Software Performance Estimation for Pipeline and Superscalar Processors

Chu-Yi Huang
Daniel D. Gajski

June 16, 1995

Technical Report #95-20

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92717-3425
(714) 824-8059

chu-yih@ics.uci.edu
gajski@uci.edu

Abstract

When moving toward hardware/software codesign, software estimation provides important information in chosing hardware implementation (ASIC) or software implementation (software running on processor). This report analizes the pipeline stall and superscalar interlock phenomenon and their influence on software performance. Simple processor profile is proposed to count these two effects. Based on generic estimation model, our estimator can produce accurate estimation without large computation time and precious resource, such as compilers or simulators for each processor. The accuracy and flexibility make our approach suitable for design automation tools.

Contents

1 Motivation **1**

2 Problem Analysis **2**
2.1 Software Estimation 2
2.2 Pipeline Stall and Superscalar Interlock 4

3 Estimation Model **4**
3.1 Pipeline Processor 4
3.2 In-Order Issue Superscalar Processor 8
3.3 Out-Of-Order Issue Superscalar Processor 9
3.4 Software Performance Estimator 11

4 Experiment **11**

5 Conclusion and Future Work **13**

6 Acknowledgements **14**

1 Motivation

In system level design, designers have to decide to implement the specification in hardware (ASIC), software (running on processor), or combination of these two. Performance and cost are the most important decision factors. To explore the space of hardware/software codesign, design tools have to provide the performance and cost information.

To provide metrics for hardware/software codesign, both hardware and software estimator are needed. Hardware estimator provides performance and cost metrics for implementing the specification in ASIC. Software estimator provides performance and cost metrics for implementing specification in software running on processor.

This paper discuss the software performance estimation, even though the tool can produce the cost metrics (such as code size, data size), too. Please reference [4] for cost estimation technique. For hardware estimation, please reference [5].

Approach	Work Flow	Flexibility	Computation Time	Accuracy
In–Circuit Simulation	Specification → Machine Code → Processor	Low	Low	High
Software Simulation	Specification → Machine Code → Simulator	Mid	High	High
Software Estimation	Specification → Processor Profile → Estimator	High	Low	Mid

Table 1: Comparision of Software Estimation Approach.

Software performance can be measured in three ways: in-circuit simulation, software simulation, and software estimation. The comparison of these three approaches are summaried in Table 1. In hardware simulation, system specification was compiled into target machine code and performance was measured by running the code on target processor. This approach need one in-circuit simulator and one compiler for each processor. Though the measured metrics is accurate but the flexibility of the tool is low due to the huge resource requirement.

In software simulation, system specification was compiled into target machine code and per-

formance was measured by running the code in software simulator of the target processor. This approach needs one compiler and one simulator for each processor. In this approach, the estimation accuracy is high (the same as in-circuit simulation) and the flexibility is higher because software simulator is more accessible than in-circuit simulator. But software simulator is very time consuming.

In software estimation, the estimator calculates the performance of the system specification based on a profile of each processor. No compiler, software simulator, or in-circuit simulator are needed. This approach only consume less computation time and need less resource, one technology file for each target processor and one estimator only. The accuracy is lower than the previous two approaches.

From this comparison, we know the software estimator is more suitable for design automation tools because its flexibility and speed allow design space exploration. In this paper, we are trying to develop an estimation method for pipeline and superscalar processors that can produce accurate enough metrics for system level design tools.

2 Problem Analysis

2.1 Software Estimation

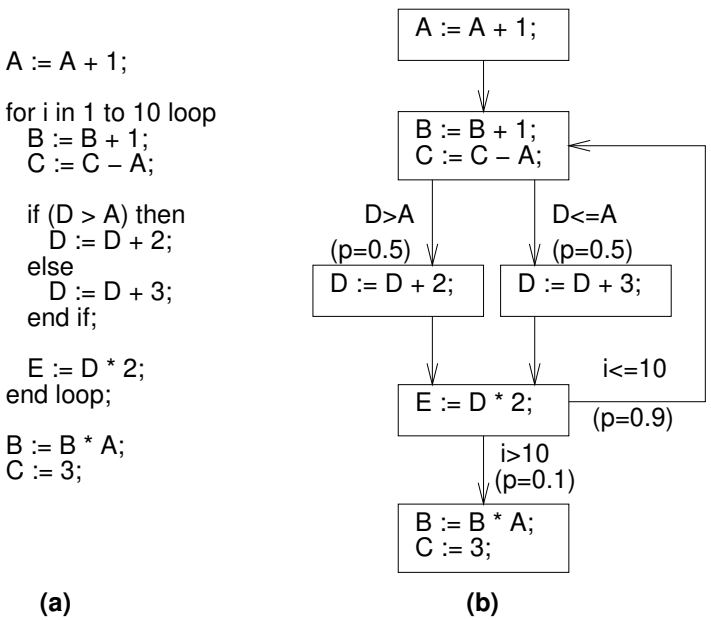


Figure 1: (a) VHDL program, (b) basic block graph.

Software estimation can be divided into two steps: flow analysis and basic block estimation. In

flow analysis, system specification was divided into several basic blocks, as the example in Figure 1. Basic block is a straight-line code which has no branches. Every branch among basic blocks is associated with a probability that this branch will be taken. The execution frequency of each basic block can be calculated based on the graph of basic block and branch probability [5].

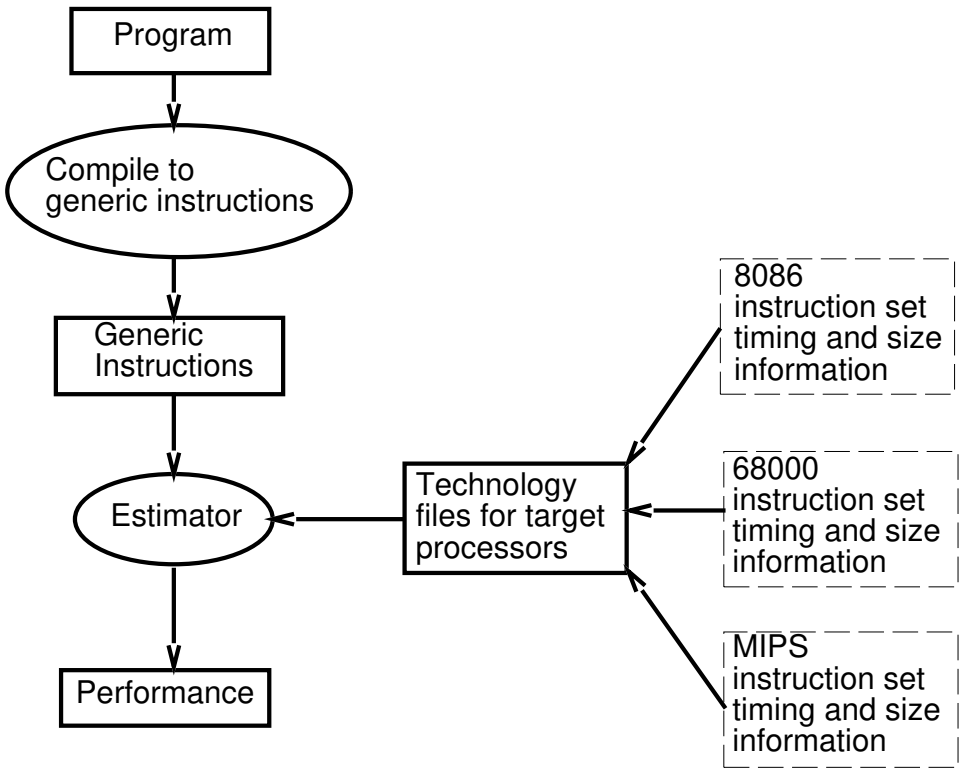


Figure 2: Generic Estimation Model.

After the execution time of each basic block was measured in the second step, the execution time of the whole specification can be calculated by following equation:

$$execution(S) = \sum_{b_i \in S} execution(b_i) \times freq(b_i) \tag{1}$$

where b_i is basic block.

This report takes the same approach and focus on the basic block estimation for pipeline and superscalar processors. In basic block estimation, we adapt the generic estimation model [4] which was shown in Figure 2. The system specification was compiled into generic three address instructions. For each processor, there is a technology file providing the timing and instruction size of

each generic instruction. Then the execution time and code size of basic block can be calculated as follows:

$$execution(B) = \sum_{I_j \in B} time(I_j) \tag{2}$$

$$size(B) = \sum_{I_j \in B} size(I_j) \tag{3}$$

where I_j is the generated generic instruction.

2.2 Pipeline Stall and Superscalar Interlock

Though generic estimation model has shown good results in estimation for non-pipelined processor in [4], the previous two equations can not measure the performance of pipeline or superscalar processor very well. The overlap time of pipeline instruction can be reflected in following equation:

$$execution(B) = \sum_{I_j \in B} (time(I_j) - pipe_depth + 1) + pipe_depth - 1 \tag{4}$$

But the result is not accurate due to pipeline stall and superscalar interlock phenomenon.

Pipeline processor can issue one instruction per clock cycle in ideal situation. When one instruction depend on the result or content a same resource, such as bus, of previous instructions that are still in the pipe, this instruction can not be issued and pipeline was stalled. Figure 3 shows an example of pipeline stall in SPARC processor. When executing the second generic instruction, instruction st_2 can not be fetched until clock cyle fifteen because it content the bus with instruction st_1 and ld_3. Two new pipeline stalls (marked by black) happened in clock cycle 15 and 16. The execution time will be 20 clock cycle, not the 18 clock cyle measured by Equation 4.

Superscalar processor can issue more than one instruction per clock. Processor check the issue rule to decide a new instruction can be issued or not. Instructions depend on the result or content same resources (such as bus or ALU) of previous instructions which are not available yet can not be issued. This is called superscalar interlock and prevent the processor achieving maximal performance. Figure 4 shows an execution sequence in Pentium superscalar processor. In this example, instruction mov_2 and mov_3 (marked by black) can be executed at the same time that save one execution clock cycle if the two generic instruction are executed continuously.

3 Estimation Model

3.1 Pipeline Processor

The hardware model of pipeline processor was depicted in Figure 5. Pipeline processor was profiled as a sequence of function unit which can handle one instruction at a time. In order to

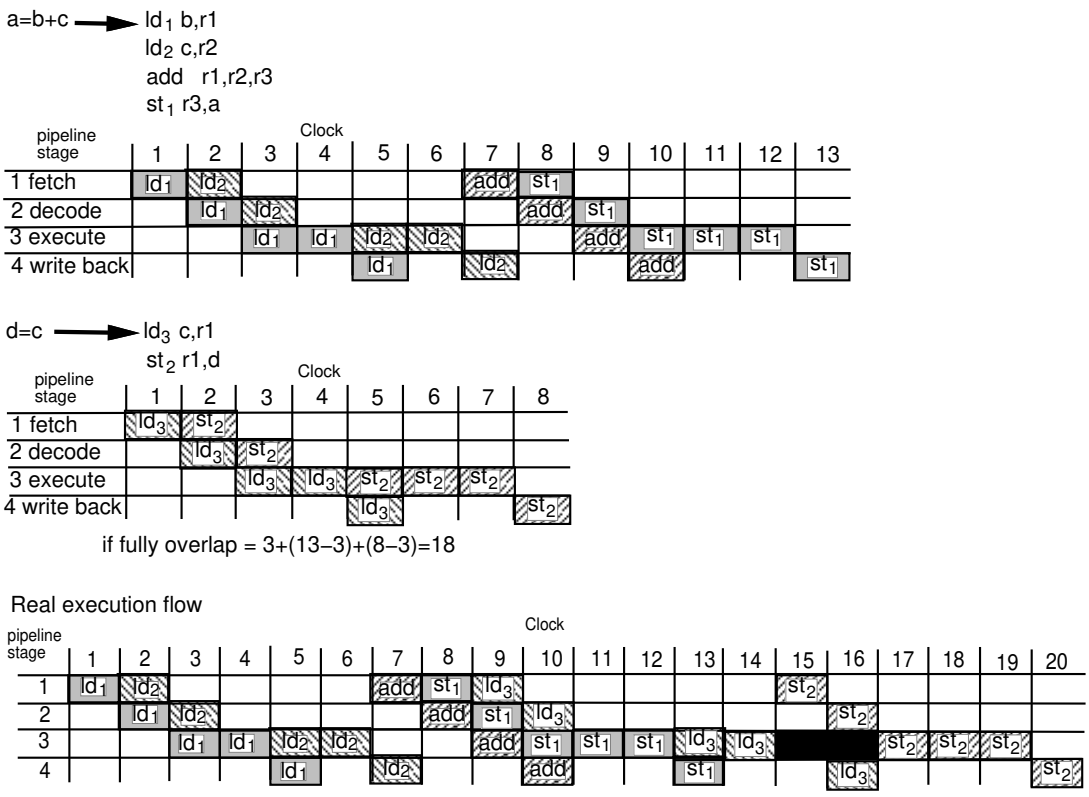


Figure 3: Pipeline Stall Example in SPARC.

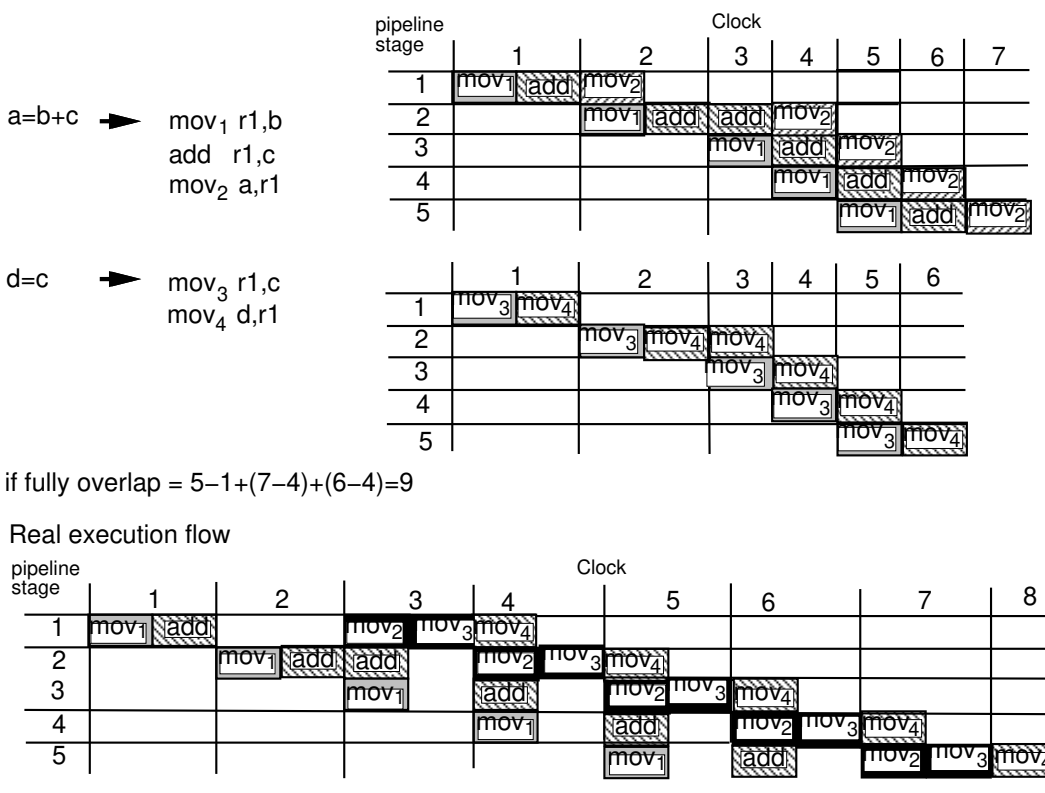
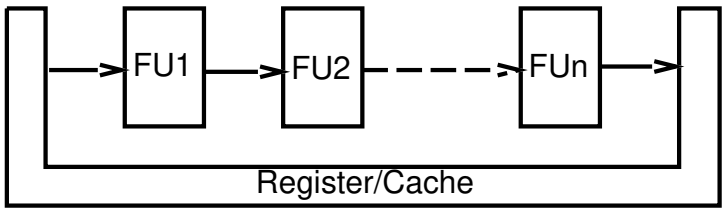


Figure 4: Execution sequence in Pentium.



Technology File

- Timing for each generic instruction
- Resource Contention Stall Table (RCStall)

	reg add	assign
reg add	0	0
assign	1	1

- Data Dependency Stall Table (DDStall)

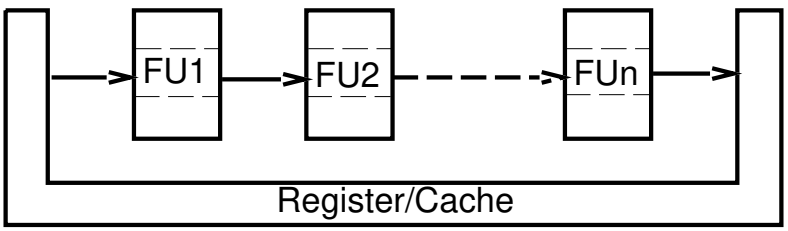
Figure 5: Pipeline Processor Model.

calculate the pipeline stall, two pipeline stall tables were included in addition to the execution time of each generic instruction. As the example in Figure 3, pipeline stalls in each generic instruction were already covered in the execution time of generic instruction. Pipeline stalls between generic instructions were stored in pipeline stall table. Pipeline stall table is a two dimension array. Both the x and y dimension are generic instructions. And the array value is the number of pipeline stalls introduced when the instruction indexed by x is following the instruction indexed by y. The two pipeline stall tables have the same format, but having different value. Data dependent pipeline stall table, *DDStall*, is used when an instruction depend on the result of its previous one instruction, while resource contention pipeline stall table is used for instructions that have no data dependency. The execution time of a basic block can be formulated as follows:

$$execution(B) = \sum_{I_j \in B}(time(I_j) + stall(I_j - 1, I_j) - pipe_depth + 1) + pipe_depth - 1 \quad (5)$$

$$\text{where } stall = \begin{cases} RCStall(I_{j-1}, I_j) & \text{if } I_j \text{ dose not depend on } I_{j-1} \\ DDStall(I_{j-1}, I_j) & \text{if } I_j \text{ depend on } I_{j-1} \end{cases}$$

3.2 In-Order Issue Superscalar Processor



Technology File

- Mapping of generic instruction to machine instruction
- Timing information for machine instruction
- Dispatch Rule: 2 add, 1 mul, 1 load
  1 add, 1 mul, 1 load, 1 branch

Figure 6: In-Order Issue Superscalar Processor Model.

The estimation model of in-order issue superscalar processor was shown in Figure 6. This model adds a new feature: each function unit can handle more than one instruction.

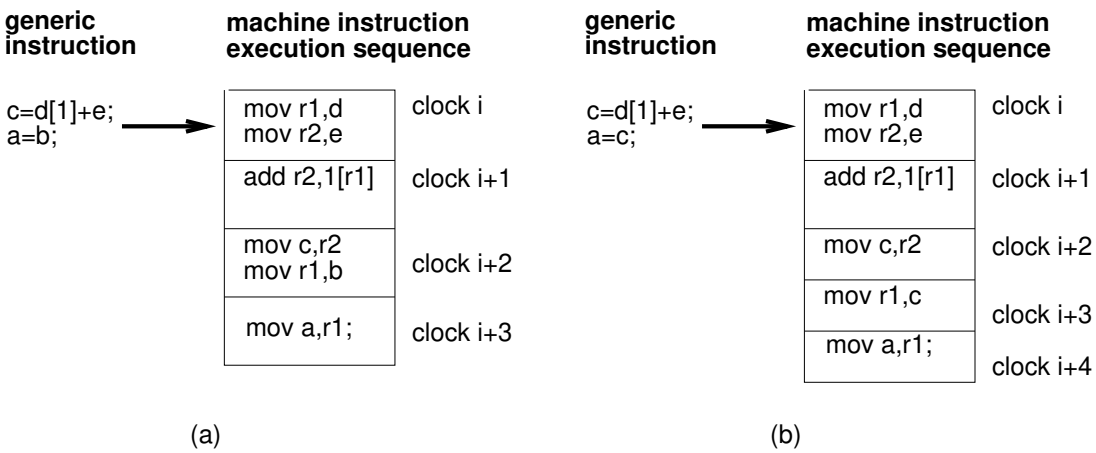


Figure 7: Grouping machine instruction in Pentium.

The technology file for this type of processor changes a lot. First, the mapping of generic instruction to machine instruction should be included. When a processor executing the machine instruction of a generic instruction, it fetches more than one machine instructions and check against issue rule. If these instruction satisfy the issue rule, they are grouped together and execute at the same time. The grouping of machine instruction of a same generic instruction is not always the same in different execution instances. For example, *mov c, r2* is grouped with *mov r1, b* in Figure 7(a) while *mov c, r2* in Figure 7(b) consist a group by itself even though they belong to the same generic instruction, an assignment statment. So the mapping of generic instruction to machine instruction should be provided, that the estimator can check the issue rule at machine instruction level.

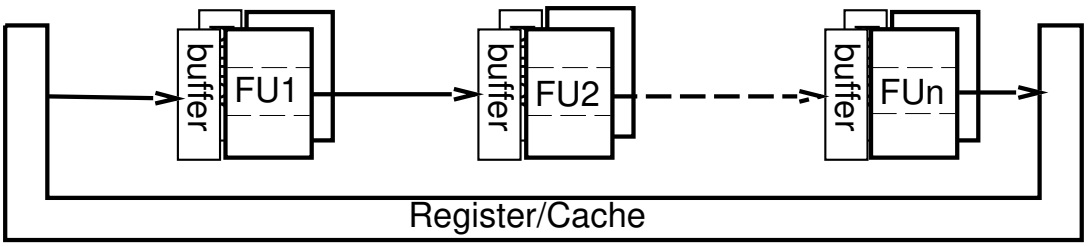
Second, the execution time of each machine instruction, not generic instruction, should be included. After checking the issue rule, the estimator can determine the parallel instruction group and calculate the performance as follows:

$$\begin{array}{rl} execution(B) = & pipe_depth - 1 + \sum_{MIG_j \in B} time(MIG_j) \\ \text{where} & MIG_j \text{ is a parallel machine instruction group,} \\ time(MIG_j) = & MAX_{MI_{jk} \in MIG_j} time(MI_{jk}) \\ \text{where} & MI_{jk} \text{ is a machine instruction of group } MIG_j. \end{array} \tag{6}$$

Of course, the issue rule is included in the technology file.

3.3 Out-Of-Order Issue Superscalar Processor

Out-of-order issue processor can execute more than one instruction at one time and rearrange the instruction order. When one instruction was stalled, the processor will look ahead several



Technology File

- Mapping of generic instruction to machine instruction
- Timing information for machine instruction
- Dispatch Rule
- Number of FUs and buffers

Figure 8: Out-Of-Order Issue Superscalar Processor Model.

instructions to find a free instruction that can be executed now. So a look ahead buffer was inserted in front of each function unit, as shown in Figure 8.

In this model, each pipeline stage may has more than one function unit and each function unit may handle multiple instructions at the same time. In most cases, there is only one function unit that can handle multiple instructions in fetch and decode stage. In execution stage, multiple function units exist and each can handle one or multiple instructions. When one function unit is busy, other function units can execute instructions from buffer.

The technology file for this type of processor has the number of function unit and size of buffer in each pipeline stage in addition to the technology file of in-order issue processor. In estimation, the estimator schedule and bind machine instructions to function unit according to issue rule, data dependency, and function unit availability. Then the performance is calculated as follows:

$$\begin{array}{rl} execution(B) = & MAX_{\text{all } FU_i} stop_time(FU_i) \\ \text{where} & FU_i \text{ is a function unit,} \\ stop_time(FU_i) = & \sum_{MI_{ij} \in FU_i} time(MI_{ij}) \\ \text{where} & MI_{ij} \text{ are machine instructions executed by } FU_i. \end{array} \tag{7}$$

3.4 Software Performance Estimator

Figure 9 shows the flowchart of software performance estimator. Specification was divided into basic blocks, first. Then each basic block was compiled into generic instructions. If it is a non-pipelined processor, look up the timing in technology file. Sum up these timing to get the execution time. If it is a pipeline processor, check the data dependency and look up pipeline stall table. Calculate execution time by Equation 5. Otherwise the generic instructions were mapped to machine instruction. If it is a in-order issue superscalar processor, check the data dependency and issue rule to determine parallel instructions. Then calculate execution time by Equation 6. If it is a out-of-order issue superscalar processor, check data dependency and issue rule to schedule instruction into different function units. Then Calculate execution time by Equation 7.

4 Experiment

To test the accuracy of the proposed estimation model, we have run three examples on three processors. The three example are elliptical filter [10], medical system [9], and MPEG decoder [3]. The three processors are SPARC [1] [2], Intel Pentium [8], and PowerPC [7].

The elliptical filter contains a few basic blocks and most of its statements are inside one basic block. The medical system contains many basic blocks (more than thirty) and each basic block only contains a few statements. The MPEG decoder has large number of basic blocks and statements.

SPARC is a pipeline processor with four pipeline stages. Intel Pentium is an in-order issue superscalar processor with five pipeline stages. It can issue at most two instruction per clock cycle.

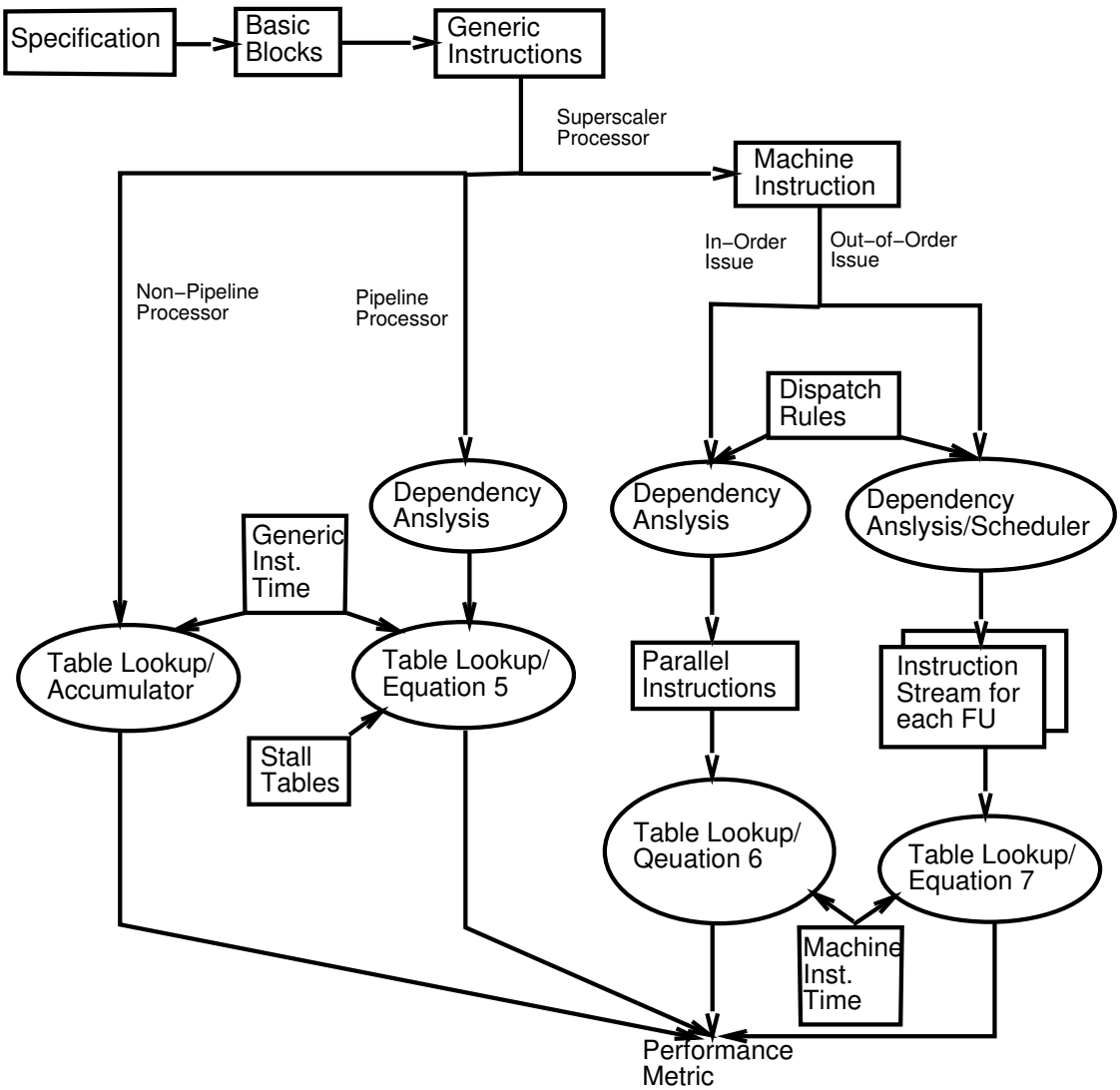


Figure 9: Software Performance Estimation Flowchart.

PowerPC 604 is a out-of-order issue superscalar processor with six pipeline stages. Even though it has six executing function units, two single cycle integer unit, one multiple cycle integer unit, one floating point unit, one load/store unit, and one branch unit, it can only issue four instructions per cycle at most.

	SPARC			Pentium			PowerPC 604		
	est. perf.	actual perf.	error	est. perf.	actual perf.	error	est. perf.	actual perf.	error
Elliptic Filter	887	868	2.1%	187	171	9.3%	473	431	9.7%
Medical System	792	754	5.0%	424	434	2.3%	574	571	0.5%
MPEG Decoder	428.5K	466.2K	–8.0%	169.2K	165.3K	2.3%	139.8K	128.8K	8.5%

Table 2: Experiment Results.

Table 2 shows the estimated performance and real performance of these three program on three processors. The performance is measured by clock cycle. To get the real performance, we first translate the SpecCharts [6] specification of these three system into C language. Then the C programs are compiled into machine instruction of each processors without optimization. The real performance is calculated based on these machine instructions assuming no cache miss.

From Table 2, we show that our estimation model can produce an accurate result on different application types and different processors. All estimation error are less than 10 percent.

5 Conclusion and Future Work

In this report, we have shown the software performance estimation that can produce accurate result with a few resource and computation time. This flexible approach was very suitable for design tools of system level hardware/software codesign.

Compiler optimization and cache miss are two issues about software estimation that are not counted in this estimation. When implementing the specification in software, most program will be compiled with optimization option. The speedup of optimized code depends on application, compiler, and processor. A simple solution is to find out the statistic speedup ratio by running many examples. The estimated performance can multiply this ratio to get the optimized performance metrics. This approach has been used in [4]. But it is very hard to find a good ratio for wide range of applications.

Another possible approach is to profile the optimization in technology file. For the most common

optimization technique, put variables in register, the technology can has another version that assumes variables are in registers. The optimization ratio can be approximated by the ratio of register number and live variable number. This accuracy and techniques for profiling other optimization method need further researches.

Cache miss has similar effect as optimization and can be approximated by a ratio. This simple solution has the same problem as optimization's. To accurately estimate the cache miss number, the system should have some sort of mapping between variable and memory address. With this information, the memory address distance between instruction can be calculated and cache miss can be predicted more precisely.

6 Acknowledgements

The authors would like to thank Jie Gong and Alfred Baehrenz Thordarson for providing the specification of experiment applications. We also want to thank Jin-Her Lin and Yirng-An Chen for compiling the specification into Pentium and PowerPC machine code.

References

[1] *CMOS BiCMOS Data Book*, Cypress Semiconductor, 1989

[2] M. Slater ed., *A Guide to RISC Microprocessor*, Academic Press, 1992

[3] A.B. Thordarson and D.D. Gajski, *Comparison of Manual and Automatic Behavioral Synthesis on MPEG Algorithm*, Technical Report ICS-05-09, UC-Irvine, 1995

[4] J. Gong, D.D. Gajski, and S. Narayan, *Software Estimation from Executable Specification*, Technical Report ICS-93-5, UC-Irvine, 1993

[5] D.D. Gajski, F. Vahid, S. Narayan, and J. Gong, *Specification and Design of Embedded Systems*, ch. 7.3, 1994

[6] D.D. Gajski, F. Vahid, S. Narayan, and J. Gong, *Specification and Design of Embedded Systems*, ch. 3.5, 1994

[7] *PowerPC 604, RISC Microprocessor User's Manual*, Motorola Inc., 1994

[8] *Pentium Family User's Manual*, Intel Corporation, 1994

[9] A. Wu, *A Microprocessor-based ultrasonic system for measuring bladder volumes*, Master Thesis in Electrical and Computer Engineering at University of Arizona, Tuscon, 1985.

[10] N. Dutt and C. Ramachandran, *Benchmarks for the 1992 high level synthesis workshop*, UC Irvine, Dept. of ICS, Technical Report 92-107, 1992.