A Retargetable Technique for Predicting Execution Time*

Marion G. Harmon
Department of Computer And Information Systems
Florida A & M University
Tallahassee, FL 32307, U. S. A.

T. P. Baker
David B. Whalley
Department of Computer Science
Florida State University
Tallahassee, Fl 32306, U. S. A.

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.

First a summary of some of the hardware and software factors that make predicting execution time difficult is presented, along with the results of experiments that evaluate the degree of variation in execution time that may be caused by these factors. Traditional methods of measuring and predicting execution time are examined, and their strengths and weaknesses discussed.

Second, we present a new technique for predicting point-to-point execution times on contemporary microprocessors. This technique is called micro-analysis. 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 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 caching and instruction overlap can be taken into account. This technique is compiler and language-independent, and retargetable.

Finally, we describe a prototype system in which the micro-analysis technique is integrated with an existing C compiler. This early version predicts the bounded execution time of statement ranges or simple (non-nested) C functions at compile time.

Keywords: Real-time systems, point-to-point execution time, best case time, worst case time, and predicting execution time.

*This work was supported in part by grant N00014-87-J-1166, from the U.S. Office of Naval Research.

1 Motivation

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 [?], Mok [?]). Stoyenko's work [?] 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 [?] 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 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 [?] have implemented a timing tool that analyzes a stream of assembly language instructions generated from the compilation of C programs (using a graph method to find the worst case path) and computes the execution time by simulating the hardware. Park and Shaw [?] implemented a timing tool for a subset of C, based on the notion of timing schema presented in [?]. A method very similar to that of Shaw is presented by Puschner and Koza [?]. 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., 68010), more research is needed to determine timing predictability on more complex contemporary machines (e.g., 68020, 68030, 80386, etc.).

1.1 Organization

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 early version predicts execution times for statment 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 when applied to contemporary processors.

2.1 Table Lookup Method

This approach 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 method is easy to implement. In fact, there are two approaches to implementing this method. One approach is to perform a post analysis on the assembly listing produced by the compiler, assuming the compiler produces an assembly language listing. Another approach is to integrate the accumulation procedure with the compiler, so that the analysis will be performed as the code is generated.

The table lookup method 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, using the code disassembly approach, 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 [?]. Also, operand size and addressing mode can influence program performance, but both are generally overlooked in discussions on execution time.

2.2 Dual Loop Benchmark

The dual loop benchmark paradigm is a commonly used method of measuring code execution time using a standard system clock [?]. 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 [?], 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), and the University of Michigan test suite [?].

The resolution of the system clock varies widely from one implementation to another. The dual loop 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). 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 contemporary processors to perform multiple bus cycles for the operand transfer.

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. Altman and Weiderman in [?] 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 final 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 optimizations issues apply here as well. For instance, a compiler may remove instructions from the control loop through optimization making it necessary to write additional code to suppress the effects of optimization; it is unlikely that the same context would exist in the application.

2.3 Simulation

Using software to simulate [?] the target processor is another common approach to determining the timing behavior of a code segment. 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 [?]. The accuracy of the prediction 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.

2.4 Direct Measurement

An oscilloscope or a logic analyzer may be used to measure the execution time of a code segment. 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. 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, 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 [?] 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 retargetable disassembler generator [?] that was used to build disassemblers for the MC68020 and 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 known as *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 [?]. 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 (68020 processor) and a Mitsuba personal computer (80386 processor). The tool has been used to predict the execution time of programs written in Ada, C, and assembly language.

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

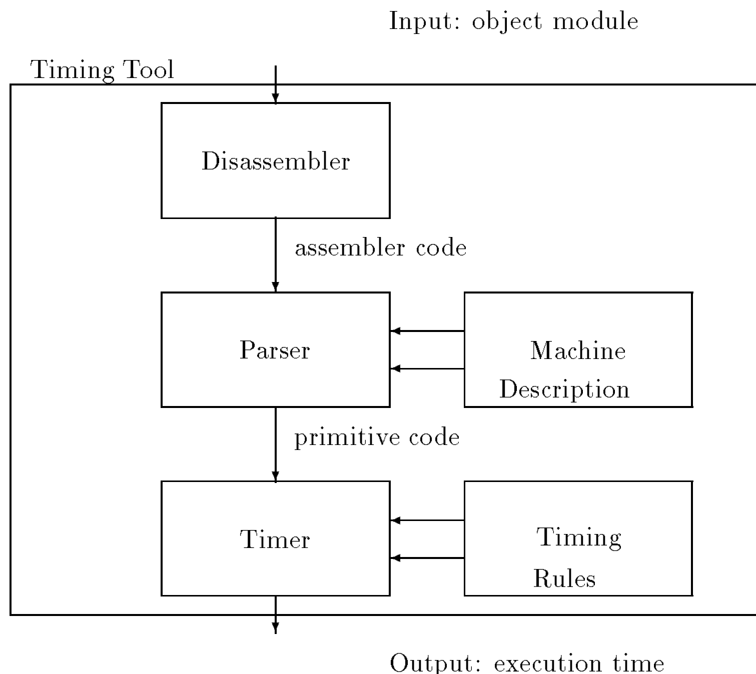

Figure 1: Overview of tool components

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 ?? shows the organization of the tool.

5 A Simple Timing Analysis Example

The Ada procedure in Figure ?? is used to illustrate the use of our tool. This sample Ada procedure was compiled using the VERDIX 5.7 cross compiler. Figure ?? shows the output that the user sees after the disassembly stage. 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 ??) that express the functionality of the corresponding machine instruction(s) in fine grain detail. For this example, let's assume the user has requested timing for statements 4 through 5. Each

primitive denotes a specific machine level operation, for instance **fetch** denotes an instruction fetch, **=Ms** denotes a data write operation, and **=sM** denotes a data read operation.

```
1 procedure EXAMPLE is
2  X, Y: integer:= 5;
  begin
3  for i in 1..100 loop
4    X := X + Y;
5    Y := i + X
6  end loop;
7 end EXAMPLE;
```

Figure 2: Line numbered Ada procedure

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

Figure 3: Fine-grain primitives for lines 4-5 in Figure 2

The timer component uses a set of timing rules which incorporate the architectural features and execution paridigm of the target processor. A timing rule consists of a left-hand side (LHS) and a right-hand side (RHS). The LHS is always a pattern (e.g., one or more primitives). 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: **fetch** & **fetch** → **pgr**. This rule will replace a pattern of fetch primitives with the higher level pattern **pgr** 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. 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.

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

Figure 4: Time is expressed in clock cycles

The table in Figure ?? compares timings predicted by micro-analysis to those measured or predicted using some well-known techniques of evaluating execution time. The instruction counting technique predicts execution time by multiplying the number of machine instructions in the code segment by a pair of constants which represent the average best and worst case instruction execution time for the target machine. The table lookup approach predicts execution time by accumulating the execution times given in the hardware user's manual. 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.

6 Experimental Evaluation

Evaluation of the micro-analysis method was done by comparing its performance to that of four traditional timing methods, on five programs. 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 was

computed by table lookup, averaging, dual loop, logic analyzer and micro-analysis. The results are displayed, using bar and line charts, in Figures ??- ??.

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 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, averaging, and micro-analysis methods were performed with software tools written in the course of this research and timing data found in users 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 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 ?? 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. Averaging performs poorly in the worst case (95% over the logic analyzer's measurement), but does much better in its best case prediction (only 28% below the logic analyzer), far better than table look up. 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 ?? and ?? compare the performance (for best and worst case) 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. Figure ?? compares the results measured on the 80386. Matrix Multiply computes 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 ?? and Figure ?? are the line graph versions of the bar graphs above. 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 suggest 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. The performance of table lookup is unpredictable. 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 averaging approach. We were not so surprised by the results of averaging,

Figure 5: D_Tree program

Figure 6: Best case performance comparison

Figure 7: Worst case performance comparison

Figure 8: Vector Add program (80386 version)

Figure 9: Best case graph comparison

Figure 10: Worst case graph comparison

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 averaging and table lookup. These results indicate that micro-analysis out-performs table lookup and averaging 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) [?]. The *ease* environment is designed to measure code produced by the back end of a C compiler known as *vpo* (Very Portable Optimizer) [?,?,?]. 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 contigious 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 (no parsing of assembly instructions is 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 functions 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 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 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 ?? is illustrated in a readable format in Figure ??. The corresponding assembly instructions produced by the compiler are also listed to the right of each instruction record. The for loop in the C function spans source lines 20-21.

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 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 blocks execution times, loop iteration data, and flow-control information are used to calculate a bounded execution time for all execution paths.

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

```
void summ(data1, data2, data3)

  int data1[], data2[], data3[];
  {
   int i;
     for (i = 0; i < 10; i++)
        data3[i] = data1[i] + data2[i];
   }
```

Figure 11: 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 20-20> <preds 3>
<mov long> <areg long (a5)> <disp|mem long (a6)>          movl    a6@(-4),a5
<unlk> <areg long (a6)>                                   unlk    a6
<ret>                                                     rts
```

Figure 12: Loop, basic block, and instruction information

performance of software on a proposed machine. Future research will first focus on developing a RISC version of the tool. One challenge will be to correctly predict the stalls due to pipeline hazards. Another planned improvement is to allow the prediction of execution times of functions containing calls which will require the construction of a call graph. A final interesting area of research would be to attempt to provide more accurate estimates of cache performance for execution paths. Better accuracy could be accomplished by determining the different states a cache can be in at the point a path can be executed.

8 Conclusion

This paper describes a retargetable tool for predicting 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 68020 and 80386 processor. 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 test are very promising.

References

[1] 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 15213, October,1987.

[2] C. N. Arnold, "Using the ETA System Multiprocessing Simulator To Prepare for The ETA^{10} I/O", 4, 4(1), 9-12. (1987)

[3] T. P.Baker, "A Single-Pass Syntax-Directed Front End for Ada", *Proceeding of the SIGPLAN'82 Symposium on Compiler Construction*, (1982), pp. 318-326.

[4] M. E. Benitez and J. W. Davidson, " A Portable Global Optimizer and Linker", *Proceedings of the SIGPLAN Notices '88 Symposium on Programming Language Design and Implementation*, Atlanta, GA, Jun 1988, 329-338.

[5] R.M. Clapp et al. "Toward Real-Time Performance Benchmarks for Ada" Communications of the ACM 29(8):760-7788, August, 1986.

[6] J. W. Davidson and C.W. Fraser, "Automatic Generation of Peephole Optimization", *Proceedings of the SIGPLAN'84 Symposium on Compiler Construction*, (1984), pp. 111-116.

[7] J. W. Davidson and C. W. Fraser, "Code Selection through Oject 'Code Optimization", *Transactions on Programming Languages and Systems* 6, 4(October 1984), 7-32.

[8] J. W. Davidson, "A Retargetable Instruction Reorganizer", *Proceedings of the SIGPLAN NOTICES '86 Symposium on Compiler Construction*, Palo Altp, CA, June 1986, 234-241.

[9] J. W. Davidson and D. B. Whalley, "Ease: An Environment for Architecture Study and Experimentation", *Proceedings SIGMETRICS '90 Conference on Measurement and Modeling of Computer Systems*, Boulder, CO. May 1990 P 259-260.

[10] A. Hook, G. A. Riccardi, and M. Vilot, "Rational For The Prototype Ada Compiler Evaluation Capability (ACEC) Version 1 and Recommendations for Research and Development of Successive Versions", IDA paper -1915, Dec 1985.

[11] Intel Corporation, *80386 Programmer's Reference Manual*, Intel Corporation, Santa Clara, CA, 1988.

[12] C. L. Liu and J.W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment", *Journal of the Association for Computing Machinery, Vol. 20, No. 1*, (January 1973), pp. 46-61.

[13] A. K. Mok, "SARTOR-a design environment for real-time systems", in Proc. 9th IEEE COMPSAC, Oct. 1985, pp. 174-181.

[14] A. K. Mok, "The Design of Real-Time Programming Systems Based on Process Models", *Proceedings of the 1984 IEEE Real-Time Systems Symposium*, (December 1984), pp. 5-17.

[15] A. K. Mok, "Evaluating Tight Execution Time Bounds of Programs by Annotations", *Sixth IEEE Workshop on Real-Time Operating Systems and Software*, (May 1989), pp. 74-80.

[16] *Motorola MC68020 User's Manual*, second edition Prentice-Hall. MC68020UM/AD REV 1.

[17] Oh, D.I., "A Table Driven Retargetable Disassembler Generator", *Master's Project (1989)*, Department of Computer Science, Florida State University.

[18] C. Y. Park, C.Y. and Shaw, A.C., "A Source-Level Tool for Predicting Deterministic Execution Times of Programs", *Technical Report # 89-09-12 (Sep. 1989)*, Department of Computer Science and Engineering, University of Washington.

[19] P.Puschner and Koza, CH., "Calculating the Maximum Execution Time of Real-Time Programs", *The International Journal of Time-Critical Computer Systems*, Vol. 1, number 2, September 1989.

[20] A. C. Shaw, "Reasoning About Time in High-Level Language Software", *Research Report, Laboratoire MASI*, University of Paris 6, April 1987.

[21] A. D. Stoyenko, "A Real-Time Language With A Schedulability Analyzer", *Ph. D. Thesis*, University of Toronto, August 1987.