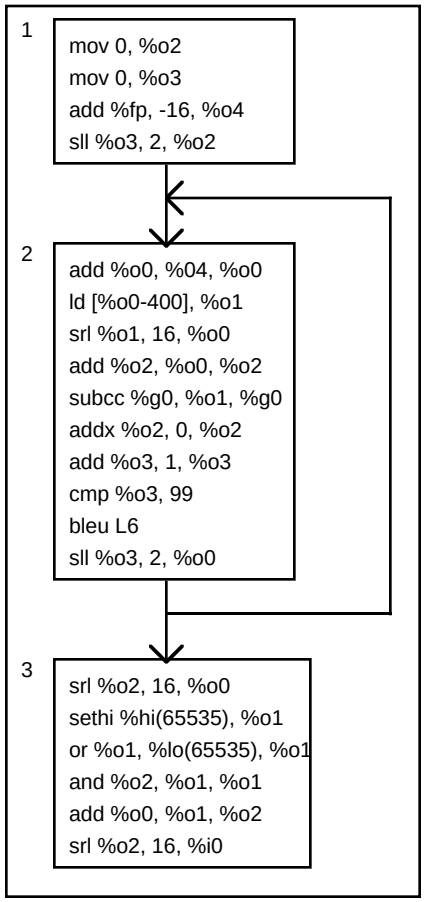

Figure 7

Two separate bounds on predicted execution time are computed: the first ignoring pipeline speedup, the second accounting for it. For the non-pipelined analysis, the first step is to compute the worst case execution time for each instruction by adding three clock cycles to the maximum execution stage times listed in Appendix A. These individual execution times

are then added to give the worst case execution time for the block. For block 1, the upper bound is 4*4 or 16 cycles; for block 2, it is 4+6+4*8 or 42 cycles; and for block three, the upper bound is 4*6 or 24 cycles. Next, these bounds for the atomic blocks are aggregated. Blocks 1 and 2 are each executed 1 time, while block 2, which forms the body and condition of the loop, executes exactly 100 times. Hence, the overall worst case execution time is 16+42*100+24 or 4240 clock cycles.

For the pipelined analysis, the first step also is to determine the worst case execution time for each atomic block. This starts with the sum of worst case execution time for each instruction assuming no pipeline stalls (from Appendix A). These initial upper bounds are 4 cycles for block 1, 12 cycles for block 2, and 6 cycles for block 3. Then, any instruction couples that match the criteria in Table 1 are adjusted by the indicated number of stalls. The ld-srl couple in block 2 matches one of the entries so the block's upper bound is adjusted to 13 cycles. Next, the total worst case execution time for each atomic block is determined, and these bounds aggregated. For this example, the total worst case block execution times are 1*4 for block 1, 100*13 for block 2, and 1*6 for block 3; the worst case execution time for the entire code fragment is then 1310. Finally, if this is an entire program being analyzed, the startup and shutdown cost must be added in. Since this is a code fragment, this step is not necessary here and the previous upper bound stands.

To empirically determine the execution time of this code fragment, the assembly language code is augmented with timing information, assembled, and executed on the NIU. A starting timestamp is saved to global variable *start*, and the ending timestamp to global variable *end*. After execution, a trap instruction forces a core dump. The values of *start* and *end* can then be determined by using a debugger on the core file and the execution time of the program under test computed as *end* – *start* – 13 (where the cost of timing code as measured with a null program is 13 cycles). The execution times for several executions as well as both sets of predicted bounds are shown in Table 2.

	Maximum
Non-pipelined analysis predicted execution times	4240
Pipelined analysis predicted execution times	1310
Actual execution times	1210

Table 2

There are several noteworthy points. First, the actual execution time is constant. This is to be expected since there is no variability in the code fragment: no if-then-else clauses and all loops have fixed bounds. Second, both predicted worst case execution times are *safe* in the sense that the actual execution time falls below the bounds, but the worst case execution time predicted by the pipelined analysis is much tighter [3]. Third, even though the code fragment has no variability, the predictions do not exactly match the actual execution time. For the non-pipelined analysis, the result is obvious since all pipeline speedup is ignored. For the pipelined analysis, this is due to the fact that a load from I/O space takes longer than a load from memory space. Since it is impossible to know, in general, to which type a particular load instruction corresponds, a load from I/O space must be assumed. If the load actually corresponded to a load from memory, as it did in this case, the predicted upper bound will necessarily be loose.

4.2 Extract maximum

This fragment extracts the maximum value from a max-heap and then reestablishes the heap property. This might be used as part of a routine for ordering tasks by priority. The example demonstrates a case where the bounds on the number of loop executions are known but the exact number depends on the state of the heap. The array *heap* contains a predefined heap, the maximum value of the heap is located at *heap*[0], and the current number of elements is stored in variable *n*.

```
.....
max = heap[0];
heap[0] = heap[n-1];
--n;

parent = 0;
lchild = 2*parent+1;
while (lchild < n) {

   rchild = lchild + 1;
   if ( rchild < n && heap[lchild] < heap[rchild] )
         bigchild = rchild;
   else
         bigchild = lchild;

   temp = heap[parent];
   heap[parent] = heap[bigchild];
   heap[bigchild] = temp;

   parent = bigchild;
   lchild = 2*parent+1;
}
.....
```

Computation of bounds on worst case execution time is straight-forward. The assembler code, organized into the atomic blocks, is shown in Figure 8. For the non-pipelined analysis, the worst case bounds are 72, 22, 12, 44, and 58 cycles for blocks 1, 2, 3, 4, and 5, respectively. Blocks 3, 4, and 5 constitute the body of the loop. Examination of the code reveals that this loop body is executed at most $\lceil \lg n \rceil - 1$ times and, therefore, the overall worst case execution time is bounded by $94 + 114 \times (\lceil \lg n \rceil - 1)$. The pipelined analysis is similar. There, the worst case bounds are 27, 7, 3, 12, and 22 cycles for blocks 1, 2, 3, 4, and 5, respectively. Hence, the overall worst case execution time is bounded by $34 + 37 \times (\lceil \lg n \rceil - 1)$.

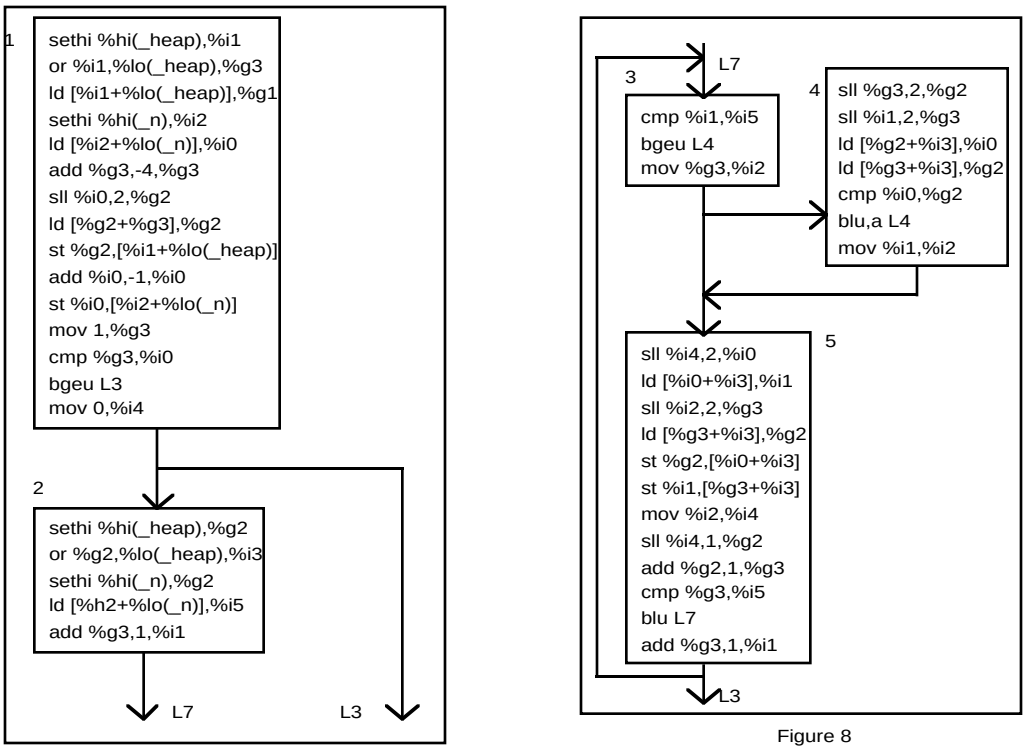

Figure 8

Table 3 summarizes the predictions and actual execution times for heaps of size 13. For the execution, the initial heap contents, in index order, was {100, 80, 90, 71, 60, 89, 70, 40, 30, 58, 21, 50, 79}. It is easy to see that this is indeed a worst case since the element swapped into the maximum position after 100 is removed must bubble all the way back down the heap. (It is important that we use a data set that gives the worst case performance. Otherwise, errors in the method of aggregating the timings of basic blocks will corrupt our measurements.)

	Maximum
Non-pipelined analysis predicted execution times	437
Pipelined analysis predicted execution times	145
Actual execution times	121

Table 3

Again, both sets of predicted bounds are *safe* but inexact. The bounds predicted by the non-pipelined analysis have 261% slack whereas the bounds by pipelined analysis have 20% slack.

4.3 Function calls

This example tests the method on code containing function calls. For the purposes of this paper, assume that the function call depth is less than seven. (Otherwise, one needs to include possibility of synchronous window overflow trap. This is doable but is rather tedious.)

```
int null_function() {

   return 0;
}

......
timeout = 0;
while (timeout < 100) {
   if (null_function())
         break;
   else
         ++timeout;
}
.....
```

The assembly code arranged into the atomic blocks is shown in Figure 9.

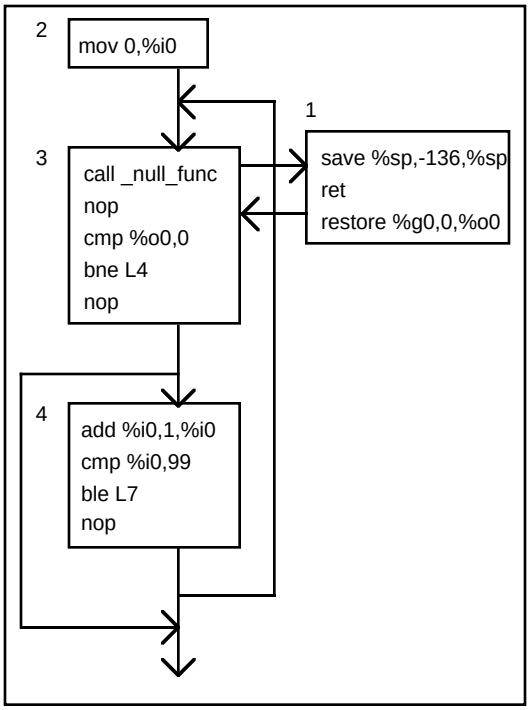

Figure 9

Analysis and measurement of actual executions produce the results in Table 4. Again, both bounds are *safe* but the worst case time obtained from the pipelined analysis is much tighter (in fact, it is exact since there is no variability and no load instructions).

	Maximum
Non-pipelined analysis predicted execution times	3604
Pipelined analysis predicted execution times	1301
Actual execution times	1301

Table 4

4.4 Accounting for interrupts

This final example demonstrates the incorporation of interrupts into the analysis of worst case execution time using the methods of Section 3.4. The specific interrupt used is the clock interrupt, a strictly periodic interrupt with an (exact) period 65555 clock cycles. Accommodating synchronous interrupts is simpler and is overviewed later. The code is almost identical to the checksumming code presented earlier. However, the packet size is now sufficiently large that clock interrupts are expected to occur.

```
sum = 0;
for ( i=0; i < 50000 ++i ) {
```

```
    sum += (data[i]&0xffff);
    sum += (data[i]>>16);
}

sum = (sum>>16) + (sum&0xffff);
sum += (sum>>16);
```

The first step is bounding the worst case execution time without interrupts. The computation is very similar to that described for the original checksum program. The overall worst case execution time is 1*8 + 50000*12 + 1*6 = 600,014 cycles.

The next step is determining the cost of handling the interrupt. The clock interrupt handler consists of two code segments: the instructions in the interrupt table branch to the actual handler, and the actual handler increments a counter. The aggregated code organized into basic blocks is shown in Figure 11, and the worst case execution time of the handler can be computed to be 12 clock cycles. Adding in the worst case overhead of 13 clock cycles for taking an asynchronous interrupt gives a total worst case cost per interrupt of 25 clock cycles.

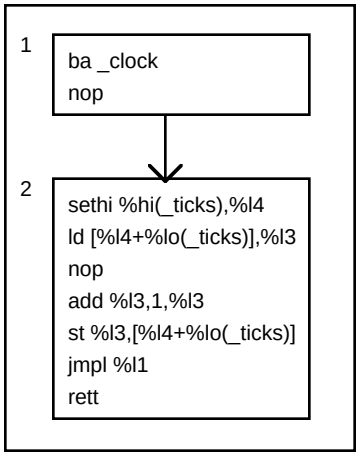

Figure 11

The final step is adjusting the worst case execution time to reflect the cost of interrupt handling. Using the dilation formula given in Section 3.3, we have

$$T' < \frac{T+H}{1-F\cdot H} = \frac{600,014+25}{1-\frac{1}{65555}\cdot 25} = 600,267 \text{ clock cycles}$$

The program was annotated with timing code and executed several times. All invocations took identically 550,182 cycles with exactly eight clock interrupts actually occurring. The fact that the number of interrupts is constant over all invocations is characteristic of this particular packet size and probably would not hold in general.

The predicted worst case time is much larger than the observed execution times considering the fact that there is no variability in the program. The problem stems not from the interrupt handling analysis, however. The analysis predicts that at most $\lceil T' \cdot F \rceil = \left\lceil 6,000,267 \cdot {}^{1}\!/_{65,555} \right\rceil = 10$ interrupts can occur. The actual value is 8 so the error due to over-counting interrupts is at most 2*25 = 50 cycles. Indeed, interrupt handling overall only adds 253 cycles to the original prediction. It turns out that the pollutant is again the load instruction. Since one cannot know in general whether it is a load from memory or from I/O space, the latter has to be assumed. This error alone causes an over-estimate of 50000 cycles. The remaining slack results from inexactness in the number of interrupts and the cost of each interrupt.

5. Conclusions and Future Work

We conclude that accounting for the speedup due to pipelining can be done and results in much tighter bounds on worst case execution time than is possible without accounting; to within a few percent of observed worst case in instances. Pipelined analysis is not without its flaws, however. There is the general problem in computing overall worst case bounds as the aggregate of the individual worst case bounds (that transcends this work). This is problematic when some of the individual worst cases are mutually exclusive, and this approach would produce overly pessimistic bounds. Pipelined analysis, as described in this paper, follows this approach and necessarily suffers the same fate. There is also a very specific problem, for the hardware architecture used in this project, with the use of loads and stores. Although most loads and stores are between the processor and memory space, it is usually impossible to statically verify this. Consequently, the larger cost of an I/O load or store must be used in the analysis. Since the number of loads and stores in a load/store architecture is understandably high, this leads to rather excessive pessimism in bounding worst case execution time, and, for this reason, an architecture with separate instructions for I/O operations would be preferable. This holds for hardware systems in general since I/O operations are usually slower than memory operations.

Several extensions to this work are possible. A reasonably simple one is to also compute the lower bounds on execution time. This would give a range of execution times which might give more insight when a program must not execute too slowly nor too quickly. A more important extension is the removal of the constraint that only a single task execute on a single processor, *i.e.*, no processor sharing. Allowing arbitrary context switching, however, is likely to render pipelined analysis ineffective since the method depends on knowing the state of the pipeline at any point in time. A reasonable middle ground is to use

deferred preemption where a task is not preempted until it has completed some base number of instructions forming a defined block. This allows some state of the pipeline to be inferred and the cost of startup is then only paid on a block basis versus on every instruction. Alternatively, if a bound on the total number of context switches is known, an approach similar to that used to introduce interrupts is possible. The time without context switching can be computed then this time dilated to include the cost of doing the context switch plus restarting instructions that were aborted. Another useful extension is a software tool to aid in the analysis. The rules described are deterministic and can be incorporated into a computer program. Ideally, the program, or set of programs, would take the source code, compile it to assembler, construct the execution graph from the assembler, bound the worst case execution time of each block, and then aggregate the worst case bounds to obtain an overall bound.

6. References

1. Sparc RISC User's Guide. ROSS Technology, Inc, Feb. 1990.
2. Alan Shaw, "Reasoning About Time in Higher-Level Language Software," *IEEE Trans. on Software Eng.*, Vol. 15, No. 7, July 1989.
3. Chang Yun Park and Alan Shaw, "Experiments With a Program Timing Tool Based on Source-Level Timing Schema," *Proceedings Real-Time Systems Symposium*, 1990, pp. 72-81.
4. N. Zhang, A. Burns, and M. Nicholson, "Pipelined Processors and Worst Case Execution Times," *Real Time Systems*, Vol. ?, No. 5, 1993, pp. 319-343.
5. R. Singh, S. Tell, S. Bharrat, D. Becker, and V. Chi, "A Programmable HIPPI Interface for a Graphics Supercomputer," *Proceedings Supercomputing*, 1993, pp. 124-32.
6. D. Stevenson and J. Rosenman, "VISTAnet Gigabit Testbed," *IEEE J. Selected Areas in Communications*, Vol. 10, No. 9, Dec. 1992, pp.1413-1420.
7. K. Jeffay, "Scheduling Sporadic Tasks with Shared Resources in Hard-Real-Time Systems," *Proc. 13th IEEE Real-Time Systems Symp.*, Phoenix, AZ, December 1992, pp. 89-99.
8. L. Sha,R. Rajkumar, J.P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," *IEEE Trans. on Computers*, Vol. 39, No. 9, (September 1990), pp. 1175-1185.
9. C.L. Liu, J.W.Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *Journal of the ACM*, Vol. 20, No. 1, (January 1973), pp. 46-61.
10. J.W.S. Liu, *et al.*, "PERTS: A Prototyping Environment for Real-Time Systems," *Proc. 14th IEEE Real-Time Systems Symp.*, Phoenix, AZ, December 1993, pp. 184-188.

11. D.B. Kirk, J.K. Strosnider, “SMART Cache Design Using the MIPS R3000,” *Proc. 11th IEEE Real-Time Systems Symp.*, Lake Buena Vista, FL, December 1990, pp. 322-330.

12. P. Puschner, Ch. Koza, “Calculating the Maximum Execution Time of Real-Time Programs,” *Real-Time Systems*, Vol. 1, No. 2 (Sept. ‘89), pp. 159-176.

Appendix A- Execution Stage Times

Name	Operation	Cycles
LDSB	Load Signed Byte (from I/O space)	2 (3)
LDSH	Load Signed Half-word (from I/O space)	2 (3)
LDUB	Load unsigned byte (from I/O space)	2 (3)
LDUH	Load unsigned halfword (from I/O space)	2 (3)
LD	Load word (from I/O space)	2 (3)
LDD	Load doubleword (from I/O space)	3 (4)
STB	Store byte (to I/O space)	3 (4)
STH	Store Halfword (to I/O space)	3 (4)
ST	Store word (to I/O space)	3 (4)
STD	Store doubleword (to I/O space)	4 (5)
ADD (ADDcc)	Add (and modify icc)	1
ADDX (ADDXcc)	Add with carry (an modify icc)	1
SUB (SUBcc)	Subtract (and modify icc)	1
SUBX (SUBXcc)	Subtract with carry (and modify icc)	1
MULScc	Multiply step and modify icc	1
AND (ANDcc)	And (and modify icc)	1
ANDN (ANDNcc)	And Not (and modify icc)	1
OR (ORcc)	Or (and modify icc)	1
ORN (ORNcc)	Or Not (and modify icc)	1
XOR (XORcc)	Exclusive Or (and modify icc)	1
XORN (XORNcc)	Exclusive Or Not (and modify icc)	1
SLL	Shift left logical	1

SRL	Shift right logical	1
SRA	Shift right arithmetic	1
SETHI	Set high 22 bits of register	1
SAVE	Save caller's window	1
RESTORE	Restore caller's window	1
Bicc	Branch on integer condition codes	1
CALL	Call	1
JMPL / RET	Jump and Link	2
RETT	Return from trap	2
Ticc	Trap on integer condition codes	1 (4 if taken)