

An Instruction Throughput Model of Superscalar Processors

Tarek M. Taha

Department of Electrical and Computer Engineering
Clemson University
Clemson, SC 29631, USA
tarek@clemson.edu

D. Scott Wills

School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, GA 30332, USA
scott.wills@ece.gatech.edu

Abstract

With advances in semiconductor technology, processors are becoming larger and more complex. Future processor designers will face an enormous design space, and must evaluate more architecture design points to reach a final optimum design. This exploration is currently performed using cycle accurate simulators that are accurate but slow, limiting a comprehensive search of design options. The vast design space and time to market economic pressures motivate the need for faster architectural evaluation methods.

The model presented in this paper facilitates a rapid exploration of the architecture design space for superscalar processors. It supplements current design tools by narrowing a large design space quickly, after which existing cycle accurate simulators can arrive at a precise optimum design. This allows a designer to select the final architecture design much faster than with traditional tools. The model calculates the instruction throughput of superscalar processors using a set of key architecture and application properties. It was validated with the SimpleScalar out-of-order simulator. Results were within 5.5% accuracy of the cycle accurate simulator, but executed 40,000 times faster.

1 Introduction

With transistor counts per processor expected to cross a billion soon [1], the design possibilities for these systems become enormous. A variety of architectural options for utilizing a billion transistor processor have been presented [2], ranging from fully reconfigurable processing grids [3] to large superscalar uniprocessors [4]. The task of architecture design becomes significantly more challenging with the processor design space expanding rapidly and with the possibility of building increasingly complex systems. Many more design options need to be evaluated to choose a final processor design.

The key objective in most architecture designs is increased performance. IPC (instructions per cycle) is gener-

ally used to define the effectiveness of a processor architecture. During the design of a microarchitecture, the IPC is generally measured using cycle accurate simulators. These tools mimic the cycle by cycle behavior of a processor running an application binary in great detail and produce very accurate results (depending on the level of detail implemented). As a rule of thumb, the execution of several billions of instructions needs to be evaluated to accurately assess the performance of a processor. Given the high level of detail implemented in simulators and the long program runs needed for accurate evaluations, these simulation runs tend to take a long time to evaluate.

For instance, consider the SimpleScalar out-of-order superscalar processor simulator [5]. Running on a 440 MHz UltraSPARC-IIi [6] based Sun Ultra-10 workstation, this simulator takes approximately 4.5 hours to evaluate a billion instructions. If a designer had to evaluate 2000 architecture configurations with 8 application runs of a billion instructions each, it would take over 8.21 years to complete all the simulations serially.

Processor design times have been increasing steadily over the years, with present designs taking as long as seven years to complete [7]. A significant portion of this time is taken up by microarchitecture design. Given the expected increases in processor complexity in the future, this design process is likely to get longer. Market demands however have been pushing towards special purpose commodity processors that are produced with a short turn around time. With current architecture design methods taking long to evaluate, it becomes challenging designing an optimum architecture in a short period of time.

The microprocessor design process would significantly benefit from a faster architecture performance evaluation methodology. For instance, a method that can assess an architecture in 0.1 seconds can complete the 2000 architecture evaluations described earlier in 26.7 minutes, as opposed to 8.21 years with current tools. These tools do not necessarily have to match the accuracy of currently used design tools (such as cycle accurate simulators). As long as they are significantly faster than current tools and reasonably accurate, they could cull the bulk of the processor de-

sign space and help narrow down architecture design parameters. Prevalent tools could then be used to further refine the architecture to a final design. This method would speed up the overall design process and allow a more extensive search of the design space in a shorter time.

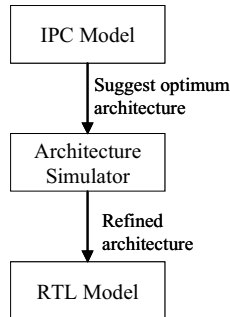


Figure 1: IPC Model in design flow

This paper presents an analytical model to calculate the instruction per cycle throughput (IPC) of superscalar processors. The model narrows down the architecture design space through a fast performance evaluation. As shown in Figure 1, cycle-based simulators can then be employed to hammer out a finished design. Overall design time is reduced because in this method, cycle-based simulators need to analyze a smaller design space. Hence fewer runs of these time-consuming simulations are needed.

The IPC model uses a set of statistical parameters to represent architecture and application properties. Application properties are extracted in a one-time step of program evaluation and can be used to evaluate a variety of architectures through the model. Application properties include the data and control dependencies in a program, instruction distributions, and memory usage patterns. Architecture properties provide a high level description of a processor and include information such as the superscalar width, the number of pipeline stages, and the functional unit mix of the processor.

This paper starts off with an investigation of related work in this area. Architecture and application input parameters to the model are then defined along with a description of how they are extracted. It then describes how these parameters are combined within the model to obtain a performance estimate. A validation section is then presented in which the MIPS R12000A [8] processor performance is estimated using the model and a modified version of the SimpleScalar out-of-order simulator. The model performance estimates are close to the simulator results.

2 Related Work

This work builds on a significant body of prior research in architectural performance modeling. Jouppi [9] suggested a simple processor performance evaluation model based on separately studying the non-uniformity in machine and program parallelism. Noonburg and Shen [10] proposed an analytical model that is based on an extension to Jouppi's concept of separate machine and program parallelism and the idea of program non-uniformity. They later suggested a more detailed statistical framework [11] based on Markov chains to model the interaction between machine and program parallelism. Kim et. al. [12] and Zhu et. al. [13] derived superscalar processor models based on queuing theory.

Michaud et. al. [14] developed a simple leaky bucket model of superscalar processor performance. The intent of the model was to develop a simple relation between the instruction fetch rate and branch prediction rate on superscalar processor performance. It assumed a perfect processor and cache with restrictions only on the fetch width and the branch prediction rate.

This paper extends Michaud's algorithm to model realistic superscalar processors. Extensions include modeling the functional units, reorder buffer, retirement bandwidth, and memory hierarchy. It also incorporates the idea of non-uniformities within systems as defined by Jouppi, as well as application and architecture modeling methods used in the other papers.

3 Model parameters

As mentioned earlier, cycle accurate simulators evaluate each instruction in the execution trace of a program individually. A large number of instructions have to be evaluated, making this process time consuming. The model presented in this paper saves time by estimating processor performance based on a key set of architecture and application metrics. These metrics are defined in separate architecture and application specifications. The architecture metrics describe the structure of key components in a processor – such as the superscalar width and the number of functional units. The application specification lists a set of parameters that characterize the overall behavior of the execution trace studied. These parameters attempt to capture the variation in program behavior in order to model program properties accurately. The following two subsections take an in-depth look at the architecture and application specifications.

3.1 Architecture Specification

The architecture specification defines the execution parallelism available in a processor by defining the characteristics of key structures within the processor. As this paper is centered on superscalar processors, features that are particular to this type of architecture are listed in the specification. Figure 2 lists the main architecture parameters used in the model. In this study, the architecture is divided into four main parts: the front end, issue stage, back end, and memory.

- | |
|---|
| <ol style="list-style-type: none"> 1. Superscalar width 2. Pipeline stages before issue stage 3. Central instruction window width 4. Issue width 5. Functional units. For each type of unit: <ol style="list-style-type: none"> a. Number of units b. Latency c. Throughput 6. Reorder buffer width 7. Retirement bandwidth 8. Cache information (size and latencies) |
|---|

Figure 2: Superscalar processor parameters.

The front end of the processor is responsible for fetching instructions from memory and supplying them to the issue stage. The main parameters used to define the front end are the superscalar width of the processor, the pipeline depth of the front end, the instruction fetch mechanism, and the branch predictor.

The front end dispatches instructions to the instruction window and the reorder buffer. Instructions cannot be dispatched from the front end if either the reorder buffer or the issue stage is full. Instructions issue from the instruction window out-of-order for execution, while they are retired from the reorder buffer in-order after completion.

The back end contains functional units to execute instructions from the issue stage. The number of functional units of each type is specified along with its throughput and latency. After completing execution, instructions are ready to retire from the reorder buffer.

The memory array consists of a hierarchy of caches. The size and access times (in terms of processor cycles) of these cache hierarchies are defined in the architecture specification.

3.2 Application Specification

Instead of simulating a full application binary, the analytical model uses key statistical properties of a program in

order to reduce architecture evaluation time. These properties are listed in the application specification and define the parallelism available within a program. Parameters in the application specification are extracted in a one-time step of running the program through a set of analysis tools. It is expected that the time saved by using the model will more than amortize the cost of program analysis.

Table 1: Application specification parameters.

Parameter	Description
Instruction histogram	Mix of instructions within program
Basic block size	Average number of instructions between control flow instructions
Issue dependence	Relation between instruction issue rate and instruction window occupancy
Retirement dependence	Relation between instruction retirement rate, reorder buffer occupancy, and completed instructions in reorder buffer
Branch prediction rate	Rate at which control flow outcomes are correctly predicted
Macro-block distribution	Variation in number of instructions between two consecutive mispredicted branches
Cache access pattern	Instruction and data cache access rate and miss rate

The first step in the analysis is to compile the program code. The program binary is then run through a set of modified simulators that extract statistical properties from the execution trace. Since program binaries are used in this analysis, the results obtained for each application depend on the compiler, source code, source language, input data set, and instruction set architecture used. Table 1 lists the parameters extracted from each application along with a description of the parameters.

The first four of the parameters listed in Table 1 are based on program properties only. These constitute the micro-architecture independent parts of the application specification. The other three parameters are micro-architecture dependent and are obtained for a specific set of micro-architectural structures. The specification may list a program property for several varieties of a micro-architectural structure. For instance, several branch prediction rates could be specified corresponding to different branch predictors or several cache hit rates specified for different cache hierarchies. These different values can all be collected through one simulation run using a simulator that evaluates a variety of caches or branch predictors in parallel.

Control dependencies determine how many useful instructions are visible to a processor at a time. A simple measure of control dependencies is the frequency of mispredicted branch instructions within a program – a higher

frequency of mispredicted branches corresponds to a lower amount of control parallelism. The frequency of mispredicted branch instructions is captured in a single term defined as the macro-block size – the average distance between two consecutive mispredicted branches. If the branch prediction rate is bp and the average basic block size is Bb , then an average macro-block is $(Bb \cdot bp)/(1-bp)$ instructions long.

Data dependencies determine an upper limit of how many instructions can be executed from a program. In this study, data dependencies are measured using two metrics: issue dependence and retirement dependence. The issue dependence determines how many of the instructions in the issue stage of a processor are ready to issue. It is expected that as the occupancy of the issue stage grows, the average number of instructions ready to issue will increase as well. This is because more candidate operations for issue will be visible. This property of programs is measured using a set of experiments where the issue buffer size of a processor is constrained in a perfect processor (all other components unconstrained). The issue buffer size is varied and the processor issue rate is measured. Figure 3 shows the result of this experiment.

The retirement dependence determines the order in which instructions in the reorder buffer complete – and thereby, the maximum retirement rate for a program. Completed instructions are retired out of the reorder buffer in program order. The retirement dependence is quantified similar to the issue dependence using a simulation of a perfect machine, except that in this case the reorder buffer size is constrained.

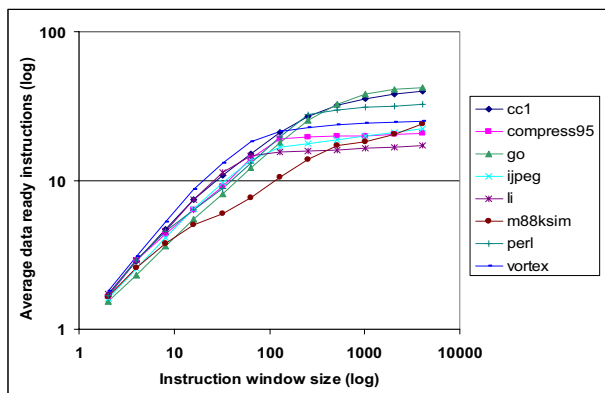


Figure 3: Data dependence plots for SPECint95.

Cache hit rates vary with programs and the memory configurations. As we are not aware of any good cache hit rate models, cache access patterns are determined from simulation for specific memory configurations. The hit rates for these configurations are then used in the IPC model. Access patterns can be measured through simulation for different cache configurations simultaneously. This data is collected for both instruction and data caches.

4 IPC Model Process

In this model, the throughput of a processor is represented as a series of macroblocks executed over time. Figure 4 illustrates this idea, with each shaded region representing a single macroblock of instructions. The instruction throughput of a processor can then be approximated as simply the rate in which an average macroblock of instructions is executed:

$$IPC = \frac{[\text{Number of instructions in macroblock}]}{[\text{Time to execute macroblock}] + [\text{Pipeline refill penalty}]}$$

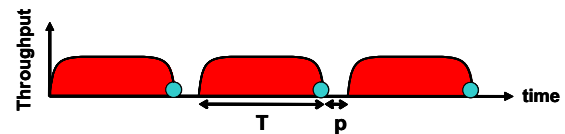


Figure 4: Execution represented as a series of macroblocks.

The last instruction in each macroblock is a mispredicted branch instruction (shown as circles in Figure 4). Since these trigger a flush in the front end pipeline of a processor, each macroblock ends with a period in which no instructions are executed (although other instructions may execute during this period in an out-of-order processor, this assumption simplifies the model significantly). The period of no execution is the pipeline refill penalty (same number of cycles as the number of pipeline stages in the front end).

In the IPC equation above, since the macro-block size and the pipeline refill penalty are specified in the model inputs, the only unknown is the time to execute a macro-block. This time is determined using an iterative algorithm that studies the execution of one full macroblock of instructions through the processor. It uses the following state variables to track the number of instructions in different stages of the processor in each iteration:

- C_i : Instruction yet to be fetched from cache
- W_i : Instructions in instruction window
- R_i : Instructions in reorder buffer
- $P_{n,i}$: Instructions in stage n of effective functional unit

Each parameter listed above has a subscript i to denote the algorithm iteration count. At the start of the algorithm, the full macroblock of instructions (M long) lies in the instruction cache, while rest of the processor is assumed empty. Thus the initial conditions are:

- $C_0 = M$
- $W_0 = 0$
- $R_0 = 0$
- $P_{n,0} = 0$

All the functional units in a processor are modeled as a unified effective unit with an effective latency, L_{eff} , repre-

sentative of all the units. The number of instructions in each pipeline stage of this effective functional unit is defined by the parameters $P_{n,i}$, where $0 \leq n < L_{\text{eff}}$. The effective latency is defined based the instruction mix of the application, and the multiplicity, latency, and throughput of the individual functional units. For a detailed description of how the effective functional unit is defined, please refer to [15].

During each iteration F_i instruction of the macroblock are fetched from the instruction cache and placed in the instruction window and reorder buffer. E_i of instructions from the instruction window are issued for execution, while G_i are retired from the reorder buffer. In addition, instructions in each stage of the effective functional unit move forward to the next pipeline stage. These three values (F_i , E_i , and G_i) are determined based on application properties and the state variables listed above. Thus the following equations define the algorithm iteration equations:

<i>Processor stage</i>	<i>Iteration function</i>
Cache:	$C_{i+1} = C_i - F_i$
Instruction window:	$W_{i+1} = W_i + F_i - E_i$
Reorder buffer:	$R_{i+1} = R_i + F_i - G_i$
Functional units:	$P_{0,i+1} = E_i$
	$P_{n+1,i+1} = P_{n,i}$

Fetching stops once the full macro-block is fetched. The algorithm ends when a full macroblock of instructions is issued from the instruction window. Since the assumption that the reorder buffer is empty initially is not necessarily true, two algorithm runs are carried out. The reorder buffer occupancy at the end of the first run is used as the starting occupancy of the reorder buffer for the second run. Thus the second run gives a better indication of the macroblock execution time.

It is important to note that this algorithm process is not the same as a cycle accurate simulation. This is because only the number of instruction in different parts of the processor is tracked in the algorithm. In fact this number can be fractional. No information about specific dependencies is tracked (as would be in a simulation). This model tries to determine the average rate of movement based on average program and processor properties. The instruction transfer rates, F_i , E_i , and G_i are defined in following sub-sections.

4.1 Fetch Rate

The front end of a superscalar processor is responsible for fetching instructions and ultimately supplying them to the instruction window and reorder buffer. In this study, it is assumed that the only bottleneck in the front end is presented by the instruction fetch rate. This rate is approximated as:

$$\text{Fetch_rate} = \min(\text{basic block size, superscalar width})$$

The number of instructions fetched at any given iteration in the algorithm is limited by the fetch rate above, the number of instructions from the macro-block remaining in the cache, and the remaining capacity in both the instruction window and reorder buffer. Thus the number of instructions fetched in any given iteration is:

$$F_i = \min(\text{Fetch_rate}, C_i, \text{instruction_window_size} - W_i, \text{reorder_buffer_size} - R_i)$$

4.2 Execution Rate

Data dependencies between instructions in the issue buffer prevent all the instructions from being ready to issue simultaneously. Instructions are ready when their input data values are generated and are then issued when a functional unit becomes available to execute them. The average fraction of instructions that are ready to issue is determined using the issue dependence properties of the application – a central instruction window with W_i operations has on average $D(W_i)$ instructions ready to issue. The actual issue rate in the model is however capped by the fetch bandwidth because a processor cannot have a sustained issue rate higher than its average fetch rate. Thus the number of instructions ready to issue in the model can be defined as:

$$I(W_i) = \min(D(W_i), \text{fetch_rate})$$

Functional unit resource conflicts can limit the fraction of these $I(W_i)$ instructions that may issue for execution. The match between the instruction mix of the program run and the functional unit mix of the processor determine the fraction of these ready instructions that actually issue. This mix is defined by the following variables:

F_C : classes of functional units in processor.

F_n, L_n, T_n ; multiplicity, latency, and throughput of functional units of class n , $0 \leq n < F_C$.

t_n : fraction of instruction in program of class n , $0 \leq n < F_C$.

Assuming a uniform mix of instructions within the program, a window with $I(W_i)$ ready instructions will have $t_n I(W_i)$ instructions of class n waiting to issue. Given that there are only F_n functional units of this class, only $\min(t_n I(W_i), F_n)$ instructions can issue.

The distribution of instructions actually issued per iteration needs to be the same as the mix of instructions in the program. This is because each algorithm iteration represents an average case – therefore, without the same distribution of instructions, there will be an uneven execution of operations of different classes. If the fraction of instructions that can be issued per class is:

$$I_n = \frac{\min(t_n I(W), F_n)}{t_n I(W)}, \text{ for } 0 \leq n < F_C$$

then the fraction actually issued is $I_f = \min(I_0, I_1, \dots, I_{F_C-1})$. Thus $I_f I(W_i)$ instructions are issued per cycle for execution.

This issue rate is based on the fact that a window of W_i instructions has on average $D(W_i)$ ready operations. We found that tracking the variation in $D(W_i)$ produces a better estimate of the issue rate. The variation measures the probability of there being n ready instruction in the window with n swept from 1 to W_i – instead of using an average value of $D(W_i)$. This entails more calculations, but generates better results. With multicycle functional units, an effective instruction window size has to be used in the algorithm based on execution latencies and the issue buffer size (for more details on this, please refer to [15]).

4.3 Retirement Rate

The reorder buffer keeps track of the sequential order of the instructions in the processor. Completed instructions at the end of the buffer are retired in program order. The buffer is useful during interrupts and branch mispredictions, so that all instructions beyond a certain point can be flushed without affecting the processor state.

Instructions in the reorder buffer could be in one of three states: i) in the central window waiting to be issued to a functional unit, ii) executing in a functional unit, or iii) completed and waiting to be retired (graduated). The number of instructions in the reorder buffer waiting to retire is given by the following equation:

$$\text{Completed instructions}_i = R_i - W_i - \text{sum}(P_{n,i})$$

where R_i is the reorder buffer occupancy, W_i is the number of operations in the instruction window, and $\text{sum}(P_{n,i})$ is the total number of instructions executing in the functional units. A portion of the completed instructions in the reorder buffer can be retired each cycle. This is provided retirement dependence data, and is given by $\text{Ret}(\text{completed instructions})$. As this rate is limited by the retirement bandwidth, the number of retired (graduated) instructions is given by:

$$G_i = \min(\text{Ret}(\text{completed instructions}), \text{retirement bandwidth})$$

4.4 Memory

The data movement functions considered so have not incorporated the effect of cache misses. This section looks at how a processor with a realistic cache hierarchy can be modeled.

Since analytical models for cache hit rates based on program properties are not available, the application specification provides cache hit rates for different memory configurations. The architecture specification provides memory access times.

We found that the best cache model was to incorporate data cache misses into the algorithm and add a separate instruction cache miss CPI component to the algorithm gen-

erated CPI. Data cache misses are incorporated into the algorithm by increasing the latency of memory operations to include data cache access times. Thus the CPI is calculated as:

$$\text{CPI} = \text{CPI}_{\text{processor with real data cache}} + \text{CPI}_{\text{instruction_cache}}$$

5 Validation

The model was verified with a modified version of the SimpleScalar [5] out-of-order simulator. The simulator was modified to incorporate a central instruction window, so that all un-issued instructions are pooled in this buffer. The original RUU based window within the simulator was used as a reorder buffer. The simulator was configured to the MIPS R12000 [8] processor for the validation because the specifications of this processor are one of the most widely published. The R12000 is a 4-issue superscalar processor with a reorder buffer of 48 entries. It has three instruction queues with 16 entries per queue, two integer units, two floating point units, and one memory unit.

SPEC95 integer and floating point applications along with Mediabench applications were used for validation. The test input sets were used for the SPEC95 benchmarks and up to 100 million instructions were simulated.

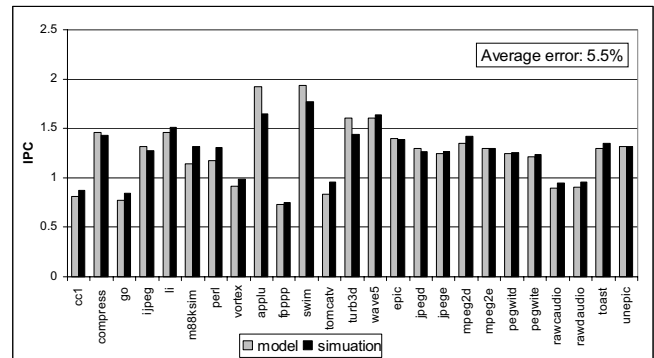


Figure 5: Model versus simulator results.

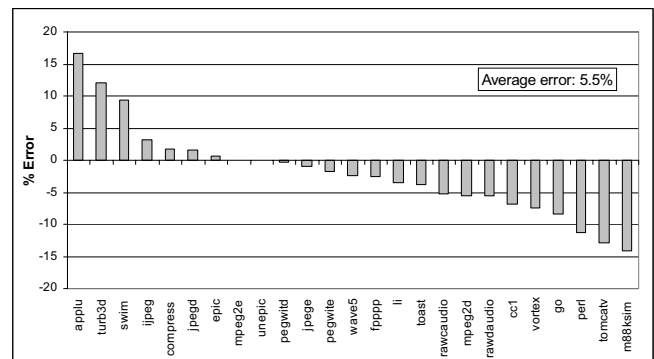


Figure 6: Model error distribution.

The results of the IPC model versus cycle accurate simulations for the MIPS R12000 processor are shown in figure

5. Figure 6 shows the errors for each of these runs. The average error was 5.5% over all the benchmark applications.

The IPC model took 1 second to evaluate all the configurations, compared with 11 hours for the simulator. This corresponds to a speedup of over 40,000 times with an error of about 5.5%. Figure 7 shows a comparison between the run times. The run time of the algorithm per evaluation was approximately 0.04s while the simulator was 30 minutes. However the simulator run time is dominated by computation while the algorithm is dominated by the time to load model input files from disk – the algorithm kernel took only 0.001s. This amounts to a speed up of approximately 1.6 million times compared to the simulator for the core calculations.

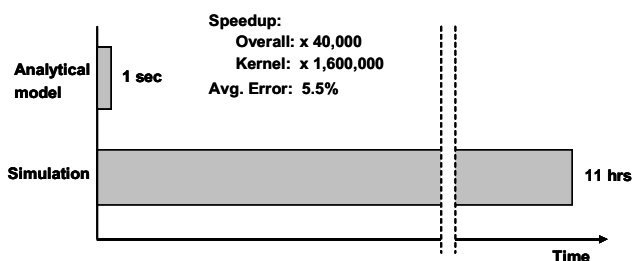


Figure 7: Run time of the model versus simulator.

6 Conclusion

An analytical model has been presented to evaluate the instruction throughput performance of superscalar processors. The model defines applications and processors using key properties of each. Application definition parameters define the control parallelism, data parallelism, and memory access pattern of the application and are extracted in a one-time step through an analysis tool. In carrying out evaluations, the model considers the non-uniformities within the system to improve performance estimates. The main results of this model are that:

- Overall model results were within 5.5% of simulation runs using the SimpleScalar simulator when modeling the MIPS R12000A processor with the SPEC95 and MediaBench application sets.
- The model evaluates architectures 40,000 times faster than the SimpleScalar simulator. The model kernel shows a speedup of over 1.6 million times compared to simulators.

Processor design times are growing impractically long as design complexities increase. Current design methods using cycle-accurate simulators, although accurate, take a long time to generate performance estimates. Given market demands, companies need to churn out new processor designs in a short amount of time. It can be expected that designing

with cycle-accurate simulators will limit the number of evaluations an architect can carry out in a fixed time. This could possibly hinder an architect from achieving an optimal processor design.

The instruction throughput model presented produces close approximations to cycle-accurate simulations at fraction of the evaluation time. Validation against simulation runs of the MIPS R12000A processor show an accuracy within 5.5% of the simulator with a kernel speed up of about 1.6 million times. Although this model is not meant to replace cycle accurate simulators, it can be used at an early design stage to cull a large architecture design space. The model allows an architect to evaluate more processor architecture alternatives with a larger base of benchmark application in a given time limit. This allows a more efficient and through search of the processor design space and aids in designing the best architecture needed for a given application.

References

- [1] A. Cataldo. Intel describes billion-transistor four-core Itanium processor. *EE Times*, October 16, 2002. <http://www.eet.com/story/OEG20021015S0036>
- [2] D. Burger and J. R. Goodman. Billion-Transistor Architectures. *IEEE Computer*, 30(9):46-48, September 1997.
- [3] K. Sankaralingam, R. Nagarajan, D.C. Burger, and S.W. Keckler. A Design Space Evaluation of Grid Processor Architectures. In *International Symposium on Microarchitecture*, 2001.
- [4] Y. N. Patt, S. J. Patel, M. Evers, D. H. Friendly, and J. Stark. One billion transistors, one uniprocessor, one chip. *IEEE Computer*, 30(9):51-57, September 1997.
- [5] D. Burger and T. A. Austin. The SimpleScalar tool set, version 2.0. Technical Report #1342. University of Wisconsin-Madison Computer Science, June 1997.
- [6] K. B. Normoyle, M. A. Csoppenszky, A. Tzeng, T. P. Johnson, C. D. Furman, and J. Mostoufi. UltraSPARC-IIi: expanding the boundaries of a system on a chip *IEEE Micro*, 18(2):14-24, April 1998.
- [7] S. S. Mukherjee, S. V. Adve, T. Austin, J. Emer, P. S. Magnusson. Performance simulation tools. *IEEE Computer*, 35(2):38-39, Feb. 2002.
- [8] I. Williams. An Illustration of the MIPS® R12000TM Microprocessor and OCTANE System Architecture. White paper, 1999. <http://www.sgi.com/products/remarketed/octane/octane.pdf>
- [9] N. P. Jouppi. The Nonuniform Distribution of instruction-level and machine parallelism and its effect on performance. *IEEE Transactions on Computers*, 38(12):1645-1658, December 1989.

- [10] D. B. Noonburg and J. P. Shen. Theoretical modeling of superscalar processor performance. In International Symposium on Microarchitecture, 1994.
- [11] D. B. Noonburg and J. P. Shen. A Framework for statistical modeling of superscalar processor performance. In International Symposium on High Performance Computer Architecture, 1997.
- [12] H. J. Kim, S. M. Kim, and S. B. Choi. System performance analyses of out-of-order superscalar processors using analytical method. IEICE Transactions On Fundamentals of Electronics communications and computer sciences, E82A(6): 927-938, June 1999.
- [13] Y. Zhu and W. F. Wong. Modeling architectural improvements in superscalar processors. In The Fourth International High Performance Computing in the Asia-Pacific Region, 2000.
- [14] P. Michaud, A. Sez nec, and S. Jourdan. Exploring instruction-fetch bandwidth requirement in wide-issue superscalar processors. In International Conference on Parallel Architectures and Compilation Techniques, 1999.
- [15] T. M. Taha. A Parallelism, Instruction Throughput, and Cycle Time Model of Computer Architectures. Georgia Institute of Technology, Ph.D. Thesis, November 2002. Available at <http://www.ces.clemson.edu/tarek/taha-thesis.pdf>.