Sensitivity Analysis of a Superscalar Processor Model

Y. Zhu **W. F. Wong**

Department of Computer Science
School of Computing
National University of Singapore
3, Science Drive 2
Singapore 117543
Email: wongwf@comp.nus.edu.sg

Abstract

Superscalar processors obtain their performance by exploiting instruction level parallelism in programs. Their performance is therefore limited by characteristics of programs and the design of the processor. Due to the complexity involved, estimating the performance of any superscalar processor design is a difficult task. Quick prediction of performance improvement arising from architecture modifications is even more difficult. In this paper, a model of superscalar processors using a network of *Multiple Class and Multiple Resource Queue*s is described and studied. In this model, we are able to model and study instruction classes, instruction dependencies, the cache, the branch unit, the decoder unit, the central instruction buffer, the functional units, the retirement buffer, the retirement unit and instruction issue policy in an integrated manner. This model has been verified against measured performance and has shown an average error of 5%. From this starting point, we applied sensitivity analysis on the model and studied *qualitatively* three important classes of improvements one can make to a superscalar processor's design. The insights we derived show how a good model can be used to accurate pinpoint bottlenecks and assign relative importance to them. This will in turn guide development efforts.

Keywords: Superscalar processors, queuing theory

1 Introduction

Superscalar processing is the de-facto standard architecture for commerical off-the-shelf microprocessor. It is by nature a complex scheme involving dynamic instruction issue performed by hardware. Modeling such processors is therefore also a complex task. However the advantages of having a good model for such processors is numerous. Such a model can be used to design future processors as well as a means to gain insight into applications and their behavior under superscalar processing. Previous work on modeling this class of machines includes those of Austin and Sohi [Austin and Sohi, 1992], Lam and Wilson [Lam and Wilson, 1992], and Dubey et al [Dubey et al,. 1994] which are based on probability theory. These studies are generally experimental studies which attempt to obtain performance by simulation. Noonberg and Shen [Noonberg and Shen, 1994] proposed the use of probability vectors as a modeling tool. However, none of the above use queuing theory in their modeling. In our earlier work[Zhu and Wong, 1997], a queuing model was proposed for modeling superscalar processors based on a network of multiple class and multiple resource (MCMR) queues. In this theoretical model, a trace from a benchmark is analyzed only once to compute its instruction dependencies, classifications and distributions of instruction. These features essentially form an abstraction for the characteristics of the trace. Each processor specification is then analyzed by modeling the fetch unit, branch unit, central window, functional units and retirement unit as the *resource set*. In another work of ours[Zhu and Wong, 1998], instruction dependency and saturation states, i.e. situations where certain requests for resources have to be blocked, were considered. The accuracy of the model was verified against a cycle-by-cycle simulator. In this paper, we extend our previous works by studying three classes architecture modifications through sensitivity analysis. This current paper provides the detailed analysis of the results given in a shorter paper [Zhu and Wong, 2000]. This allows us to gain qualitative insight into the sensitivity of performance improvement with respect to these architecture changes thereby assessing the merits of these changes. Further, we believe a detailed exposition of the techniques we used allows for application of these same technique to other classes of architectures, such as EPIC [Schlansker and Rau, 1999].

Section 2 describes a general superscalar processor architecture that is modeled. Section 3 describes the architectural improvements that we studied. Section 4 states our MCMR model and shows its correspondence with the general superscalar architecture. Section 5 gives some analytical results of our model in particular with considerations on instruction dependency and saturation states. The verification of our model is described in Section 6. Section 7 gives sensitivity analysis and our main results. This is followed by a conclusion.

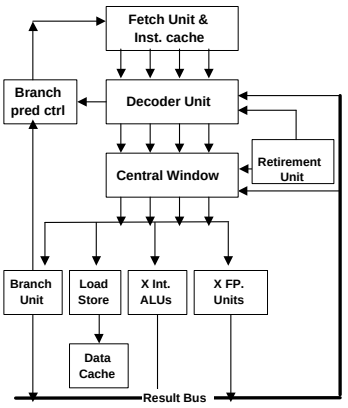

Figure 1: A Generic Superscalar Processor

2 Generic Superscalar Architecture and Common Modification

Early RISC processors relied on their compilers to order instructions so as to avoid pipeline conflicts. Superscalar processors use hardware to dynamically perform instruction reordering and instruction level parallelism are exploited further by executing instructions out of program order.

Fig. 1 shows the generic superscalar processor modeled in this paper. In many ways, it is similar to most of the current generation of superscalar processors such as the HP PA-8000, the Sun UltraSPARC, the MIPS R10000 and the DEC Alpha21164. The elements of the model are as follows.

The *fetch unit* working with the *instruction cache* feeds the processor with instructions. The *decode unit* consists of an instruction decoder, a branch prediction unit and reorder buffer with a register file. The reorder buffer is maintained as a first-in-first-out queue. Each entry in the reorder buffer is made up of a tag, the destination register number, the result and a valid bit. The reorder buffer preserves the program order as well as supports register renaming. As their names imply, the branch prediction unit is used to perform branch prediction while the decoder decodes instructions. In the *central instruction window*, instructions are queued up for issue. An instruction is ready for issue if (a) its inputs are available, (b) its destination register is available, and (c) a functional unit is available to execute it. The *functional units* of a superscalar processor are similar to those of earlier generation RISC processors. These are the branch unit, the load/store unit, the data cache and a number of computational units. These units execute instructions of the corresponding classes. The *retirement unit* updates the architectural registers with the computed results from the rename registers and removes entries associated with executed instructions from the reorder buffer in program order.

3 Common Architecture Modifications

In this paper, we studied the following three categories of common architectural modifications:

1. *Mechanisms to reduce dependencies.* Several mechanisms have been implemented to reduce the effect of instruction dependency on issue and retirement rates. Register renaming reduces anti and output instruction dependencies while in-order issue ensures antidependencies never occur. Some more aggressive mechanisms such as speculation execution have been proposed. These will not consider them in our study.

2. *Techniques to enhance the effectiveness of the instruction and data caches.* Highly associativity caches coupled with branch prediction are used to lower the instruction cache miss ratio. These increase the probability of finding the instructions following (correctly predicted) branches in the instruction cache. However, the cache miss penalty is not reduced. For the data cache, data prefetching schemes decrease the miss ratio by loading data needed in advance. Again, the cache miss penalty is not reduced. Current superscalar processors often work with cache whose small line size trades off miss ratio for lower hit latency and miss penalty.

3. *Improvements to the functional units.* To reduce the waiting time for instructions ready to be issued, one may either increase the number of functional units or reduce the number of pipeline stages of functional units.

4 The MCMR Model

A *multiple-class multiple-resource* (MCMR) system is a queuing system where there are several classes of customers, each to be serviced by a particular set of resources. There have been a number of studies on MCMR systems. Assuming a simple distribution for the arrival process, Lazowka et. al. [Lazowska et al., 1984] described a simple method to estimate a MCMR system's performance. Some steady state solutions were also reported in works by Matta and Shankar [Matta and Shankar, 1995] and Jain [Jain, 1991]. To model the superscalar processor described above, we used a *network* of MCMR systems shown in Fig. 2.

In our MCMR model of a superscalar processor, instructions are modeled as customers which belong to one of several classes. The various functional components of the processors are the resources. Let R be a set of *resources* and C a set of *customer classes*. Each class in C represents a class of customers that requires a particular set of resources. Specifically, each class c, $c \in C$, its members require some set, R_c, of resources, $R_c \subseteq R$. Furthermore, a class c customer requires a number of units, denoted by $req_c(r)$, of each resource, $r \in R_c$. The set of customers who request

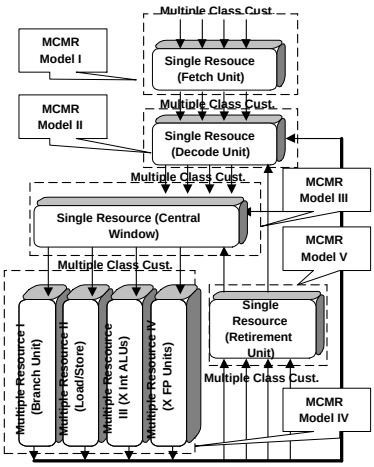

Figure 2: A Network MCMR Queuing Model of A Superscalar Processor

resource r is denoted C^r. In our model, each instruction belongs to exactly one customer class and before a member of an instruction class can be executed, a number of functional units have to be acquired. An arriving class c customer is *blocked at a resource* r if and only if $req_c(r)$ exceeds the amount of the resource that is currently available. An arriving class c customer is *blocked* if and only if it is blocked at any r.

The stages of a superscalar pipeline are modeled as separate MCMR queuing models with different numbers of resources and classes as well as arrival and departure rates of customers. The instruction fetch stage and the branch and decode stage are modeled as MCMR queues (Model I and II of Fig. 2 respectively) which have multiple customer classes and a single resource. The customer classes correspond to the different instruction classes. The single resource represents the fetch unit and the branch and decode unit collectively. The process of an instruction leaving the branch and decode unit is modeled by another MCMR component (Model III). The process of instructions leaving the central window for functional units for execution is modeled by yet another MCMR component (Model IV). Here, the functional units are the resources. Every instruction needs only one functional units. At the retirement stage, a MCMR queue models the retirement unit (Model V) as the single resource.

Some general analytical measures for the MCMR model were derived by Jain [Jain, 1991] and Lazowska [Lazowska et al., 1984]. However, these results assume sufficient capacity. This may not be true for the central instruction window or retirement unit. In the next section, we will give a different set of equations for throughput and queue length. These measures are the *processing capacity*, the *utilization*, the *residence time*, and the *queue length*. Details of the derivation are shown in appendix.

5 Analysis of Instruction Dependencies and Saturation States

We shall now analyze the effect of instruction dependencies on the central window, the retirement unit and the decoder. Let $\mathcal{P}$ be a benchmark program and $\mathcal{P}(\mathcal{I})$ be the execution trace obtained by executing $\mathcal{P}$ with input $\mathcal{I}$. Let $u, v \in \mathcal{P}(\mathcal{I})$ be two instructions of $\mathcal{P}$ that are found in the execution trace of $\mathcal{P}$ on input $\mathcal{I}$. We use $u \xrightarrow{D} v$, read as '*v depends on u with a distance of D instructions*', to represent the relationship between u and v satisfying the following conditions:

1. v is either *flow*, *anti* or *output* dependent on u;
2. there are exactly D instructions in $\mathcal{P}(\mathcal{I})$ (exclusive) between u and v, and
3. if $u \xrightarrow{D} v$, then there is no $w \in \mathcal{P}(\mathcal{I})$ such that $w \xrightarrow{D'} v$ and $D' < D$.

v is the *current* instruction (or simply the instruction), while u is called its *antecedent*. To further indicate the exact type of dependency involved, we shall use $u \xrightarrow{D}_f v$, $u \xrightarrow{D}_a v$ and $u \xrightarrow{D}_o v$ to represent flow, anti and output dependencies, respectively. On the other hand, we write $u \not\longrightarrow v$ to mean that v *does not depend on* u. $\overline{D}$ in Table 1 is the *average dependency interval*, i.e. the average of all D's in all of the dependencies of $\mathcal{P}(\mathcal{I})$.

For all $u, v \in \mathcal{P}(\mathcal{I}), u \xrightarrow{\overline{D}} v$, $\overline{P}_{\text{dep}}$ designates the possibility that $\exists w \in \mathcal{P}(\mathcal{I}), w \xrightarrow{D} u$. For in-order processors, only flow and output dependencies are considered. For out-of-order issue, $\overline{P}_{\text{dep}}$ is computed using all the flow dependencies, plus some[1] output and anti dependencies. Let T_{dep} be the average time for an antecedent instruction to pass through the functional units and forward its results to the reorder buffer and central window and is given by

$$T_{\text{dep}} = (1 + \overline{P}_{\text{dep}}) \times \sum_{i \in \text{Units}} (t_i \times S_i + 1) \qquad (1)$$

where t_i is the average time for an antecedent instruction to pass through a functional unit i, i.e. the functional unit's latency, and therefore is the number of pipeline stages of a functional unit, except for load/store unit. For the latter's t_i, the data cache miss penalty and hit rate (typically 90%) needs to be considered. S_i is the fraction of all instructions executed by functional unit i. Competition between instructions arriving at the same functional unit can also increase t_i.

5.1 Analysis of the Central Window

In our model, the central window works as the instruction buffer. Instructions stay in the central window after they are decoded until they are issued.

[1]Since we know nothing about how much output dependencies are actually eliminated by register renaming, we used an arbitrary value of 50% to characterize imperfect register renaming.

Benchmark	go	compress	m88ksim	applu	fpppp
IEU1	42.7%	46.3 %	42.7 %	20.7 %	21.7 %
IEU2	12.8%	5.1%	12.9 %	3.8 %	1.0 %
FPU1	0.0%	0.3%	0.0 %	3.9 %	0.1 %
FPU2	0.1%	0.6 %	0.0 %	20.2%	20.0 %
FPU3	0.0%	0.0 %	0.0 %	4.0 %	2.8 %
BRU	15.9%	6.1%	19.6 %	2.7 %	1.0 %
LSU	28.5%	41.5%	23.8 %	48.2%	53.4 %
PDU	100.0%	100.0%	100.0%	100.0%	100%
$\overline{D}$	1.9	1.9	1.8	2.1	2.0
$\overline{I}_{\rm br}$	5.3	15.4	5.4	36.0	99.0
$\overline{P}_{\rm dep}$	0.4	0.4	0.4	0.2	0.4
$T_{\rm dep}$	3.0	2.7	3.2	3.4	3.8
p	0.21	0.19	0.19	0.22	0.24
q	0.63	0.72	0.72	0.66	0.75
Tot. Inst.	17,959,093,034	7,518,280	595,293,999	19,744,107,711	2,705,422,029

Table 1: Parameters of the benchmarks used in our analysis.

Some instructions are discarded because of incorrect branch prediction. In addition to instruction cache misses, the decoding rate also suffers from these mispredictions. We derive our set of formulas from Pyun's [Pyun et al., 1998] work.

For in-order issue policy, $I_{\rm np}$, the issue rate of the central window with perfect branch prediction is as follows.

$$I_{\rm np} = \sum_{k=1}^{N} \rho_k(I_0, I_1, ..., I_{W-1}) \times k$$

where ρ_k denotes the probability that k instructions are issued from the window of size W. N is the number of functional units.

$I_{\rm np}$ can also be expressed as

$$I_{\rm np} = \sum_{k=1}^{N} p^{k(k-1)/2} \times \sum_{j=k}^{N} \left(\begin{array}{c} N \\ j \end{array} \right) q^j (1-q)^{N-j} \times k$$

where p represents the probability that the first instruction I_0 and i-th instruction I_i in the window are independent. q is the probability that an instruction in the window to be issued to an functional unit. Both p and q are assumed to be constant. Our formulas are simplifications of those found in Pyun et al.'s paper and the interested reader should refer to the original paper for insights into these formulas. The average issue rate is none other than q for a instruction window size of 1. The average value of issue rates at each position of the window is p.

We found that p and q varies significantly across the SPEC95 benchmarks running on SPARC processors though the two parameters can be measured easily. For our MCMR model, we have a separate p and q, represented by p_i and q_i for each instruction class i. Similarly, we define N_i as the number of functional units or resources that an instruction of class i can be issued to. Hence, $I_{\rm np}$ is re-defined as follows:

$$I_{\rm np} = \sum_{i} \sum_{k=1}^{N} {p_i}^{k(k-1)/2} \times \sum_{j=k}^{N_i} \left(\begin{array}{c} N_i \\ j \end{array} \right) q_i^j (1-q)^{N_i - j} \times k \quad (2)$$

After taking into consideration branching, we get the following expression for the average issue rate at the central window:

$$\mu_{\rm win} = [1 - (\text{Br. prob.} \times \text{Mispred.} \times \text{Br. Penalty})] \times I_{\rm np} \quad (3)$$

For a out-of-order processor, the instruction issue process can keep on searching for instructions after encountering an instruction involved in dependencies. By assuming that the pairs of instructions involved in dependencies are independent of one another, we have

$$\rho_k(I_0, I_1, ..., I_{W-1}) = P_k(I_0, I_1, ..., I_{W-1}) \times P_{\rm pipe}(k)$$

where

$$\begin{aligned} P_k(I_0, I_1, ..., I_{W-1}) &= P_{k-1}(I_0, I_1, ...I_{W-2}) \\ &\quad \times P(I_{W-1}) + P_k(I_0, I_1, ..., I_{W-2}) \\ &\quad \times (1 - P(I_{W-1})) \end{aligned} \quad (4)$$

and

$$P_{\rm pipe}(k) = \sum_{j=k}^{N} \left(\begin{array}{c} N \\ j \end{array} \right) q^j (1-q)^{N-j}$$

where $P(I_k)$ denotes the probability that instruction I_k is independent of its preceding instructions. This then allows us to compute $I_{\rm np}$ and $\mu_{\rm win}$.

5.2 Analysis of the Retirement Unit

Let $W_{\rm ret}$ denote *retire width*, i.e. the maximum number of instructions that can be retired in one cycle. The decoding rate decreases with increasing cache misses and branch mispredictions as instruction dependencies limit the retirement rate. Let's take the example of $W_{\rm ret} = 4$, and let the top four instructions at the retirement unit be u_1, u_2, u_3, u_4 say. These are grouped together for retirement provided there is no dependency between them. In general, if $\overline{D} < W_{\rm ret}$, dependencies will degrade the retire rate significantly. For example, suppose $u_1 \xrightarrow{\overline{D}} u_4$ where u_1 is a floating point instruction that finishes in 3 cycles and no

other dependencies exist among $u_1, ..., u_3$. The total time for retiring $u_1, ..., u_4$ is 4 cycles. However, there is a possibility of retiring more than 4 instructions during the 4 cycles since $u_5, ..., u_7$ can be retired together with $u_1, ..., u_4$ as long as they are not dependent on each other. The maximum retirement rate for $u_1, ..., u_7$ is therefore 7/4 instructions per cycle.

The above example illustrates how we can approximate the retirement rate by using mean value analysis. With $\overline{D}$ and T_{dep}, the time to retire $2 \times \overline{D}$ instructions is $1 + T_{\text{dep}}$ cycle when $\overline{D} < W_{\text{ret}}$. Therefore, for $\overline{D} < W_{\text{ret}}$, the average retirement rate of the retirement unit, where an in-order retire policy is implemented, is

$$\mu_{\text{ret}} = (2 \times \overline{D})/(1 + T_{\text{dep}}) \tag{5}$$

For $\overline{D} \geq W_{\text{ret}}$, the average retirement rate degradation is not significant. For example, for $u_1 \xrightarrow{\overline{D}} u_6$, where u_1 is a floating point instruction requiring three cycles and $W_{\text{ret}} = 4$, $u_1, ..., u_4$ are retired in cycle one, and u_5 is retired in cycle two. u_6 has to remain in the retirement unit until the fourth cycle when instructions $u_6, ..., u_9$ are retired together. Thus, 4 cycles are spent to retire 9 instructions. In fact, instead of u_6 which is dependent on u_1, the situation would be the same for any of $u_5, ..., u_7$, i.e. for $\overline{D}$ ranging from 4 to 7. In summary, for $W_{\text{ret}} \leq \overline{D} < T_{\text{dep}} \times W_{\text{ret}}$ and an in-order retire policy,

$$\mu_{\text{ret}} = (2 + \overline{D})/(1 + T_{\text{dep}}) \tag{6}$$

If $\overline{D} \geq W_{\text{ret}}$ and $\overline{D} \geq T_{\text{dep}} \times W_{\text{ret}}$, v does not need to wait for u which would have finished before v can be retired even if the policy is in-order retire policy. Therefore, the retirement rate is just the maximum retirement rate of W_{ret} instructions per cycle. Note that μ_{win} is usually greater than W_{ret} when a out-of-order issue policy is adopted.

5.3 Analysis of Decoder Unit

In the decoder unit, branch misprediction plays a role similar to that of instruction dependencies in instruction issue. Let W_{dec} denote the *decode width*, i.e. the maximum number of instructions that can be decoded in one cycle. Let $\overline{I}_{\text{br}}$ be the average number of (non-branch) instructions between two branch instructions (inclusive of one of them), T_{br} be the misprediction penalty time (the time taken to fetch and decode the correct instructions), $P_{\text{ins,miss}}$ be the instruction cache hit ratio, and $T_{\text{ins},p}$ be the instruction cache miss penalty. Again, let us start with an example. Suppose u_1 is a branch instruction and $u_{\overline{I}_{\text{br}}+1}$ is the immediately following branch instruction. If u_1 is mispredicted, the instructions following u_1 have to be re-fetched and re-decoded. Let T_{br} be the total time required for the instruction cache to reload as well as for the fetch and decode units to fill their pipelines again. If P_{prtd} is the probability of a correct branch prediction, then the average time to decode a branch and its following instructions is

$$1 \times P_{\text{prtd}} + (1 + T_{\text{br}}) \times (1 - P_{\text{prtd}}) \tag{7}$$

When instruction cache misses are also taken into consideration, the above equation is changed to

$$P_{\text{prtd}} + (1 + T_{\text{br}}) \times (1 - P_{\text{prtd}}) + N_{\text{dec}} \times T_{\text{ins},p} \times (1 - P_{\text{ins,miss}}) \tag{8}$$

where N_{dec}, the number of instructions decoded during the above time, is a function of $\overline{I}_{\text{br}}$. If $\overline{I}_{\text{br}} < W_{\text{dec}}$ and the second branch instruction $u_{\overline{I}_{\text{br}}+1}$ is mispredicted, then only $\overline{I}_{\text{br}}$ instructions are decoded during the period. Otherwise, if $u_{\overline{I}_{\text{br}}+1}$ is predicted correctly, W_{dec} instructions can be decoded. Therefore, the average decoding rate without overflow of the central window, $\mu_{\text{dec},n}$, is

$$\mu_{\text{dec},n} = \frac{C_1}{C_2 + C_3 \times T_{\text{ins},p} \times P_{\text{ins,miss}}} \tag{9}$$

where C_1 denotes $\overline{I}_{\text{br}} \times (1 - P_{\text{prtd}}) + W_{\text{dec}} \times P_{\text{prtd}}$, C_2 denotes $1 \times P_{\text{prtd}} + (1 + T_{\text{br}}) \times (1 - P_{\text{prtd}})$ and C_3 denotes $[\overline{I}_{\text{br}} \times (1 - P_{\text{prtd}}) + W_{\text{dec}} \times P_{\text{prtd}}]$. Further analysis of the decoder, including different expressions for $\mu_{\text{dec},n}$ under different circumstances, can be found in appendix.

Using bottleneck analysis, we approximate the overall throughput, measured in instructions per cycle, by

$$\min\{\mu_{\text{dec},n}, \mu_{\text{win}}, \mu_{\text{ret}}\} \tag{10}$$

5.4 Out-of-order Issue with Large Reservation Stations

Out-of-order issue may increase the overall performance substantially in some circumstances. The conditions are as follows:

1. Once the queues at the reservation stations are of enough length, the dependencies among the instructions in the queues could be resolved before the instructions are issued to the functional units. This is possible when there is sufficient capacity at the reservation stations.
2. To sustain a high issue rate, out-of-order issue should be used.
3. An issue width greater than 4 requires a high issue rate and also puts pressure on the caches. Therefore matching performance is expected of the caches.
4. Functional units should be complex yet fast so that the reservation stations do not overflow often.

Once these conditions are met, the retirement unit will no longer be the bottleneck even if out-of-order issue policy is implemented. This is because the dependencies are already resolved before the instructions are issued to the functional units as these instructions generally will wait for a substantial amount of

time in the longer queues. The overall performance is therefore

$$min\{\mu_{\text{dec},n}, \mu_{\text{win}}\} = \mu_{\text{dec},n} \tag{11}$$

The cost is larger capacity at the reservation stations. This added capacity is not necessarily for in-order issue or out-of-order issue with in-order retirement. In addition some mechanisms have to be implemented to handle overflow at the reservation stations.

If any of the above conditions is not satisfied, some instructions may be issued into functional unit before anti and output dependencies involving these instructions are resolved. Accordingly, they must wait for the previous instructions to finish at the retirement unit. In this case, the overall performance is $\min\{\mu_{\text{dec},n}, \mu_{\text{win}}, \mu_{\text{ret}}\}$.

6 Model Verifications

Five benchmarks from the SPEC95 suite, namely `099.go`, `124.m88ksim`, `120.compress`, `110.applu` and `145.fpppp` were used to verify the model. Parameters for our model were obtained via a cycle-by-cycle simulator[Loh, 1997]. This simulator, configured as an UltraSPARC, has been tested with the SPEC92 suite, and a strong *sample coefficient of determination*[2] of more than 0.91 has been observed between simulated cycle numbers and the actual execution time. In addition to processing the traces, the simulator also collected various parameters needed by the model. These are shown in Fig. 1.

The processor's components mentioned in Fig. 1 are two 64 bit integer ALUs (IEU 1 & 2), a floating-point divider/square root unit (FPU 1), a floating-point adder (FPU 2), a floating-point multiplier (FPU 3), a branch unit (BRU), a load and store unit (LSU) and a fetch and decode unit (PDU). Note that on the SPARC processor, all instructions must pass through the PDU.

For an in-order issue UltraSPARC, we measured the actual throughput (in instructions per cycle) for the SPEC95 benchmarks. The actual throughput for each benchmark is obtained by running and timing the benchmarks and then dividng the elapsed time by the number of instructions executed as reported by the simulator. The measured performance is then compared with our model's predictions and shown in Table 2. On average, a relative error of about 5.08% was achieved. This, we believe, demonstrated the accuracy of our model.

7 Sensitivity Analysis and Results

Confident that we have a good model, we now turn to using the model to produce qualitative insights into the working of superscalar processors. We achieve this by performing a sensitivity analysis to determine how performance is affected by the architectural modifications mentioned in section 3.

7.1 Performance Sensitivity to Dependency Reduction

The probability of the existence of a dependency[3], P_{dep} does not affect the decoding rate. However, it does affect μ_{ret}. The performance sensitivity to dependency reduction is accordingly defined as the derivative of μ_{ret} with respect to P_{dep}: $\frac{\partial \mu_{\text{ret}}}{\partial P_{\text{dep}}}$. The following expressions can be obtained from Eq. 1, Eq. 5 and Eq. 9 by noting that $\frac{\partial T_{\text{dep}}}{\partial P_{\text{dep}}} = 1$, $\frac{\partial \mu_{\text{ret}}}{\partial P_{\text{dep}}} = \frac{\partial \mu_{\text{ret}}}{\partial T_{\text{dep}}}$.

$$\frac{\partial \mu_{\text{ret}}}{\partial T_{\text{dep}}} = \frac{\partial \mu_{\text{ret}}}{\partial P_{\text{dep}}} = \frac{(2+\overline{D})(1+\sum_{i\in\text{Units}} t_i \times S_i)}{(1+T_{\text{dep}})^2} \tag{12}$$

when $\overline{D} < W_{\text{ret}}$ and

$$\frac{\partial \mu_{\text{ret}}}{\partial T_{\text{dep}}} = \frac{\partial \mu_{\text{ret}}}{\partial P_{\text{dep}}} = \frac{(2\overline{D})(1+\sum_{i\in\text{Units}} t_i \times S_i)}{(1+T_{\text{dep}})^2} \tag{13}$$

when $\overline{D} \geq W_{\text{ret}}$.

Given a processor, the t_i's are fixed. Therefore for two programs with similar instruction mix (i.e. similar S_i), performance is determined by P_{dep} which affects both T_{dep} and $\overline{D}$.

From another perspective, we can say that performance is very sensitive to dependency reduction. Perfect register renaming eliminates all anti and output dependencies. Thus, $\overline{D}$ reduces to $\overline{D}_f$, average flow dependency interval. Meanwhile, P_{dep} decreases to P_{flow}, the probability of flow dependency. Due to these two effects, we conclude that mechanisms to reduce dependencies, such as register renaming, is the most effective means of improving performance.

Though there is no expression for p directly in terms of P_{dep}, we may assume that both p and P_{dep} are equally sensitive to dependency variation. Then the sensitivity of μ_{win} to p is comparable with the sensitivity of μ_{ret} to P_{dep}. The sensitivity of μ_{win} to p can be obtained from $\frac{\partial \mu_{\text{win}}}{\partial p}$ where μ_{win} is obtained from Eq.3.

There is no general expression for since according to Eq. 2 $\frac{\partial \mu_{\text{win}}}{\partial p}$ as N_i is different for different instruction classes. For example, the following we get two different expressions for the sensitivities of floating point instructions and integer instructions with respect to p_i, assuming in-order-issue.

$$[1-(\text{Br. prob.}\times\text{Mispred.}\times\text{Br. Pen.})]\times[6q_i^2(1-q)+2q_i^3+9p_i^2q_i^3] \tag{14}$$

where $i =$ FP units, and

$$[1-(\text{Br. prob.}\times\text{Mispred.}\times\text{Br. Pen.})]\times 2q_i^2 \tag{15}$$

for $i =$ Integer units.

[2] This is the square of the sample correlation coefficient. The sample correlation coefficient is a number between zero and one that measures the degree to which two variables are linearly related with zero signifying "no relationship".

[3] $\overline{P}_{\text{dep}}$ encountered earlier is the *average* value of P_{dep}.

Benchmark	go	compress	m88ksim	applu	fpppp
Thr. (measured)	0.74	0.92	0.89	0.88	0.82
Thr. (model)	0.81	0.89	0.93	0.92	0.79
Thr. (rel. err.)	9.46%	-3.26%	4.49%	4.54%	-3.66%
bottleneck	Window	Decoder	Window	Window	Retire

Table 2: Measured and Analytical Throughput for an in-order issue UltraSPARC

7.2 Performance Sensitivity to Improvements in the Caches

Since the miss ratio and the miss penalty are the main metrics of any cache systems, the performance sensitivity to improvements in the data cache is expressed as derivatives of μ_{ret} with respect to $P_{\text{dat,miss}}$ and $T_{\text{dat},p}$. Let us first consider the situation when $\mu_{\text{dec}} \geq \min\{\mu_{\text{win}}, \mu_{\text{ret}}\}$. We have

$$\frac{\partial \mu_{\text{ret}}}{\partial T_{\text{dat},p}} = \frac{\partial \mu_{\text{ret}}}{\partial T_{\text{dep}}} \frac{\partial T_{\text{dep}}}{\partial T_{\text{dat},p}} \tag{16}$$

where $\frac{\partial \mu_{\text{ret}}}{\partial T_{\text{dep}}}$ is shown in Eq. 12 and Eq. 13, and

$$\frac{\partial T_{\text{dep}}}{\partial T_{\text{dat},p}} = S_i P_{\text{dat,miss}} + S_i \overline{P}_{\text{dep}} P_{\text{dat,miss}} \tag{17}$$

where i denotes the load/store unit. $\frac{\partial T_{\text{dep}}}{\partial T_{\text{dat},p}}$ is usually less than 1 in a typical processor. Therefore, $\frac{\partial \mu_{\text{ret}}}{\partial T_{\text{dat},p}} < \frac{\partial \mu_{\text{ret}}}{\partial P_{\text{dep}}}$ in most cases. In other words, performance improvement is less sensitive to data cache miss penalty reduction than dependency reduction.

Similar for $T_{\text{dat},p}$, we have

$$\frac{\partial \mu_{\text{ret}}}{\partial P_{\text{dat,miss}}} = \frac{\partial \mu_{\text{ret}}}{\partial T_{\text{dep}}} \frac{\partial T_{\text{dep}}}{\partial P_{\text{dat,miss}}} \tag{18}$$

$$\frac{\partial T_{\text{dep}}}{\partial P_{\text{dat,miss}}} = (1 + P_{\text{dep}}) S_i (T_{\text{dat},p} - 1) \tag{19}$$

When the miss penalty is high, performance improvement will be more sensitive to miss penalty reduction than dependency reduction as $\frac{\partial T_{\text{dep}}}{\partial P_{\text{dat,miss}}} > 1$ in this case.

When $\mu_{\text{dec}} \leq \min\{\mu_{\text{win}}, \mu_{\text{ret}}\}$, performance improvement is sensitive to the instruction cache miss ratio, $P_{\text{ins,miss}}$ and miss penalty, $T_{\text{ins},p}$. For different expressions of $\overline{I}_{\text{br}}$, we obtained different derivatives of $\mu_{\text{dec},n}$ with respect to $P_{\text{ins,miss}}$ from Eq. 9, Eq. 24 and Eq. 25 respectively. It should be noted that the overall performance is sensitive to $P_{\text{ins,miss}}$ in these circumstances resulting in $\mu_{\text{dec}} \leq \min\{\mu_{\text{win}}, \mu_{\text{ret}}\}$. This occurs when branch instructions account for a large proportion of the instruction mix. This is also the case in out-of-order issue with big reservation stations.

We have

$$\frac{\partial \mu_{\text{dec},n}}{\partial P_{\text{ins,miss}}} = -\frac{C_1}{C_2/C_3 + T_{\text{ins},p} P_{\text{ins,miss}}} \tag{20}$$

from Eq. 9,

$$\frac{\partial \mu_{\text{dec},n}}{\partial P_{\text{ins,miss}}} = -\frac{C_4}{C_5/C_6 + T_{\text{ins},p} P_{\text{ins,miss}}} \tag{21}$$

from Eq. 24, and

$$\frac{\partial \mu_{\text{dec},n}}{\partial P_{\text{ins,miss}}} = -\frac{C_7}{C_8/C_9 + T_{\text{ins},p} P_{\text{ins,miss}}} \tag{22}$$

from Eq. 25.

The derivatives of $\mu_{\text{dec},n}$ with respect to $T_{\text{ins},p}$ have the exactly same form as those with respect to $P_{\text{ins,miss}}$ in Eq. 20, Eq. 21 and Eq. 22. Note that in Eq. 9, 24 and 25, the numerators, C_3, C_5 and $C_7 \gg 1$ while the denominators are just around 1.

7.3 Performance Sensitivity to Improvements in the Functional Units

When $\mu_{\text{dec},n} \geq \min\{\mu_{\text{win}}, \mu_{\text{ret}}\}$, performance improvement is also sensitive to t_i, which is usually due to better implementation of the functional units. For the retirement unit, we have the following sensitivity expression.

$$\frac{\partial \mu_{\text{ret}}}{\partial t_i} = \frac{\partial \mu_{\text{ret}}}{\partial T_{\text{dep}}} \frac{\partial T_{\text{dep}}}{\partial t_i} \tag{23}$$

where $\frac{\partial T_{\text{dep}}}{\partial t_i} = S_i \times (1 + \overline{P}_{\text{dep}})$.

For instance, we showed the sensitivity to $t_{\text{integer\_unit}}$, the integer unit's latency, in the fourth row of Table 3 and 4. Since S_i is usually less than 0.5, $\frac{\partial \mu_{\text{ret}}}{\partial t_i}$ is accordingly less than $\frac{\partial \mu_{\text{ret}}}{\partial T_{\text{dep}}}$. Hence, we conclude that more emphasis should be put on reducing dependency than enhancing the functional unit.

Table 3 shows the sensitivity with respect to various changes for in-order issue and out-of-order issue policy without big reservation workstations. Tab. 4 gives the similar results for the case of out-of-order issue with reservation workstations of sufficient capacity. Insensitivity is labeled as 'NA' in Table 3 and Table 4. The insensitivity entries indicate the absence of bottlenecks.

To sum up our sensitivity analysis, we argue that the overall performance improvement is most sensitive to the data cache miss ratio except for programs where branch instruction are very frequent. Next, we see that overall performance improvement is also sensitive to dependency reduction. The overall performance improvement is relatively less sensitive to improvements to the functional units when compared with other modifications. In other words, according to our analysis, the throughput of a superscalar processor is affected by the following in *decreasing* order of importance:

Benchmark	go	compress	m88ksim	applu	fpppp	Avg.
$\partial_{\text{throughput}}/\partial P_{\text{dep}}$ or p	0.260	NA	0.320	0.177	0.471	0.307
∂throughput$/\partial T_{\text{dat},p}$	0.021	NA	0.023	0.035	0.035	0.029
∂throughput$/\partial P_{\text{dat,miss}}$	1.041	NA	1.144	1.735	1.761	1.420
∂throughput$/\partial t_{\text{integer_unit}}$	0.312	NA	0.229	0.149	0.143	0.208
∂throughput$/\partial P_{\text{ins,miss}}$	NA	4.794	NA	NA	NA	4.794
Bottleneck	window	decoder	window	window	retire	NA

Table 3: Performance sensitivities for an in-order-issue UltraSPARC-like machine

Benchmark	go	compress	m88ksim	applu	fpppp	Avg.
∂throughput$/\partial T_{\text{dep}}$	0.808	NA	0.974	0.781	0.471	0.759
∂throughput$/\partial T_{\text{dat},p}$	0.032	NA	0.032	0.045	0.035	0.036
∂throughput$/\partial P_{\text{dat,miss}}$	1.612	NA	2.264	2.258	1.761	1.973
∂throughput$/\partial t_{\text{integer_unit}}$	0.312	NA	0.229	0.149	0.143	0.241
∂throughput$/\partial P_{\text{ins,miss}}$	NA	4.794	NA	NA	NA	4.794
Bottleneck	retire	decoder	retire	retire	retire	NA

Table 4: Performance sensitivities for an out-of-order issue UltraSPARC-like machine

1. the data cache miss ratio,
2. instruction dependency,
3. the instruction cache miss ratio,
4. improvements of the functional units, and
5. the cache miss penalty.

In the light of Tables 3 and 4 where our analytical predictions were quantified, we propose that more efforts should be made to decrease data cache miss ratio and dependencies by using data prefetching or register renaming. Meanwhile, the results suggest that good branch prediction can significantly improve the overall performance of some applications in which branch instructions are very frequent. As the density of the processor chip goes up, we also propose increasing the capacities of reservation stations so that aggressive out-of-order issue policy can be implemented which in turn will increase overall performance substantially. After this is achieved , then enhancing the instruction cache is the next important issue.

8 Conclusion

In this paper, we described a realistic queuing model for superscalar processors based on a network of MCMR queues. In our model, instruction traces only needs to be analyzed once to obtain a few key parameters which characterize the traces. Using these parameters, we computed the performance of any number of superscalar processor configurations. In contrast, to obtain the results we reported via trace driven simulation would mean doing tens, if not hundreds, of simulation runs over billions of instructions.

As far as we know, sensitivity analysis has seldom been carried out. Using sensitivity analysis of the model, we explored the relative efficacy of performance improvement due to three categories of architecture modifications. Our results show that, the performance of a superscalar processor is limited by the instruction decode, issue rates and retirement rate. In terms of sensitivity, we conclude that, for in-order issue policy and out-out-order issue policy without large capacity reservation stations, the reduction of data cache miss ratio and instruction dependency is the most promising way to improve overall performance. Instruction cache improvements are helpful in some applications where branching is frequent. Speculative mechanism such as out-of-order issue gives much better overall performance only if some conditions are met, the most important of which is that reservation stations must be of sufficient capacity.

References

[Austin and Sohi, 1992] Austin, T. M., and Sohi, G. S. (1992). 'Dynamic Dependency Analysis of Ordinary Programs', *Proc. 19th Int'l Symp. on Comp. Arch.*, pp. 342-351.

[Dubey et al,. 1994] Dubey, P. K., Adams, G. B., and Flynn, M. J. (1994). 'Instruction Window Size Trade-Offs and Characterization of Program Parallelism', *IEEE Trans. on Comp.*, 43(4), pp. 431-442.

[Heinrich, 1996] Heinrich, J. (1996). *MIPS R10000 User's Manual. Version 1.1.* MIPS Technologies, Inc. CA, USA.

[Jain, 1991] Jain, R. (1991). *The Art of Computer Systems Performance Analysis*, pp. 535-536, John Wiley & Sons, Inc. New York USA.

[Kleinrock, 1975] Kleinrock, L. (1975-1976). *Queueing Systems* (vols. 1 & 2), John Wiley & Sons, Inc. New York USA.

[Lam and Wilson, 1992] Lam, M. S., and Wilson, R. P. (1992). 'Limits of Control Flow on Parallelism', *Proc. of 19th Int'l Symp. on Comp. Arch*, pp. 46-57.

[Lazowska et al., 1984] Lazowska, E. D., Zahorjan, J., Graham, G. S., and Sevolk, K. C. (1984). *Quantitative System Performance*, pp. 135-136, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, USA.

[Loh, 1997] Loh, K. S. (1997). 'Superscaler Processor Simulator', *Honour Year Project Report*, Dept. of Information Systems and Computer Science, National University of Singapore.

[MacDougall, 1987] MacDougall, M. H. (1987). *Simulating Computer Systems Techniques and Tools*, MIT Press.

[Matta and Shankar, 1995] Matta, I., and Shankar, A. U. (1995). 'Z-Iteration: A Simple Method for Throughput Estimation in Time-Dependent Multi-Class Systems', *Proc. of SIGMETRICS '95*, pp. 126-135. Ottowa, Canada.

[Noonberg and Shen, 1994] Noonburg, D. B., and Shen, J. P. (1994). 'Theoretical Modeling of Superscalar Processor Performance', *Proc. of MICRO 27*, pp. 53-62, San Jose, USA.

[Pyun et al., 1998] Pyun, Y.H., Park, C.S., and Choi, S.B. (1998). 'The effect of instruction window on the performance of superscalar processors', *IEICE Transactions on Fundamentals of Electronics Communications and Computer Sciences*, E81A: (6) 1036-1044, Jun 1998.

[Schlansker and Rau, 1999] Schlansker, M., and Rau, B.R. (1999). 'EPIC: An Architecture for Instruction-Level Parallel Processors', *HP Labs Technical Report HPL-1999-111.* 1999.

[Zhu and Wong, 1997] Zhu, Y., and Wong, W. F. (1997). 'Performance Analysis of Superscalar Processor Using a Queueing Model', *Proc. of Computer Architecture '97: The Second Australasian Conference*, pp. 147-158, Sydney, Australia, Springer-Verlag.

[Zhu and Wong, 1998] Zhu, Y., and Wong, W. F. (1998). 'The Effect of Instruction Dependency on Superscalar Processor Performance', *Australian Computer Science Communications*, pp. 215-226, Volume 20, Number 4, Springer-Verlag.

[Zhu and Wong, 2000] Zhu, Y., and Wong, W. F. (2000). 'Modeling Archietctural Improvements in Superscalar Processors', Proc. of HPC-Asia 2000, pp. 28-30. Beijing, P.R.C.

Appendix

Let $|C|$ be the cardinality of the set, C. Each class, $c \in C$, is an open class with arrival rate, λ_c. We denote the vector of arrival rates by $\vec{\lambda} \equiv (\lambda_1, \lambda_2, ..., \lambda_{|C|})$.

1. *Processing Capacity* A system is said to have sufficient *capacity* to process a given offered load $\vec{\lambda}$ if it is capable of doing so when subjected to the workload over a long period of time. For multiple class models, sufficient capacity exists if the following inequality is satisfied: $\max_r\{\sum_{c\in C^r} \lambda_c^r d_c^r\} < 1$ where d_c^r is the service time demanded by a class c customer on resource r.

2. *Utilization* According to the Utilization Law [Kleinrock, 1975], *utilization* is expressed in terms of arriving rate vector, λ_c^r, $c \in C^r$ and the service demand of the customer, d_c^r: $U_c^r(\vec{\lambda}) = x_c^r(\vec{\lambda})\mu_c^r = \lambda_c^r d_c^r$

3. *Residence Time* In Lazowska [Lazowska et al., 1984], the *residence time* for queuing servers under the FCFS is given by $Res_c^r(\vec{\lambda}) = \frac{d_c^r}{1-\sum_{j=1}^{|C|} U_j^r(\vec{\lambda})}$. In relation to our model, the residence time is the time which class c instructions have to spend before they can be served by a functional component, r, of the processor.

4. *Queue Length* Assuming a FCFS discipline and applying Little's Law to the residence time equation above, the queue length of class c at server k, $q_c^r(\vec{\lambda})$, is given by: $q_c^r(\vec{\lambda}) = \lambda_c Res_c^r(\vec{\lambda}) = (U_c^r(\vec{\lambda}))/(1 - \sum_{j=1}^{|C|} U_j^r(\vec{\lambda}))$.

Other measures like throughput, system response time, average number of instructions in the system and blocking probability can also be calculated [Zhu and Wong, 1997].

We shall now analyze the decoder. If $\overline{I}_{\rm br} \geq W_{\rm dec}$ and $\overline{I}_{\rm br}/W_{\rm dec} < T_{\rm br}$, then $u_{\overline{I}_{\rm br}+1}$ is predicted before u_1 is resolved. If u_1 is predicted wrongly, $u_{\overline{I}_{\rm br}+1}$ and the instructions fetched due to its prediction will have to be discarded. Therefore, both the branch penalty arising from u_1 and the possible penalty caused by $u_{\overline{I}_{\rm br}+1}$ will affect the decoding rate. Otherwise, only $u_{\overline{I}_{\rm br}+1}$ needs to be considered in the computation of the decoding rate. Consequently, to decode $\overline{I}_{\rm br}+2$ instructions without overflow at the central window, the average time needed is $T_{{\rm dec},n} = C_5 + (C_6) \times T_{{\rm ins},p} \times (P_{\rm ins,miss})$, where C_5 stands for $[(\overline{I}_{\rm br}+2)/W_{\rm dec} + T_{\rm br} \times (1 - P_{\rm prtd})] \times P_{\rm prtd} + [(\overline{I}_{\rm br}+2)/W_{\rm dec} + T_{\rm br} + T_{\rm br} \times (1 - P_{\rm prtd})] \times (1 - P_{\rm prtd})$, C_6 stands for $(\overline{I}_{\rm br}+2)$. The average decoding rate without overflow at the central window and reorder buffer is given by

$$\mu_{{\rm dec},n} = (C_4)/(T_{{\rm dec},n}) \tag{24}$$

where C_4 denotes $(\overline{I}_{\rm br}+2)$.

If $\overline{I}_{\rm br} > W_{\rm dec}$ and $\overline{I}_{\rm br}/W_{\rm dec} \geq T_{\rm br}$, then $u_{\overline{I}_{\rm br}+1}$ is predicted only after u_1 is resolved. Therefore, we only need to consider $u_{\overline{I}_{\rm br}+1}$ in calculating the decoding rate. This is shown in the following equation

$$\mu_{{\rm dec},n} = C_7/[C_8 + C_9 \times T_{{\rm ins},p} \times (P_{\rm ins,miss})] \tag{25}$$

where C_7 stands for $(\overline{I}_{\rm br}+2)$, C_8 stands for $\overline{I}_{\rm br}/W_{\rm dec}+T_{\rm br}\times(1-P_{\rm prtd})$ and C_9 stands for $(\overline{I}_{\rm br}+2)$.

When $\mu_{{\rm dec},n}$ exceeds $\mu_{\rm win}$ or $\mu_{\rm ret}$, the decode unit will not be working at a stable speed $\mu_{{\rm dec},n}$ since the unit stalls when the central window or reorder buffer overflows, and continues to work only when enough room ($\mu_{{\rm dec},n}$ in our model) is available in the central window or the reorder buffer. The decode unit, accordingly, works in alternating stalling and decoding phases. Having considered this, the average decoding rate with overflow in the central window is as follows

$$\frac{\mu_{{\rm dec},f}=(\mu_{{\rm dec},n}^2/(\mu_{{\rm dec},n}-\mu_{\rm win}))}{(\frac{\mu_{{\rm dec},n}}{\mu_{{\rm dec},n}-\mu_{\rm win}}+\mu_{{\rm dec},n}/\mu_{\rm win})} \tag{26}$$

However, using the fact that central window becomes a bottleneck when $\mu_{\rm win}$ is low, we can simplify the equation to

$$\mu_{{\rm dec},f}=\mu_{{\rm dec},n} \tag{27}$$

The Eq. 27 shows that the effective service rate of the decode unit is determined by $\mu_{{\rm dec},n}$ when the central window is filled up from time to time. In other words, the decoder, which decodes faster than the central window issues instructions, does not make effective contributions to the overall throughput. The result is also true with $\mu_{\rm win}$ in Eq. 26 changed into $\mu_{\rm ret}$ when the reorder buffer overflow is considered. In the case where $\mu_{{\rm dec},n}$ is greater than $\mu_{\rm win}$, the average queue length at the central window is given by

$$\frac{(\frac{\mu_{{\rm dec},n}}{\mu_{\rm win}}Z_w+\frac{\mu_{{\rm dec},n}}{\mu_{{\rm dec},n}-\mu_{\rm win}}(Z_w-\mu_{{\rm dec},n}))}{(\frac{\mu_{{\rm dec},n}}{\mu_{\rm win}}+\frac{\mu_{{\rm dec},n}}{\mu_{{\rm dec},n}-\mu_{\rm win}})} \tag{28}$$

where Z_w is the size of the central window. When $\mu_{{\rm dec},n}$ is greater than $\mu_{\rm ret}$, $\mu_{\rm ret}$ should be substituted for $\mu_{\rm win}$ in Eq. 28.

For certain very small applications, we have observed values of $\mu_{\rm win}$ exceeding $\mu_{{\rm dec},n}$. From Section 4, the average queue at the central window is given by

$$(\mu_{{\rm dec},n}/\mu_{\rm win})/(1-\mu_{{\rm dec},n}/\mu_{\rm win}) \tag{29}$$

Glossary of Terms

1. MC, MR: the multiple instruction classes, and processor components
2. $\mathcal{P}(\mathcal{I})$: the execution trace obtained by executing $\mathcal{P}$ with input $\mathcal{I}$
3. $\overline{D}$: average dependency interval (inclusive of one of the instruction in the dependency)
4. $\overline{I}_{\rm br}$: the average number of (non-branch) instructions between two branch instructions (inclusive of one of them).
5. $\overline{P}_{\rm dep}$: designates the possibility that $\exists w \in \mathcal{P}(\mathcal{I}), w \xrightarrow{D} u$, where $D \leq \overline{D}$. item $T_{\rm dep}$: average time for an antecedent instruction to pass through the functional units
6. Z_w: the size of the central window
7. $\mu_{{\rm dec},n}, \mu_{{\rm dec},f}$: the average decode rate without or with overflow at the central window
8. $\mu_{\rm win}$: the average issue rate of the central window
9. $W_{\rm dec}, W_{\rm iss}$: the decode and issue width
10. $T_{{\rm ins},p}, T_{{\rm dat},p}$: the instruction, data cache miss penalty time
11. $P_{\rm prtd}$: the probability of a correct branch prediction
12. $u \xrightarrow{D}_o v$, $u \xrightarrow{f}_f v$, $u \xrightarrow{D}_a v$: instruction u is output, flow or anti -dependent on v with the dependency interval D