Real-Time Systems, 7, 159–182 (1994)
© 1994 Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

A Retargetable Technique for Predicting Execution Time of Code Segments

MARION G. HARMON
Department of Computer And Information Systems, Florida A & M University, Tallahassee, FL 32307, USA.

T. P. BAKER AND DAVID B. WHALLEY
Department of Computer Science, Florida State University, Tallahassee, Fl 32306, USA

Abstract. Predicting the execution times of straight-line code sequences is a fundamental problem in the design and evaluation of hard real-time systems. The reliability of system-level timings and schedulability analysis rests on the accuracy of execution time predictions for the basic schedulable units of work. Obtaining such predictions for contemporary microprocessors is difficult. This paper presents a new technique called micro-analysis for predicting point-to-point execution times on code segments. It uses machine-description rules, similar to those that have proven useful for code generation and peephole optimization, to translate compiled object code into a sequence of very low-level (micro) instructions. The stream of micro-instructions is then analyzed for timing, via a three-level pattern matching scheme. At this low level, the effect of advanced features such as instruction caching and overlap can be taken into account. This technique is compiler and language-independent, and retargetable. This paper also describes a prototype system in which the micro-analysis technique is integrated with an existing C compiler. This system predicts the bounded execution time of statement ranges or simple (non-nested) C functions at compile time.

1. Introduction

A computer program that interacts with and reponds to real world processes in a timely fashion, and must complete execution prior to it's scheduled deadline, is called a "hard" real-time program. It is not sufficient for the implemented algorithm to be correct. The real-time program must provide the correct response (computation) on time. A late computation is usually no better, possibly even worse, than one that is on time but imprecise. The timing behavior of each real-time program component (task) must be predictable if one is to build reliable deterministic real-time systems.

Much of the research in hard real-time scheduling theory assumes that the execution time of each task is constant, and available a priori (e.g., Liu and Layland (1973), Mok (1985)). Stoyenko's work (1987) on the schedulability analyzer for Real-Time Euclid addressed the problem of worst case timing analysis of a task, by assuming the execution time of each instruction is constant. However, the hardware builders (Motorola 1985) concede that the exact execution time of a given instruction may vary, depending upon the surrounding instructions and the current state of the machine.

Recently some researchers (Niehaus 1991, Park 1993, Park and Shaw 1991, Mok 1989) have challenged this basic assumption made by much of the hard real-time scheduling theory as being unrealistic and have begun to develop tools to assist in determining more precise bounds on the execution time of programs. Mok and his students (Mok 1989) have implemented a timing tool that analyzes a stream of assembly language instructions generated from the compilation of C programs. They used a graph method to find the worst case path and computed the execution time by simulating the hardware. Park and Shaw (1989) implemented a timing tool for a subset of C, based on the notion of timing schema presented in Shaw (1987). A method very similar to that of Shaw is presented by Puschner and Koza (1989). These approaches are similar in that they all assume that the execution time of each machine instruction is constant and that the behavior of the underlying hardware is both deterministic and known. Shaw acknowledges that although his approach seems to work well when applied to simple deterministic hardware (e.g., Motorola 68010), more research is needed to determine timing predictability on more complex machines (e.g., Motorola 68020, Motorola 68030, Intel 80386, etc.).

The rest of this paper is organized as follows. First, we briefly examine several traditional methods of measuring and/or predicting execution time. Second, we present a new technique for predicting best and worst case bounds for point-to-point execution times, based on a pattern matching scheme that uses a machine description and a set of timing rules similar to those that have proven useful for code generation and peephole optimization. This new technique, which we call *micro-analysis*, is capable of taking into account the architectural characteristics of the target processor and their effect on instruction execution time. We also present results of experiments that compare the performance of micro-analysis and traditional timing methods. Finally, we describe a prototype system which integrates the micro-analysis technique with a C compiler. This prototype version predicts execution times for statement ranges or entire functions (non-nested).

2. Traditional Timing Methods

Several methods for predicting the execution time of time-critical code segments have evolved over the years. In this section a discussion of some more commonly used methods for predicting and/or measuring execution time of code segments is presented along with an examination of their strengths and weaknesses.

2.1. Table Lookup Method

The table lookup method analyzes the target code segment at the assembly language instruction level. The execution time of each individual instruction is computed by adding the time to prepare the operands to the time to perform the operation. The sum of the execution times of the instructions is considered the total execution time of the target code segment. This method will be referred to later as the table lookup method. Timing information relative to each instruction and addressing mode is usually determined by the processor's manufacturer and is printed in the user's manual or programmer's reference manual.

This approach has several disadvantages. First, some compilers do not generate assembly code (e.g., the Verdix Ada compiler version 5.5). This problem can be overcome by disassembling the object code, although it does add an extra step to the analysis process. Second, it may be difficult to match the assembly code which requires timing analysis with the corresponding high-level language code. A possible solution to this problem is to insert markers (i.e. identifiable labels) around the target source code segment, that will remain in place and be compiled through to the object code level. The main difficulty here is to insure that the markers are not repositioned or eliminated during optimization. Third, the accuracy of the timing predictions made using this approach is dependent on the accuracy of the timing information provided by the vendor. Fourth, the table lookup method performs poorly when applied to processors that implement a high degree of concurrency (e.g., instruction prefetching, pipelining, etc). For example, Intel suggests increasing performance estimates that are computed using cycle counts provided in their user's manual by 5% in order to account for occasional degradation in performance due to refilling the pipeline after a successful branch (Intel 1988). Also, operand sizes and addressing modes can influence program performance, but both are generally overlooked in discussions on execution time.

2.2. Instruction Counting

Instruction counting, like table lookup, analyzes the target code segment at the machine language level. The instruction counting method predicts execution time by multiplying the number of machine instructions in the target code segment by a pair of numeric constants which represent the average best and worst case instruction execution times for the target processor. The best and worst case numeric constants are determined by measuring the execution time of a large representative sample of machine instructions on the target processor. In addition to sharing many of the same deficiencies of the table lookup method, instruction counting does not take into account the actual timings of the individual instructions that are executed.

2.3. Software Monitors

The use of software monitors is one of the simplest timing techniques. This approach uses instructions (e.g. calls to read the system clock) which are added to the target process to gather timing data. This technique can be used to make scheduling decisions to meet timing constraints during the execution of the process. Although this technique is simple to implement, it has several disadvantages. First the actual machine is required to obtain the timing measurement. This requirement precludes measurement of execution times for software on proposed processors. In addition, the overhead involved with reading the system clock will affect the accuracy of the time measurement. Finally, software monitors can only provide rough timing measurements, depending upon the precision of the system clock (McKerrow 1988).

2.4. Dual Loop Benchmark

The dual loop benchmark paradigm is a commonly used method of measuring code execution time using a standard system clock (Altman and Wiederman 1987). The resolution of the system clock may vary from one implementation to another. However, the dual loop

```
-- context specification to define external packages
with TEXT'IO; use TEXT'IO;
package DUR'IO is new TEXT'IO.FIXED'IO(DURATION)
with DUR'IO; use DUR'IO;
with CALENDAR; use CALENDAR;

-- procedure containing dual loop
procedure DL'EXAMPLE is
  LOOP'COUNT         : constant INTEGER := VALUE;
  AVG'EXECUTION'TIME: DURATION;
  T1, T2, T3, T4: TIME;
begin
T2 := CLOCK;                  -- get start time
 for I in 1..LOOP'COUNT loop  -- benchmark loop
    TEST'CODE'SEGMENT;        -- (note it contains the
                              --  test code to be measured)
 end loop;
T1 := CLOCK;                  -- get end time

T4 := CLOCK;                  -- get start time
 for I in 1..LOOP'COUNT loop  -- control loop
     null;                    -- (note there no code
                              --  within the loop)
 end loop;
T3 := CLOCK;                  -- get end time

-- compute the average execution time
AVG'EXECUTION'TIME := (((T1 - T2) - (T3 - T4))/LOOP'COUNT);
end DL'EXAMPLE;
```

Figure 1. Dual loop timing example.

benchmark approach deals with imprecise clocks by extending the duration of the test to a length that the clock can measure. This is accomplished by inserting the test code in a loop that is sandwiched between calls to the system clock. The execution time of the test code is determined by executing the loop many times, (e.g., 100K times) and computing the average time for the benchmark loop. The overhead introduced by the loop construct distorts the measurement and must be subtracted away. This is done by measuring the execution time of a second loop that is identical to the benchmark loop without the test code (a null body). Figure 1 shows an example of the dual loop benchmark approach being used to measure the execution time of an Ada program[1].

In fact, this dual loop paradigm can be found in three commonly used Ada benchmark suites, namely the Ada Compiler Evaluation Capability (ACEC) test suite (Hook, Riccardi, and Vilot 1986), the Performance Issues Working Group (PIWG) test suite developed by a working group of the Association for Computing Machinery's Special Interest Group for Ada (SIGADA) (Roy 1990), and the University of Michigan test suite (Clapp et al. 1986).

A major weakness of the dual loop benchmark method is that it assumes that textually equivalent code constructs require the same amount of time to execute. In particular, the time required to execute the loop constructs of the control loop and the benchmark loop may not be same. Dual loop benchmarking requires adding new code to the code segment to be timed (i.e. the loop, increment, test, and bound). Removing this code after the measurements are taken can change instruction alignment and execution time. The misalignment of word and long-word operands can cause multiple bus cycles to be performed for the operand transfer. Altman and Weiderman (1987) showed that identical loops exhibited substantial variations in execution time (as much as 12 percent) on specific test systems. For example, if the control loop fits into cache but the benchmark loop does not, then the control loop will execute faster than the benchmark loop. This variation in the execution time of the control loop and benchmark loop will cause the time calculation to be erroneous.

Another problem with the dual loop approach is that the application copy of the timed routine and the test copy may yield different executions times, due to differences in cache and alignment (instructions and data). Timing a code segment in isolation requires a specially constructed test harness in which to make the measurement. The constructed dual loop test with its supporting code will be different from the actual application environment. Also data dependences and optimization issues apply here as well. For instance, an optimizing compiler may remove instructions from the control loop, making it necessary to write additional code to suppress the effects of optimization. It is unlikely that the same context would exist in the actual application.

2.5. Simulation

Using software to simulate the target processor is another common approach to determining the timing behavior of a code segment (Arnold 1987). For instance, the General Code Analyzer component which is part of the SARTOR (Software Automation for Real-Time OpeRations) environment implements a general hardware simulator (Mok 1985). Simulators are complex, require much effort to construct, and are also very slow. Simulators are only capable of measuring the execution time of a single test case, other test cases may produce different execution times. The accuracy of the measurement depends on two important factors: how well the simulator models the execution algorithm of the target processor and the accuracy of the timing data used by the simulator.

The implementation of an accurate simulator requires very detailed and precise information about the internal functions of the target processor; information that is usually proprietary. This is still a common method of predicting processor performance. Software developers who need accurate timing can purchase simulators from the processor manufacturer if necessary.

2.6. *Direct Measurement*

A logic analyzer, also called an oscilloscope, may be used to measure the execution time of a code segment. A logic analyzer is a hardware device that uses electrical probes to monitor processor activity. A logic analyzer is particularly useful when looking at time relationships of data on a bus (e.g., a microprocessor address, data, or control bus). Most logic analyzers are two analyzers in one. The first part is the timing analyzer and the second is a state analyzer. The state analyzer captures all state information between trigger points, while the timing analyzer computes the elapsed time between states and trigger points. Hardware timing tools are usually more accurate than the other timing methods discussed here, however they also tend to be expensive in several respects.

There are several disadvantages to using hardware instruments to measure software timing. First, the timing instrument itself is expensive to purchase. Second, it requires a skilled individual to perform the measurements. One must have a working knowledge of the instrument as well as the software to be timed. Third, this method requires that the target processor be available, because the timing data is measured directly on the processor. It is not uncommon to develop software systems to run on hardware that has not yet been built, in order to shorten the total system development cycle. Fourth, like simulation this method produces a single execution time measurement for the code segment in question; that measurement is accurate only for the data and processor state present when the measurement is taken. Variables that influence timing, like data size, cache contents, operand and instruction alignment, wait states, interrupts and virtual memory will cause the execution time of a code segment to vary from one execution to another when it is executed within the context of a complete application. Some of these variables (e.g., cache contents) will become stable, once the code segment starts up and runs for a while, and so are not a major problem for the direct measurement approach. To acquire accurate timings using a logic analyzer the timing technician must take several measurements under various processing conditions. Still, there is a possibility of missing the test case that would cause the code segment to execute longer than the maximum measurement or less than the minimum measurement observed.

3. The Micro-analysis Technique

The micro-analysis approach was influenced by results produced in the area of compiler design, most notably work by Davidson and Fraser (1984a) involving retargetable peephole optimizers. It is based on the concept of using a machine description, in the form of a set of translation rules, to translate compiler object code into a sequence of very low level instructions. The stream of micro-instructions is then analyzed for timing via a multi-level pattern matching scheme. At this level, the effects of advanced features such as caching and instruction overlap can be taken into account. Micro-analysis is a three step process. The three steps are:

1. Compile the program and disassemble the object module. The tool suite includes a

`ftch`	`instruction fetch`
`=%p`	`move program counter to Memory Address Register (MAR)`
`=%s`	`move source address to MAR`
`=%d`	`move destination address to MAR`
`+p`	`increment program counter`
`=sM`	`memory read to update source register`
`=dM`	`memory read to update destination register`
`=Ms`	`memory write operation`
`+sd`	`add source to destination`
`*sd`	`multiply source by destination`
`NZVCX`	`set condition codes`

Figure 2. A subset of the machine description language.

retargetable disassembler generator (Oh 1989) that was used to build disassemblers for the Motorola MC68020 and Intel 80386.

2. Transform machine instructions into a sequence of primitive operations which express the functionality of each machine instruction in fine-grain detail. The transformations are performed by a parser that uses the machine description of the target processor to produce a sequence of primitive operations. This process is called *micro-translation.*

3. Scan the stream of primitive operations, identifying patterns and applying rules which either specify a replacement pattern or an execution time. The execution times are specified as integers which represent the number of clock cycles required for a pattern of primitive operations to execute. When the analysis is complete, the execution time of the target code segment is displayed as a bounded integer time interval (i.e. [best case, worst case]).

The parser (i.e. micro-translator) is constructed using a parser generator developed by Baker (1982). This parser generator was modified so that it would produce a parser capable of emitting primitive operations as it parsed the assembly language instructions of a code segment. The disassembler generator, parser generator and execution time analyzer were used to construct a timing tool for the MVME133A-20 single-board computer (Motorola MC68020 processor) and a Mitsuba personal computer (Intel 80386 processor). The tool has been used to predict the execution time of programs written in Ada, C, and assembly language.

3.1. Machine Description Language

The machine description language (MDL) is set of syntactic patterns. Each pattern denotes a specific micro-level operation. Combination of these patterns denote specific processing events that occur during the execution of associated machine instructions. Figure 2 lists a subset of the patterns which make up the micro-language. The letters `s` and `d` denote internal source and destination registers not accessible by the user. Our model assumes that all computations take place in these registers.

4. The Timing Tool

The timing tool predicts a best case and worst case execution time of code segments (i.e. point-to-point execution time). The tool is composed of three independent retargetable components that correspond to the three steps in the micro-analysis process:

- Disassembler: a program that disassembles object code and produces assembly level instructions.
- Parser: a program that transforms machine instructions into a sequence of primitive operations which express the functionality of each instruction in fine grain detail. The parser is driven by a machine description specified in the form of an attributed grammar.
- Timer: a rule-driven pattern matching program that evaluates sequences of primitive operations to determine execution time.

The tool also includes an interactive user interface which prompts the user to input information that is generally undecidable, such as the minimum and maximum number of loop iterations, and the beginning and ending point of the code segment to be analyzed. The timer component is designed to take into account the specific architectural features of the target processor through its parameterized interface. For instance, the current version of the tool predicts timing for code segments executed on the 68020 and 80386 processors. They both handle the processor features that influence instruction timing, such as memory speed, cache memory size (68020 version), memory refresh, pipelining, etc. Figure 3 shows the organization of the tool.

4.1. Timing Rules

The timing analysis is guided by a set of pattern-driven timing rules. A timing rule consists of a left-hand side (LHS) and a right-hand side (RHS). The LHS is always a pattern consisting of one or more primitive operations. The RHS is either a macro-level pattern or a time interval. Macro-level patterns blend the execution time of two or more adjacent or near adjacent primitives (operations). These macro-level patterns perform a type of "time folding" on operations whose execution may overlap.

The example processor that will be used to illustrate the micro-analysis technique is the Motorola MC68020. Figure 4 shows the pipeline structure of the MC68020 (Motorola

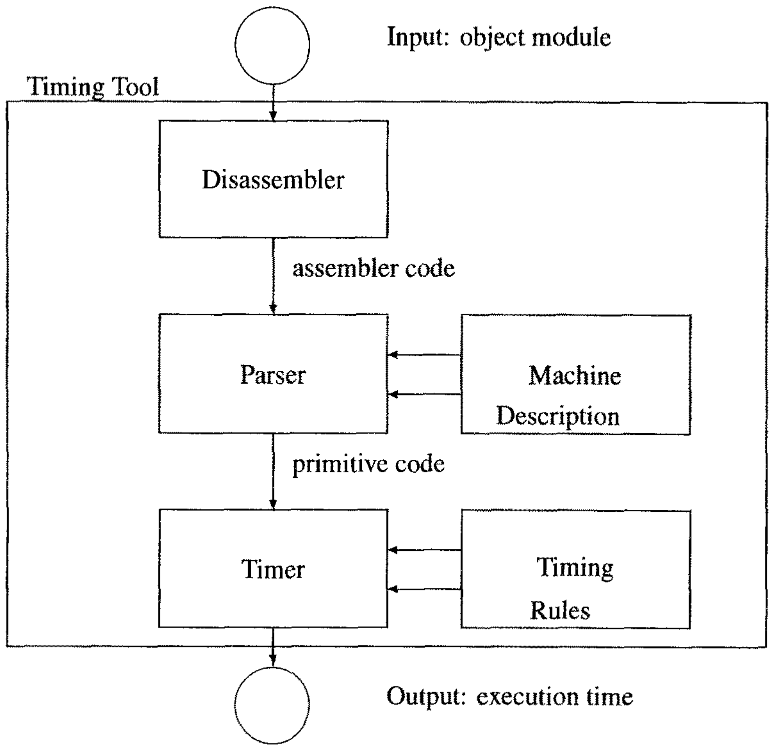

Figure 3. Overview of tool components.

1985). The processor contains a three stage pipeline that allows up to three operations to be performed simultaneously.

For example, the Motorola MC68020 processor always reads a long word (32 bits), thus providing an opportunity for overlap during the instruction fetch cycle. Several of the instructions in the MC68020 instruction set require only one word (16 bits) of memory, making it possible to fetch two adjacent single word instructions at the same time. The second instruction is effectively loaded in zero clock cycles. The possibility of fetching multiple instructions concurrently is handled by the timing rules for fetching instructions.

Five of the more than 125 timing analysis rules for the MC68020 are listed below. Note that PGR denotes a half word read operation and FPR denotes a long word read operation.

1. `ftch` → `PGR`
2. `=%p =sM` → `PGR`
3. `=%p =dM` → `PGR`

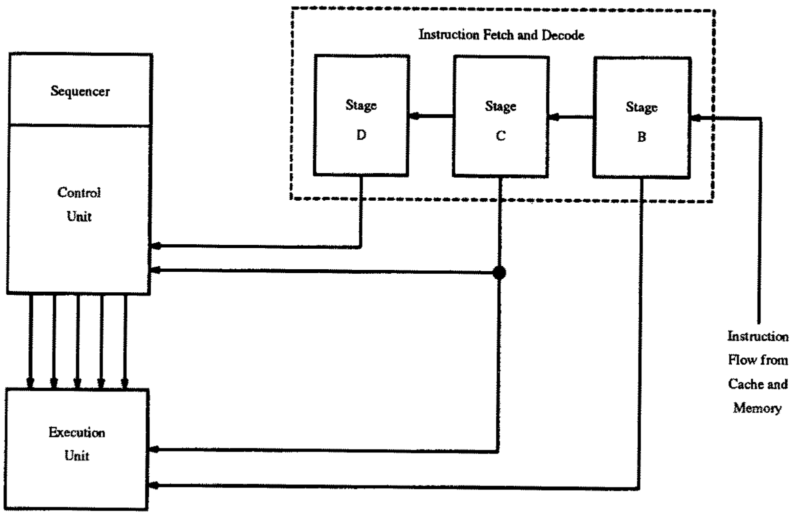
Instruction Fetch and Decode
Sequencer
Control Unit
Stage D
Stage C
Stage B
Execution Unit
Instruction Flow from Cache and Memory

Figure 4. Motorola MC68020 pipeline structure.

4. PGR PGR → FPR

5. FPR → [bc, wc]

Rule 4 instructs the timer to replace two consecutive occurrences of the pattern PGR with the pattern FPR, which denotes a long word read operation. This replacement indicates that two words from the instruction stream were loaded during one fetch cycle, so the second word is loaded for free. Rule 1 replaces each occurrence of `ftch` with PGR in the stream of macro-level patterns. Rules 2 and 3 support the loading of extension words during an instruction fetch cycle. The number of extension words fetched may increase the number of bus cycles required to load the entire instruction and therefore must be considered in the analysis. Rule 5 is one of several rules that replace patterns with bounded execution times. The purpose of this rule is to replace the pattern FPR with the predicted bounded execution required to complete the fetch operation. These bounded execution times accumulate during the Micro-analysis process to compute a bounded execution time for the target code segment.

5. A Simple Timing Analysis Example

A simple Ada procedure is used to illustrate the use of our tool. This simple Ada procedure in Figure 5 was compiled using the VERDIX 5.7 cross compiler. Figure 6 shows the

```
  --A simple Ada procedure
1 procedure EXAMPLE is
2  X, Y: integer:= 5;    -- X and Y are integers
  begin                 -- initialized to 5
3  for i in 1..100 loop -- i iterates from 1 to 100
4    X := X + Y;        -- loop body thats
5    Y := i + X;        -- disassembled in Figure 6
6  end loop;
7 end EXAMPLE;
```

Figure 5. Line numbered Ada procedure.

```
30: move.l a6@(-08), d0          # move value at [a6 - 8] to d0
34: add.l  a6@(-0c), d0          # add value at [a6 - 12] to d0
38: trapv                        # check for overflow
3a: move.l d0, a6@(-08)          # store d0 at [a6 - 8]
3e: move.l a6@(-010), a6@(-0c)   # move value at [a6 - 16] to
                                 # [a6 - 12]
44: add.l  d0, a6@(-0c)          # add value in d0 to [a6 - 12]
48: trapv                        # check for overflow
```

Figure 6. Disassembled code for loop body in Figure 5.

disassembled object code for this procedure. The source level code is annotated with line numbers that are used by the user to specify which code segments to time. The assembly code corresponding to the source level statements specified by the user are passed on to the parser, which transforms each assembly language instruction into a sequence of primitive operations (Figure 7) that express the functionality of the corresponding machine instruction(s) in fine grain detail.

The timer component uses a set of timing rules which incorporate the architectural features and execution paradigm of the target processor. The RHS may be another (higher level) pattern, or a time. For example, the MC68020 always reads a long word (32 bits), thus providing an opportunity for overlap during the instruction prefetch cycle. The rule to handle prefetching is as follows: PGR & PGR $\rightarrow$ FPR. This rule will replace a pattern of fetch primitives with the higher level pattern FPR which denotes a program read operation. After applying the rules to the stream of primitives in a systematic manner until it converges, the timer predicts a bounded execution time for the code segment.

To illustrate this technique we will compute the execution time of the two instructions located at addresses 30 and 34 shown in Figure instruction number 4 in the annotated source

```
30: ftch =%p =sM =+sa<6> =%s =sM +p =<d0>s NZVC
34: ftch =%p =sM =+sa<6> =%s =sM +p =dd<0> +sd NZVCX =<d0>s
38: ftch trappv
3a: ftch =sd<0> =%p =dM +da<6> =%d =Ms NZVC
3e: ftch =%p =sM =+sa<6> =%s =sM +p =%p =dM +da<6> =%d =Ms NZVC
44: ftch =sd<0> =%p =dM +da<6> =dM +sd NZVCX =%d =Ms
48: ftch trappv
```

Figure 7. Fine-grain primitives for lines 4–5 in Figure 5.

listing in Figure instructions were aligned in a linear format. Figure 8 illustrates how the micro-analysis technique predicts a bounded execution time for these two instructions.

The pattern matcher identifies a sequence of micro-patterns and replaces it with another pattern, a token which denotes the operation initiated by the sequence of micro-patterns. These higher-level replacement patterns also form a linear sequence which undergo further analysis by other timing rules which detect instances of execution overlap. For example, the first pattern found in this example is a program read operation and it is replaced by the token PGR. The instruction generating this program read operation has one extension word as indicated by the =sM micro-pattern. Recall that the MC68020 always reads a long word (32-bits) and PGR denotes a long word instruction fetch. Next an address translation pattern is found, and it is replaced by the ADT token. The next pattern is found by the pattern matchers lookahead feature. Note that the pattern matcher must skip forward to locate the next program read operation. This is necessary because the target processor supports instruction prefetching. Notice also that the third pattern was generated by the instruction at address 34.

The timing rules are applied to the higher-level patterns, to identify instances of overlap and to compute the execution time of the code segment. For instance, the first high-level pattern in Figure since the pipeline is considered empty at startup time. The second and third macro-patterns could execute in parallel since the address translation and the bus operation needed to prefetch the next instruction can be performed currently. Column B specifies the replacement pattern for a given sequence of low-level primitive patterns. Column C indicates the patterns that execute in parallel, for example the pattern in row 4 column B, which denotes a data read operation (DAR) executes in parallel with patterns 5, 6 and 7, which denote a register assignment REG, set condition code SCC, and address translation ADT respectively. Patterns 8 and 9 are identified as a program read operation, but more significantly this program read occurs before the data read, this is because the instruction look ahead (prefetch) operation has higher priority than data read and data write operations. Note also that the prefetch instructions are not part of the target code segment being timed, however the time consumed by this program read must be computed into the time required to execute the two instructions being timed. Columns D and E show the bounded execution time computed at each step in the analysis and column F shows the predicted overlap for operations that execute in parallel.

	A	B	C	D	E	F
	Micro Patterns	Macro Pattern	Overlapped Patterns	Min Cycles	Max Cycles	Over-lap
1)	ftch =%p =sM +p	PGR	1	2	5	0
2)	=+sa<6> =%s	ADT	2,3	4	10	2
3)	ftch =%p =sM +p	PGR				
4)	=sM	DAR	4-7	9	16	3
5)	=<d0>s	REG				
6)	NZVC	SCC				
7)	+sa<6> =%s	ADT				
8)	ftch	PGR	8,9	11	21	5
9)	ftch					
10)	=sM	DAR	10,11	16	26	1
11)	=dd<0>	REG				
12)	+sd	ADD	12	18	27	0
13)	=<d0>s	REG	13	19	28	0
14)	NZVC	SCC	14	20	29	0

Figure 8. An example of micro-analysis.

Figure 9 graphically depicts the worst case execution of the two instructions described in Figure 8. The instruction overlap was determined by conducting timing experiments aided by the logic analyzer and using instruction execution information provided by the hardware reference manual (Motorola 1985).

In this illustration of the micro-analysis technique, the timing rules performed the analysis under the assumption that cache is enabled and that all data operands are on even word boundaries. The rules can be altered to reflect the characteristics of the target processor, through a parameterized interface.

Table 1 compares the execution times computed for source statements 4 thru 5 in Figure 5. For this example our tool predicted a best case time of 39 clock cycles and a worst case time of 62 clock cycles. This time bound is very close to that measured under the same assumptions on our logic analyzer. In particular, the logic analyzer predicted a best case time of 37 clock cycles and a worst case of 59 clock cycles. There is a 5% difference in the measurements predicted by these two techniques. However, it should be noted that the logic analyzer provides a measurement for a one particular execution of the code segment. This is a fundamental disadvantage of the logic analyzer approach, since there is no why of knowing whether or not the measured time is the true worst case execution time of the code segment.

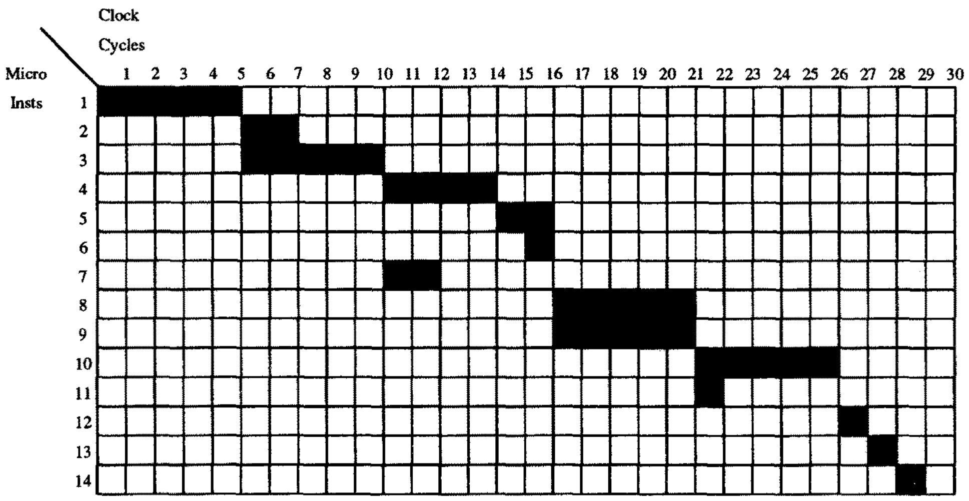

Figure 9. Graphical illustration of worst case execution time.

The table lookup and instruction counting methods overestimate the actual execution time computed by the logic analyzer by over 31 and 19 percent respectively. Both of these approaches also considerably underestimate the best case timing, due to the overly optimistic calculations of overlap and the inability to consider context when measuring overlap. The micro-analysis technique overestimates the worst case time by 3 clock cycles or 5 percent and overestimates the best case execution time by 2 clock cycles or 5 percent. For purposes of scheduling real-time tasks, a large overestimated task execution time reduces the potential for greater processor utilization. On the otherhand underestimating task execution time decreases system reliability.

6. Experimental Evaluation

Evaluation of the micro-analysis method was accomplished by comparing its performance on five programs to that of four traditional timing methods. Since micro-analysis predicts the execution time of code segments, rather than whole programs, each program in our test suite represents many tests at the code segment level. For example, the sort program in our test suite contained more than 10 code segments and over 56 execution paths. Each program had to be subdivided into code segments (basic blocks) and each code segment measured separately. Some special flow-analysis programs were written to automate the process of finding all the code segments that make up an execution path and preparing them for timing. The best and worst case execution time for the program were determined by comparing the timing results of each execution path. The execution time of each program

	Execution Cycles	
Timing Technique	Best case	Worst case
Logic Analyzer	37	59
Instruction Counting	21	70
Table Lookup	29	77
Micro-analysis	39	62

Table 1. Timing results of Ada the example.

was computed by table lookup, instruction counting, dual loop, logic analyzer and micro-analysis. The test programs used in the dual loop measurements were manually inspected after compilation to verify that the code segments being measured were not optimized away or moved outside the test loop by the compiler. The results are displayed using bar and line charts in Figures 10–12.

Our test suite consisted of five programs varying in size, instructions used and structure. All of the programs were tested on the primary development processor, the MC68020. One program uses the MC68881 coprocessor for floating-point operations. One of these programs was also run on the Intel 80386 machine to demonstrate the retargetability of micro-analysis. Programs tested on the MC68020 were compiled by the Verdix Ada cross compiler and the one tested on the 80386 processor was compiled by the Janus Ada compiler.

The programs compiled for the MC68020 processor were executed on a 32-bit monoboard computer (MVME133A-20) and the program designed for the 80386 was executed on a Mitsuba computer. Logic analyzer measurements for these programs were made with a Hewlett Packard 1650A logic analyzer. The clock resolution of the HP 1650A is 10 nanoseconds. Dual loop measurements were performed on the same computer and the timing results printed to a console connected through an I/O port on the MVME133A-20 monoboard computer. Predictions made using the table lookup, instruction counting, and micro-analysis methods were performed with software tools written in the course of this research and timing data found in user manuals.

6.1. Results of Timing Experiments

The D_Tree program tests an operation that might be performed frequently during the insertion of a task identification number into a priority queue data structure based on a decision tree. A static integer array is used to implement the decision tree. In this example

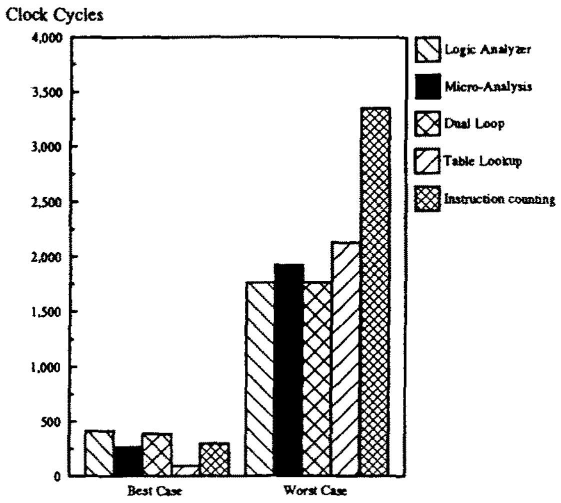

Figure 10. Performance comparison for D_Tree program.

we only measure the execution time of the insert procedure. The body of the insert procedure consists of a single bounded loop with one entry point and multiple exit points. The D_Tree program has the shortest execution time of our test programs. The bar chart in Figure 10 compares each method's best and worst case performance on the D_Tree program. It is important to note that the logic analyzer and dual loop measurements are the observed best and worst case execution times across all data sets used in the experiments. Micro-analysis is designed to predict the absolute best case and worst case execution time over all possible data sets.

Micro-analysis predicts a best case time that is 35% below the time measured by the logic analyzer and a worst case prediction that is 12% above the logic analyzer's worst case measurement. Table lookup predicts a best case time that is 77% below the logic analyzer's best case time, and its worst case time is 23% greater than the logic analyzer's worst case time. Instruction counting performs poorly in the worst case, but does much better in its best case prediction. For instance, its worst case is 95% over the logic analyzer's measurements and its best case prediction is only 28% below that measured by the logic analyzer. Dual loop performance is within 6% of the logic analyzer performance for both measurements. In the final analysis, dual loop provides the tightest upper bound for this example, however micro-analysis predicted a very realistic upper bound, and possibly a more comfortable bound for scheduling real-time programs.

The bar charts in Figures 11 and 12 compare the best and worst case times of each timing method on four different programs. The Vector Add program simply adds corresponding elements of two 1000 element integer arrays; unlike the other programs it was also tested on the 80386 processor to demonstrate the retargetability of the approach. Figure computes

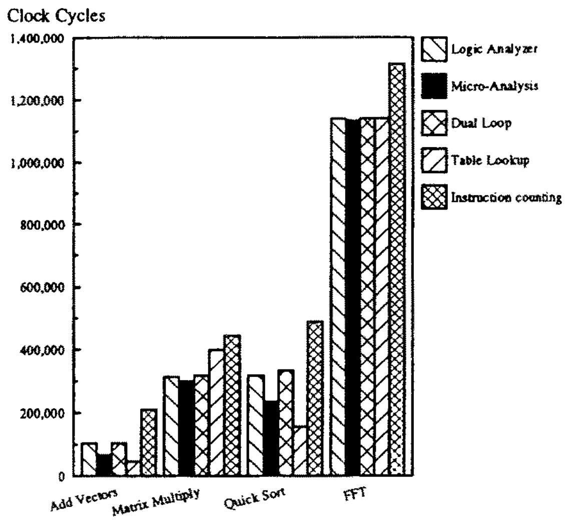

Figure 11. Best case performance comparison.

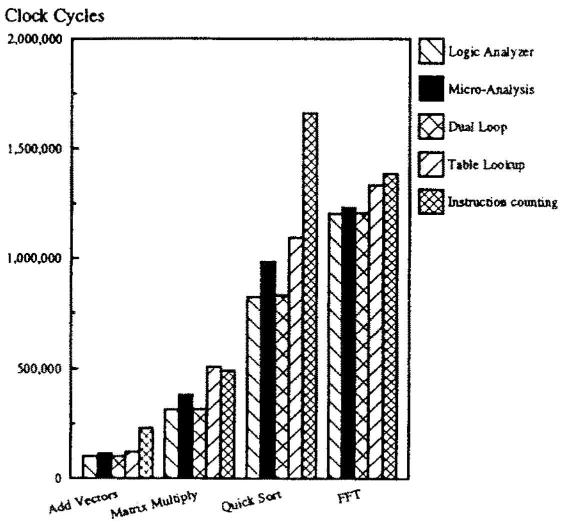

Figure 12. Worst case performance comparison.

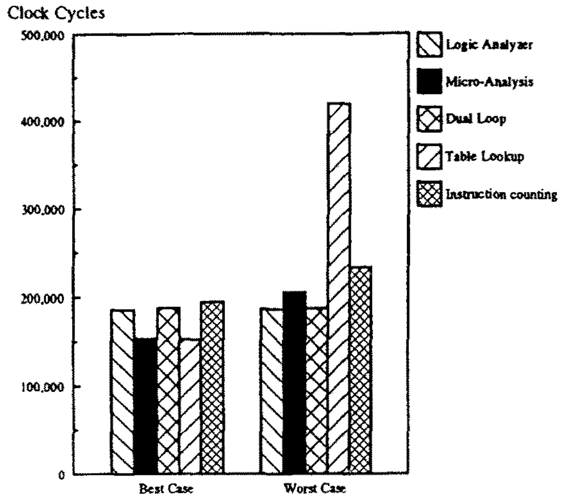

Figure 13. Performance comparison for Vector Add program.

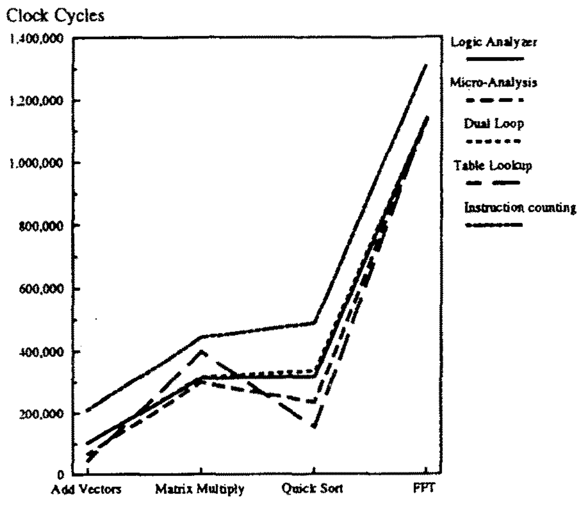

Figure 14. Best case graph comparison.

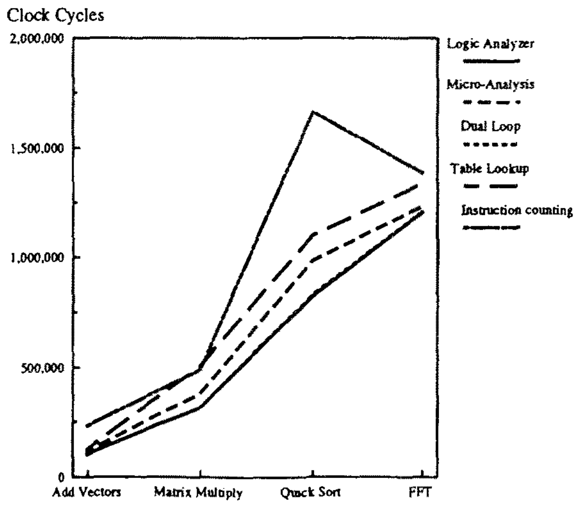

Figure 15. Worst case graph comparison.

the product of two 10 by 10 integer arrays. The implementation includes a triple level nested loop. Quick Sort is a recursive program that sorts a 100 element integer array. FFT is a discrete Fast Fourier Transform program that uses a large number of floating point operations, and it contains several nested loops. All floating point operations are performed by the MC68881 coprocessor.

The graphs in Figure 14 and Figure 15 are the line graph versions of the bar graphs in Figures 11 and 12. These graphs provide a better view of how well each method performs as program execution time increases. If the logic analyzer is considered to be the most accurate of the timing methods, then the data suggests that the dual loop method performed quite well on all tests. In fact, for all tests the measurements computed by dual loop are within 6% of those computed by the logic analyzer. It is worth noting the poor performance of the table lookup approach when applied to the Matrix Multiply program. This result illustrates the inability of the table lookup approach to accurately account for the effects of execution overlap in its predictions. The Matrix Multiply program consists of 3 nested loops that remain in cache until the program terminates after the initial iteration. This characteristic allows the processor to achieve maximum execution overlap, resulting in an actual best case execution time that is significantly less than that predicted by the table lookup approach, or by the instruction counting approach. We were not so surprised by the results of instruction counting since each instruction is assigned the same (constant) execution time regardless of its type, length, addressing mode, or execution context. The graph showing the best case analysis indicates that micro-analysis predicts a time that is consistently less than that measured by the logic analyzer. The performance of these two methods begins to converge as the execution time increases. This is the result of small errors compounding over time, causing the predictions to drift higher. The worst case analysis shows that micro-analysis predicts execution times that are slightly greater than those measured by the logic analyzer, but well below instruction counting and table lookup. These results indicate that micro-analysis out-performs table lookup and instruction counting by a small margin for short programs, but this margin increases as the execution time of the programs increases.

7. Predicting timing at Compile time

The retargetable timing analysis tool described in section 4 has been adapted to interface with *ease* (Environment for Architecture Study and Experimentation) (Davidson and Whalley 1991). The *ease* environment is designed to measure code produced by the back end of a C compiler known as *vpo* (Very Portable Optimizer) (Benitez and Davidson 1988). In this section we will describe revisions made to *ease* to support the timing analysis tool. The tool is capable of providing best and worst case execution time bounds for a user specified range of contiguous C statements or it can predict the bounded execution time for complete non-nested C functions. To invoke the tool the user need only specify the appropriate option along with the command to compile a C program. This integration of the timing tool with a compiler makes the tool easier to use and retarget, improves its efficiency since parsing of the assembly instructions is not required, and results in more accurate predictions.

7.1. The Compiler Interface

Modifications to the *ease* environment to support the timing tool were minor. When compiling a C source file with the option to collect static or dynamic frequency measurements, a file is produced that contains information about the characteristics of the instructions generated by the compiler. *ease* was modified to emit additional information to this information file about instructions, basic blocks, loops, and the control flow. The timing tool reads the information from the information file to predict execution times associated with the source code.

Data about each compiled function in the information file is structured in the following manner. First, the name of the function is emitted. Next, a record is generated for each loop within the function. The information represented for each loop includes the loop nesting level, the number of iterations if it is known, the set of basic blocks that comprise the loop, and the exit blocks (the set of blocks within the loop that have a successor that is not in the loop). Information about each basic block is produced after the loop information. First, a record containing information about the entire block is emitted. The block record contains the block number, the range of source lines associated with the block, a list of basic block predecessors, and a list of basic block successors. Source lines are associated with basic blocks instead of individual instructions due to the optimizations performed by the *vpo* back end. Thus, a request for the bounded execution times of a range of C statements in the prototype must include entire basic blocks. Following the basic block record is a description of each instruction within the basic block. Each instruction record consists of the instruction type, data type processed by the instruction operation, and indication of

```
16:   void summ(data1, data2, data3)
17:     int data1[], data2[], data3[];
18:   {
19:     int i;
20:     for (i = 0; i < 10; i++)
21:        data3[i] = data1[i] + data2[i];
22:   }
```

Figure 16. A simple C function.

```
function name: summ
loop: <nesting level 1> <num iters 10> <blocks 2 3>
<exit blocks 3>
block: <block 1> <lines 20-20> <succs 2>
<link> <areg long (a6)> <immed anyint ()>                    link  a6,#-4
<mov long> <indirect|mem long (a7)> <areg long (a5)>         movl  a5,a7@
<clr long> <dreg long (d1)> <immed anyint ()>                clrl  d1
<mov long> <areg long (a0)> <disp|mem long (a6)>             movl  a6@(data1.),a0
<mov long> <areg long (a1)> <disp|mem long (a6)>             movl  a6@(data2.),a1
<mov long> <areg long (a5)> <disp|mem long (a6)>             movl  a6@(data3.),a5
block: <block 2> <lines 21-21> <preds 3 1> <succs 3>
<mov long> <dreg long (d0)> <autoinc|mem long (a0)>     L41: movl  a0@+,d0
<add long> <dreg long (d0)> <autoinc|mem long (a1)>          addl  a1@+,d0
           <dreg long (d0)>
<mov long> <autoinc long (a5)> <dreg long (d0)>              movl    d0,a5@+
block: <block 3> <lines 20-20> <preds 2> <succs 4 2>
<addq long> <dreg long (d1)> <immed anyint ()>               addql   #1,d1
            <dreg long (d1)>
<cmp long ccset> <dreg long (d1)> <immed anyint ()>          cmpl    #10,d1
<jlt long> <label long ()>                                   jlt     L41
block: <block 4> <lines 22-22> <preds 3>
<mov long> <areg long (a5)> <disp|mem long (a6)>             movl    a6@(-4),a5
<unlk> <areg long (a6)>                                      unlk    a6
<ret>                                                        rts
```

Figure 17. Loop, basic block, and instruction information.

whether the condition codes are set and used by a subsequent instruction. In addition, the instruction record contains information about the data type, addressing mode, and register usage for each operand within the instruction.

A sample information file for the C function displayed in Figure The corresponding assembly instructions produced by the compiler are also listed to the right of each instruction record. The executable code in the C function spans source lines 20-22.

To interface the timing tool with the information file required only minor modifications to the timing tool implementation. First, the disassembler component was removed completely

	Execution Cycles	
Timing Technique	Best case	Worst case
Logic Analyzer	103120	103254
Instruction Counting	210133	231143
Table Lookup	46074	124129
Micro-analysis	67078	106795

Table 2. Micro-analysis using *EASE* vs. traditional methods.

and the parser component was reduced to a simple translator since the characteristics of each instruction is contained in the information file. The timing component was modified to compute the bounded execution time of basic blocks rather than individual instructions. Thus, even if a basic block is used in more than one path, its bounded execution time is only calculated once. The basic block execution times, loop iteration data, and flow-control information are used to calculate a bounded execution time for all execution paths.

Recently we have developed another tool which accepts assembly code from another compiler, translates the assembly to VPO's intermediate form, and passes the intermediate code into VPO to generate an information file. With this enhancement we can easily retarget the timing tool to any high-level language as long as the compiler generates assembly code or there is a disassembler available. This modification also eliminates the parsing step that was necessary in the initial version of the tool described in Section 4. We have used this new tool on assembly listings for Ada programs compiled by the Verdix Ada Cross Compiler Version 6.0.5 (Vadscross to MC68000 family). Table 2 compares timings predicted with the micro-analysis technique using information provided by *ease* to those measured by the dual loop and instruction counting approaches for a simple Ada procedure that performs the same task as the C function in Figure 16. The worst case time predicted by the micro-analysis technique bounds the worst case time measured by the logic analyzer tighter than the times predicted by instruction counting and table lookup.

Preliminary results indicate that this approach holds some promise. It is easy to use, retargetable, and reasonably accurate. Since *ease* can be used to emulate features of proposed architectures, it could also be used to predict the performance of software on a proposed machine.

8. Future Work

An important extension to this research is to address the execution time prediction of much larger code segments. There are a number of challenges to this problem. First, larger code segments will typically contain function calls. A tool must be able to determine the instructions that could be executed when these functions are invoked. This analysis requires the construction of a call graph. Another problem is dealing with the cache memory. If function calls are allowed in the code segments to be timed, then there is a much greater likelihood that cache conflicts will occur since there would be a greater number of instructions that can be reached and the functions accessed may not be contiguous in memory. The simple assumption used in the current tool that at most one miss will occur for each reference will no longer be sufficient. The authors have current work in progress that uses an iterative flow analysis technique to determine which instruction references will always be cache hits or always cache misses. A more challenging problem is predicting hits and misses in the data cache since the addresses associated with data references are not always known statically. Other issues that need to be addressed associated with Reduced Instruction Set Computers (RISC) include correctly predicting the stalls due to pipeline hazards and handling floating-point operations that take multiple cycles.

9. Conclusion

This paper describes a retargetable tool for predicting a bounded best case and worst case execution time of code segments. In addition to being retargetable, micro-analysis has the added advantage of being language and compiler independent. The timing tool is currently predicting execution time of code segments targeted for the Motorola MC68020 and Intel 80386 processors. The timing tool has been integrated with a version of the *vpo* C compiler and the *ease* environment. A prototype has been built and preliminary tests are very promising. The integrated version of the timing tool is easier to use, retargetable and preliminary results indicate that it is just as accurate as the original version. Furthermore, since *ease* is capable of emulating the features of proposed architectures, the integrated version of the timing tool can be used to predict the performance of software on a proposed machine.

Acknowledgements

This work was supported in part by grants N00014-87-J-1166 and N00014-92-J-1248, from the U.S. Office of Naval Research. An earlier version of this paper was published in the 1992 *Proceedings of the IEEE Real-Time Systems Symposium.*

Notes

1. The Ada program shown here requires that the user initialize the constant LOOP-COUNT and replace the comment TEST-CODE-SEGMENT with the actual sequence of instructions to be timed.

References

N. Altman and N. Weiderman. Timing variation in dual loop benchmarks. Technical Report CMU/SEI-87-TR-22, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1987.

C. N. Arnold. Using the ETA system multiprocessing simulator to prepare for the ETA^{10}. *I/O*, 4(1):9–12, 1987.

T. B. Baker. A single-pass syntax-directed front end for Ada. In *Proceeding of the SIGPLAN'82 Symposium on Compiler Construction*, pp. 318–326, Boston, MA, 1982.

M. E. Benitez and J. W. Davidson. A portable global optimizer and linker. In *Proceedings of the SIGPLAN Notices '88 Symposium on Programming Language Design and Implementation*, pp. 329–338, Atlanta, GA, 1988.

R. M. Clapp, L. Duchesneau, R. A. Volz, T. N. Mudge, and T. Schultze. Toward real-time performance benchmarks for Ada. *Communications of the ACM*, 29(8):760–778, 1986.

J. W. Davidson and C. W. Fraser. Automatic generation of peephole optimization. In *Proceedings of the SIGPLAN'84 Symposium on Compiler Construction*, pp. 111–116, Montreal, Canada, 1984a.

J. W. Davidson and C. W. Fraser. Code selection through object code optimization. *Transactions on Programming Languages and Systems*, 6(4):7–32, 1984b.

J. W. Davidson. A retargetable instruction reorganizer. In *Proceedings of the SIGPLAN NOTICES '86 Symposium on Compiler Construction*, pp. 234–251, Palo Alto, CA, 1986.

J. W. Davidson and D. B. Whalley. A design environment for addressing architecture and compiler interactions. *Microprocessors and Microsystems*, 15(9):459–472, 1991.

A. Hook, G. A. Riccardi, and M. Vilot. Ada compiler performance benchmark. In *Ada: Managing the Transition, Proceedings of the Ada-Europe International Conference*, pp. 33–42, Edingurgh, Scotland, 1986.

Intel Corporation. *80386 Programmer's Reference Manual.* Intel Corporation, 1988.

C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the Association for Computing Machinery*, 20(1):46–61, 1973.

P. McKerrow. *Performance Measurement of Computer Systems.* Addison-Wesley, 1988.

A. K. Mok. SARTOR—A design environment for real-time systems. In *Proceeding of the 9th IEEE COMPSAC*, pp. 174–181, 1985.

A. K. Mok. The design of real-time programming systems based on process models. In *Proceedings of the 1984 IEEE Real-Time Systems Symposium*, pp. 5–17, Austin, TX, 1984.

A. K. Mok. Evaluating tight execution time bounds of programs by annotations. In *Sixth IEEE Workshop on Real-Time Operating Systems and Software*, pp. 74–80, Pittsburgh, PA, 1989.

Motorola. *MC68020 32-Bit Microprocessor User's Manual second edition.* Prentice-Hall, 1985.

D. Niehaus. Program representation and translation for predictable real-time systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, pp. 53–63, San Antonio, TX, 1991.

D. I. Oh. A table driven retargetable disassembler generator. Master's Project, Department of Computer Science, Florida State University, Tallahassee, FL, 1989.

C. Y. Park. Predicting program execution times by analyzing static and dynamic program paths. *Real-Time Systems*, 5(1):31–61, 1993.

C. Y. Park and A. C. Shaw. A source-level tool for predicting deterministic execution times of programs. Technical Report 89-09-12, Department of Computer Science and Engineering, University of Washington, Seattle, WA, 1989.

C. Y. Park and A. C. Shaw. Experiments with a program timing tool based on source-level timing schema. *Computer*, 24(5):48–57, 1991.

P. Puschner and C. H. Koza. Calculating the maximum execution time of real-time programs. *Real-Time Systems*, 1(2):159–176, 1989.

D. Roy. PIWG measurement methodology. *ACM Ada Letters Special Edition*, 10(3):72–90, Winter 1990.

A. C. Shaw. Reasoning about time in high-level language software. Research Report, Laboratoire MASI, University of Paris 6, 1987.

A. D. Stoyenko. A real-time language with a schedulability analyzer. Ph.D. Thesis, Department of Computer Science, University of Toronto, Toronto, Canada, 1987.