

Instruction-level Microprocessor Modeling of Scientific Applications

Kirk W. Cameron¹, Yong Luo², and James Scharzmeier³

¹Louisiana State University Department of Computer Science,
298 Coates Hall, Baton Rouge, LA 70803-4020
cameron@bit.csc.lsu.edu

^{1,2}Los Alamos National Laboratory Scientific Computing Group,
Mail Stop B256 CIC-19, Los Alamos, NM 87545
{kirk, yongl}@lanl.gov

³Silicon Graphics,
1168 Industrial Blvd, Chippewa Falls, WI 54729
jads@sgi.com

Abstract. Superscalar microprocessor efficiency is generally not as high as anticipated. In fact, sustained utilization below thirty percent of peak is not uncommon, even for fully optimized, cache-friendly codes. Where cycles are lost is the topic of much research. In this paper we attempt to model architectural effect on processor utilization with and without memory influence. By presenting analytical formulas that use measurements from "on-chip" performance counters, we provide a novel technique for modeling state-of-the-art microprocessors over ASCI representative scientific applications. ASCI is the Accelerated Strategic Computing Initiative sponsored by the US Department of Energy. We derive formulas for calculating a lower bound for CPI_0 , CPI without memory effect, and we quantify utilization of architectural parameters. These equations are architecturally diagnostic and qualitatively predictive in nature. Results provide promise in code characterization, and empirical/analytical modeling.

1 Introduction

Instruction-level modeling is not new. Work found in [1,2,3,4,5] offers a variety of analytical attempts at modeling as early as 1984. This paper seeks to contribute to the body of scientific knowledge by offering a novel approach at microprocessor modeling using hardware performance counter measurements to provide data for derived models instead of using simulators as in [2,4,5]. Traditional techniques such as measuring CPI, FLOPS rate, and cache miss ratios are insufficient in quantifying the attributes and influence current architectural advances have on processor performance. Our technique has the advantage of collecting instruction-level characteristics in a few runs virtually without overhead or slowdown. In this paper, a formula of CPI_0 estimation is presented and validated (within 5% error) through some synthetic codes on a real machine.

This paper describes the application of this technique on two SGI R10000-based systems: Origin2000 and PowerChallenge, using the SGI performance counter tool

perfex and its associated libraries. Some results are directly validated by the empirical memory model [6] and the statistic model [7]. While targeting a specific architecture for example and analysis, the applied technique is general in nature. Current research includes applications on other processors.

We begin the presentation in this paper with a description of the underlying code characterization method used to derive our equations. The parameters and motivation behind this approach are discussed followed by a series of assumptions to facilitate modeling of the architecture-code relationship. We provide a two-tiered approach to analytically modeling the inner workings of a superscalar microprocessor. First we discuss the effects on CPI due to architectural limitations within the chip itself. We present equations and their derivations based on previous assumptions. Discussions of our validation methods on the MIPS R10000 are also provided. After substantial emphasis on the underlying diagnostic and predictive equations, we provide example analysis on the MIPS R10000 under two different memory hierarchical implementations for several key ASCI scientific codes. Herein we discuss analytically drawn conclusions and interesting observations. We conclude with a discussion of overall observations and directions for future work.

2 General Microprocessor Model

Today's superscalar processors are very complex incorporating architectural improvements to increase the amount of work performed while waiting on memory. These enhancements such as out-of-order execution, speculative execution, and outstanding misses contribute to the inherent difficulty in modeling processors of this type. We introduce a general microprocessor model that is applicable to most modern superscalar architectures. In particular, our model focuses on the queue lengths and dispatching capabilities of the processor under analysis. It incorporates the enhancements mentioned and is flexible enough to model future architectural changes. Before describing the model, it is necessary to discuss the parameters that will be used to characterize codes and architecture.

2.1 Application Dependent Parameters

We use a set of instruction-level parameters as described in [8] to characterize particular workloads. In order to analyze the behavior of those queues mentioned earlier, we need to measure the average inter-arrival distance in number of instructions, not cycles which are dependent on both architecture and application. We focus on the importance of using instruction-level parameters to characterize a workload so as to associate the workload performance behavior with the microprocessor architecture. When we characterize an application, one of the keys is to separate the architectural factors so that a true workload characterization can be presented. The "number of instructions between two consecutive operations" idea is borrowed from the concept of run-length defined in [9]. We calculate these values uniformly by dividing the number of x-type instructions by the total number of instructions. This λ value is a factor without a unit such that $1/\lambda_x$ is the probability of occurrence of instruction x over the incoming instruction stream. λ_{L_1} and λ_{L_2} refer to

the occurrence of L1 and L2 misses. These are inclusive and a subset of overall memory instructions.

2.2 Hardware Dependent Parameters

There are certain architectural parameters that, generally speaking, apply to all current superscalar microprocessors. We include two particular parameters as a first step in developing equations that are code dependent (relying on the aforementioned application parameters) as well as architecturally dependent. Superscalar processors generally have the ability to decode multiple instructions per clock period. This affects the rate at which instructions collect within the queues of a microprocessor and thus we deem it necessary to include in our modeling equations. We define β as the ideal instruction dispatch rate for a given microprocessor. In a similar fashion, superscalar processors often include multiple execution units to service pending requests. Differing combinations of these types of units within the given architecture influence a term we define as Δ_x , or the preset hardware execution rate of the x-queue. x is the current instruction type of interest, namely m, i, or f for memory, integer, or floating point instructions.

2.3 CPU model without L₁ misses

In Figure 1 we portray a simplified version of an arbitrary superscalar microprocessor. In this first tier of our modeling technique, we wish to concentrate only on the architecture within the chip itself ignoring all "off-chip" activity, namely memory accesses. Common architectural features of many modern superscalar microprocessors can be generalized as separated pipelines for functional units: one or more integer pipeline(s) for ALU(s), one or more floating-point pipeline(s) for FPU(s), and one or more memory operation pipeline(s) for load/store unit(s). As is the case in both tiers of our model, we need to simplify things to allow for easier characterization. Detailed explanations and justifications of our assumptions and some equations can be found in [10]. We will minimize these assumptions in future work. In the present case, for many scientific applications, this modeling technique is useful as will be shown when describing the analysis of the MIPS R10000.

Assumption 1 Uniform distribution of instructions

Assumption 2 λ values individually converge

Assumption 3 Branch influence is negligible

Assumption 4 Icache effect is negligible

Assumption 5 No data dependence

Assumption 6 Ideal (infinite) L1 cache (or no L1 misses)

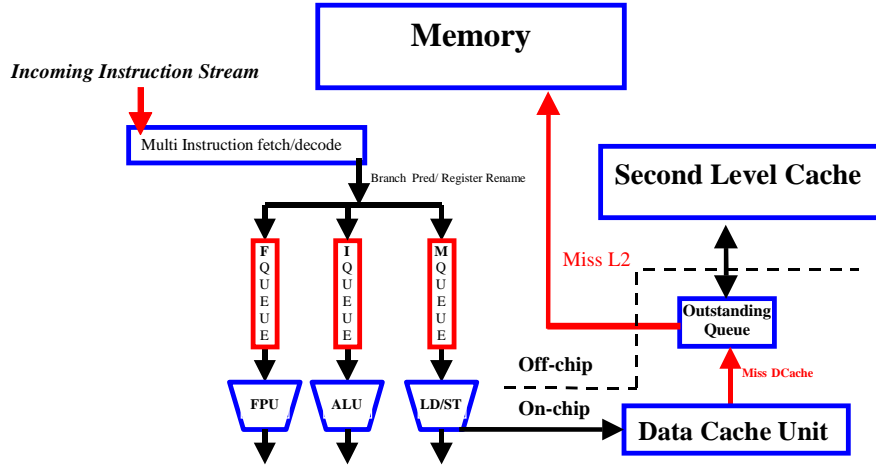


Fig. 1 General pipeline model for CPU and memory

λ values will converge as steady state is reached, and branch and icache influence are negligible for many scientific applications. Assuming ideal L1 cache for the sake of estimating CPI_0 follows CPI separating concepts discussed in [11]. Therein, Hennessy refers to CPI_0 as Pipeline CPI; the two are interchangeable. The assumptions of most concern are uniform distribution and no data dependence. Uniform distribution by its nature guarantees a lower bound on CPI_0 when compared to a normal instruction mix with the same λ values. No data dependence also guarantees a lower bound on CPI_0 , but more importantly, allows us to separate contributions to stall and model dependencies incrementally as research continues.

Conservation Equation

$$\frac{1}{\lambda_m} + \frac{1}{\lambda_i} + \frac{1}{\lambda_e} = 1 \quad (1)$$

This equation describes the property of any instruction mix dictated by previous assumptions.

Growth Equation for Queue x

Using our original assumptions and some basic algebra, we define a growth equation to describe the state of a particular queue within the microprocessor. Let us define G_x as the growth rate of queued instructions of type x within the microprocessor. We must take into account the rate at which instructions execute as well as the rate at which they are decoded giving:

$$G_x = \beta / \lambda_x - \Delta_x \quad (2)$$

where G_x is the growth rate for the x-queue of interest, β is the ideal instruction dispatching rate for the given microprocessor, $1/\lambda_x$ is the probability of encountering an instruction of type x for a given code, Δ_x is the preset hardware execution rate of the x-queue, and x is the current instruction type of interest, namely m, i, or f for memory, integer, or floating point instructions.

As a steady state is reached, positive growth rates will contribute to cpu stalls as any queue within the microprocessor reaches its capacity. A limiting factor is the key contributor to stalls within the microprocessor (excluding dependencies and memory latency as we assume infinite, 1 cycle access to L1 cache). In particular, we use our growth formula to diagnose the limiting factor for a particular code-architecture combination. This limiting factor will be the key contributor to resource stalls within the microprocessor for the code measured. It will also indicate the type of instruction that will have the greatest influence on CPI_0 . Single positive growth rates simplify determination of the limiting factor, but multiple positive growth rates lead to contemplation of K, a threshold of maximum instructions in flight; in other words in some cases we must consider queue interaction as well as individual contributions to stalling. There are some limitations of growth rates for typical superscalar processors. We discuss this in relation to another intuitive assumption.

Assumption 7 $\sum_x \Delta_x \geq \beta$ where x is i, f, or m.

Consider multiplying the conservation equation by β

$$\beta/\lambda_m + \beta/\lambda_i + \beta/\lambda_f = \beta \quad (3)$$

Let

$$\sum_x \Delta_x = \Delta_m + \Delta_i + \Delta_f \quad (4)$$

Then overall growth rate in the processor, G, is defined as,

$$G = \beta - \sum_x \Delta_x \quad (5)$$

By assumption 7 and above, we show in [10] that this implies $G_x \leq 0$ for at least one of x=m,i,f. This allows us to assume that we will never have more than two positive growth rates in a three-queue situation. So when we do analysis to determine the limiting factor for a code-architecture combination, for multiple positive growth rates we just need to figure out whether a single queue fills first or the threshold, K, is reached. This simplifies our analysis significantly. This concept is extendible to other queue architectures as well. This paper focuses on the case of single positive growth rate. In the scenario of more than one positive growth rate, the derivation process of CPI_0 and other analysis are essentially the same and vary just slightly in detail.

Lower bound for CPI_0

Since we assume an infinite L1 cache, no significant branching effect, and no data dependency, calculations of CPI_0 based on λ values must give a lower bound to CPI_0 . It is necessary to determine the limiting factor using the growth formula prior to using the following equation. CPI_0 is the cycles per instruction for an application-architecture combination that assumes no influence from memory accesses. Following our previous assumptions, we give a formula to calculate this CPI_0 based on characteristics of the application and architecture under analysis. Here, we briefly discuss its derivation.

Let C be the total number of cycles necessary to complete a problem. Let N be the total number of instructions for the same problem. We can calculate CPI_0 as the cycles, C , divided by the instructions, N , when we have an ideal L1 cache assumption as stated earlier.

$$CPI_0 = \frac{C}{N} = \frac{\text{total \# cycles}}{\text{total \# instr}} \quad (6)$$

With our other assumptions, if we can calculate C and N , we will be calculating a lower bound for CPI_0 . Such a bound will provide a base for expanding analytical formulas to estimate actual CPI_0 along with a starting point for quantifying dependencies and branch effects. When we encounter a limiting factor as described earlier, with the aforementioned assumptions, the total number of cycles for the entire problem is controlled by the dispatch rate for the limiting factor instruction type. Thus, we can calculate the number of cycles necessary to complete the entire problem.

$$C = \frac{\text{total \# of } x \text{ instr}}{\Delta_x} \quad (7)$$

$$C = \frac{\text{total \# instr} * 1/\lambda_x}{\Delta_x} \quad (8)$$

And finally using equations 6-8, we can simplify the relationship between code and hardware by quantifying our lower bound for CPI_0 as

$$CPI_0 = \frac{1}{\lambda_x \Delta_x} \quad (9)$$

This simplification is actually quite interesting as it shows CPI_0 is dependent upon the product of probability of a limiting factor instruction and the associated execution rate (in CPI) of the associated limiting factor queue. At this point, we stipulate that this is the formula for CPI_0 when a limiting factor is present. If all growth rates are negative, there is no single limiting factor. CPI_0 can then be calculated as $1/\beta$, the ideal CPI_0 . In the scenario of more than one positive growth rate, the derivation process of CPI_0 is essentially the same and varies just slightly in details.

2.4 CPU model with L₁ misses

We incorporate off-chip memory hierarchy influence on stalls within the microprocessor in the general diagram provided in Figure 1. In this second tier of our modeling technique, we focus on the architectural limitations of the microprocessor when faced with the additional complication of L1 cache misses. Here we will discuss assumptions for this level of modeling along with equations that analytically qualify other aspects of the code-architecture relationship. For this CPU memory modeling, our general assumptions 1-5 apply as well, and for the same reasons mentioned earlier. We no longer assume an infinite L1 cache as in assumption 6, but in the interest of simplicity, we follow assumption 7 indicating that we must have a particular limiting factor.

Outstanding Miss Utilization

Most of today's superscalar microprocessors allow overlap of computation through support for outstanding cache misses. Through comparison of λ values when misses to L1 cache occur, we can qualitatively infer the advantages of lengthening the number of outstanding supported on chip. This analysis is extendible to multi-layered caches and is not limited to this simple example. We define a term Q'_o as the maximum number of outstanding cache misses utilized by a code on a particular architecture.

$$Q'_o = \frac{Q_m * \lambda_m}{\lambda_{L1}} \quad (10)$$

This parameter gives us insight to the exploitation of outstanding misses for a particular code on a given architecture. Here again, we use the percentage of L1 cache misses multiplied by (in this case) the particular queue length for the memory queue. We should mention that this formula is also most useful when memory instructions are the limiting factor.

3 Model Validation on MIPS R10000

Validation of analytical methods is inherently difficult and many promising techniques go unused because of the limited ability to validate. Others turn out to be useless except in the overly safe environment provided by simulators. To validate our model, we chose to use synthetic codes on real processors using hardware performance counters to provide necessary counts as inputs. In this way, we hope to underscore the practicality of our modeling technique and the time saved using our characterization method. The modeling technique discussed so far is general in nature and easily modified for different architectures.

Both the Origin 2000 (O2K) and PowerChallenge (PC) use the MIPS R10000 RISC based microprocessor. The R10000 processor is a 4-way superscalar CPU with an integer, floating point, and memory queue each containing 16 entries. Ignoring branch and icache effects, stalls during execution are typically attributed to: 1 of 3 queues full, outstanding misses full (4 for L1 on R10K), maximum 32 outstanding

instructions reached, renaming registers consumed, and back-to-back write-backs from L1. Architectural characteristics stipulate consumption of all renaming registers and back-to-back write-backs are very rare, so we focus on the other constraints. As a good first-order approximation [12], at each cycle, the load/store unit can execute up to one memory instruction, and the two integer and two floating point units can each execute two instructions.

We have created code that we can modify to ensure certain instruction streams are fed to the microprocessor in the interest of validation. We use direct hardware counter measurements to ensure synthetically created code meets all assumptions. In Table 1, we present a series of uniformly distributed instruction mixes and measured results to show we satisfy our assumptions. The pattern descriptions consist of one or two parts. The first part describes the repeated sequence of instructions. For example, miii refers to a memory instruction followed by three integer instructions. This series constitutes a synthetic stream repeated to the point of stability (in the millions of instructions). If a stream contains more than two f's (i.e. floating point operations), we specify the types of operations after the "underscore". For example, fff_*** refers to a repeated sequence of floating point instructions of the type "multiply", "add", "multiply". We specify these in order to account for the fact that while claiming two floating point units for the MIPS R10000, in reality there is one floating point servicing only additions and another servicing only multiplication operations. The mix of addition's and multiplication's thus affects CPI via a change in Δ_x as apparent

Table 1. Results for synthetic instruction streams on MIPS R10000

Pattern	Growth Rates			Limiting Factor	λ_x	Δ_x	Meas CPI	Calc CPI	Rel Error
	G _r	G _m	G _i						
fff_***	1.9578	-0.9945	-1.9819	f	1.0107	1.5000	0.6622	0.6596	-0.40%
iff_***	0.9761	-0.9959	-0.9942	f	1.3440	1.5000	0.5192	0.4960	-4.47%
iii	-2.0000	-0.9918	1.9640	i	1.0091	2.0000	0.5057	0.4955	-2.01%
iiif	-1.0079	-0.9959	0.9898	i	1.3379	2.0000	0.3962	0.3737	-5.67%
mfff_***	0.9762	-0.0038	-1.9863	f	1.3440	1.5000	0.4989	0.4960	-0.57%
miii	-2.0000	-0.0038	0.9898	i	1.3379	2.0000	0.3960	0.3737	-5.63%
mm	-2.0000	2.9450	-1.9728	m	1.0139	1.0000	1.0010	0.9863	-1.47%
mmff_***	-0.0159	0.9882	-1.9863	m	2.0118	1.0000	0.5044	0.4971	-1.45%
mmif	-1.0079	0.9882	-0.9943	m	2.0118	1.0000	0.5072	0.4971	-2.01%
mmii	-2.0000	0.9882	-0.0022	m	2.0119	1.0000	0.5070	0.4970	-1.97%
mmmf	-1.0079	1.9803	-1.9864	m	1.3422	1.0000	0.7553	0.7451	-1.35%
mmmi	-2.0000	1.9803	-0.9943	m	1.3422	1.0000	0.7526	0.7451	-1.01%

in Table 1. We chose a mix of instructions to cover most possible permutations for a four-instruction mix without providing every single permutation. This provides us with a concise list of instances with excellent coverage.

There are several interesting observations to be made in Table 1. When a certain instruction is not present, its associated growth rate is equal to $-\Delta_x$. This should be interpreted to mean there are no instructions influencing its associated queue. In each of these examples, a single positive growth rate is found indicating a single limiting factor. We use this factor's associated queue to calculate CPI_0 in this chart. In Table 1, all of these instruction streams contribute directly to CPI_0 while our assumptions are met and there are no other contributors to CPI; thus $CPI_0=CPI$ in this context.

Table 2. Results for ideal synthetic instruction streams on MIPS R10000

Pattern	Growth Rates			β	Meas CPI	Calc CPI	Rel Error
	G_r	G_m	G_i				
iiif_+*	-0.0159	-0.9959	-0.0022	4.0000	0.2576	0.2500	-2.94%
miff_+*	-0.0159	-0.0038	-0.9942	4.0000	0.2580	0.2500	-3.11%
miif	-1.0079	-0.0038	-0.0022	4.0000	0.2577	0.2500	-3.00%

Table 1 shows our calculated and measured CPI_0 are within the tolerance of the counters themselves, implying they are quite accurate. Table 2 shows the results of perfect instruction mix giving the ideal CPI of the MIPS R10000. These results directly validate our model on the MIPS R10000. Thus, with our assumptions, we are able to model CPI_0 with a great deal of accuracy. At this time we are unable to directly validate the outstanding miss utilization formula, results in [6,7] confirm but do not validate our conclusions. At this time we use this formula qualitatively; we see it as a corollary to our main cpi_0 formula. We are thus able to conclude that the presented tier-1 formulas are valid for the given assumptions on the MIPS R10000. Since our theory is general in nature, we believe validation on other processors will support these findings.

4 Bottleneck Analysis of SGI MIPS R10000

Three applications (5 codes), which form the building blocks for many nuclear physics simulations in Los Alamos National Laboratory, were used in this study. Detailed descriptions of these codes can be found in [10]. SWEEP is a 3-d neutral particle transport code and DSWEEP is a vectorized version of the same. HYDRO is a 2-d Lagrangian hydrodynamics code and HYDRO-t is a unit stride version of the same. HEAT is an implicit diffusion PDE solver.

For the MIPS R10000 and our associated codes, we must show assumptions 1-7 are met. There are two assumptions that need some explanation. Uniform distribution is obviously not going to be found in our codes. In our technique, we extract the λ values from the measured codes. As described earlier, these values are used to create (theoretically) a synthetic, uniformly distributed, instruction stream. Our actual codes also contain dependencies. As mentioned earlier, we do not model dependencies in our equations. Instruction streams created with λ values are (again theoretically) independent of time. We can also intuitively infer that dependencies will not influence the instruction sequence committed to machine-state. Dependencies will affect the overall number of cycles for an application, but not the order in which instructions graduate from the processor. In other words, the CPI_0 calculated will be a lower bound for CPI_0 that does incorporate the effect of dependencies and instruction clustering. The argument holds for the infinite cache assumption as well. In this case, we will again be modeling a best-case scenario. For tier-2 modeling, the infinite L1 cache assumption is no longer of consequence.

To discount the effect of branch misprediction and the overhead impact of branch instructions, we also need to obtain the ratios of branch instructions and branch mispredictions to ensure the applications can be simplified as three major instruction flows (FP, Int, and Memory). On the other hand, the instruction cache miss ratio is also considered to see if the instruction fetch effect can be significant. The key of this

methodology is to estimate which of the λ values can cause stall of the microprocessor due to the limitation of architectural constraints.

Table 3 exhibits branch ratios, branch misprediction ratios, and the instruction cache miss ratios for all these codes. It is clear from Table 3 data that both branch

Table 3. Branch and icache characteristics for measured codes

	Branch Ratio (branch per instruction)	Miss Prediction Ratio (miss_pred per branch)	Branch Miss Ratio (miss_pred per instruction)	Icache Miss Ratio (icache_miss per instruction)
SWEEP	0.0653	0.1365	0.0089	0.0002
DSWEEP	0.0570	0.0340	0.0017	0.0001
HEAT	0.0554	0.0393	0.0022	0.0017
HYDRO	0.1052	0.0988	0.0104	0.0088
HYDROT	0.1057	0.1126	0.0103	0.0087

and instruction cache effect can be negligible. Under this condition, the performance study of these codes can focus on the impact of the three major instruction flows (FP, Int, & Memory).

[10] shows the variations of the λ s for all 5 benchmark codes in this study. These figures demonstrate that the λ s converge to constant values with increasing problem sizes. This is understood as the instruction flow pattern of a problem that reaches its steady state. This phenomenon proves that λ s can be used in characterizing these codes once they reach the steady state. As mentioned in the architectural description of the MIPS R10000 earlier, assumption 7 holds as well. $\beta=4$ due to the chips 4-way superscalar capability. $\Delta_m=1$, $\Delta_f=2$, $\Delta_i=2$ summing to a total of 5 functional units. Thus $G<0$ and assumption 7 follows. We have now shown that all assumptions for both modeling levels have been considered. We can now apply the models to the measurements obtained on the Origin 2000 and PowerChallenge.

Utilizing these instruction-level characteristics, we calculate the growth rates for each code over both machines in Table 4. Due to their architectural similarity, the growth rates are identical across PowerChallenge and Origin 2000. For Sweep, Dsweep, and Heat the only positive growth rate is given in G_m . This leads us to declare the memory instruction growth rate as our limiting factor for these codes on these machines. A limiting factor is the key contributor to stalls within the microprocessor (excluding dependencies and memory latency as we assume infinite L1 cache). For these codes, it is very likely the memory queue will fill, leading to stalls in decoding as entries graduate slower than they arrive. For Hydro and Hydro-t, we have positive growth rates for the memory and integer queues. This leaves us two possibilities for the limiting factor. The queue associated with the maximum of the two growth rates in the ideal case would fill first, namely the integer queue. This can

Table 4. Measured growth rates for ASCII codes on MIPS R10000

	Sweep			DswEEP			Heat			Hydro			Hydro-t		
	G_m	G_i	G_r	G_m	G_i	G_r	G_m	G_i	G_r	G_m	G_i	G_r	G_m	G_i	G_r
PowerChallenge	0.41	-0.73	-0.68	0.84	-0.65	-1.19	0.43	-0.32	-1.11	0.08	0.11	-1.19	0.08	0.12	-1.20
Origin 2000	0.42	-0.76	-0.66	0.89	-0.70	-1.19	0.42	-0.32	-1.11	0.09	0.10	-1.19	0.08	0.12	-1.20

only happen however, if the maximum instruction threshold K is not reached. As

mentioned above, $K=32$ for the MIPS R10000. Since the memory and integer queue lengths are both 16, we cannot reach the maximum number of instructions prior to stalling on a single queue. Thus, the limiting factor for both of these codes will be the integer instruction growth rate.

In Table 5, the values for λ_{L1} over the codes and machines are given. The results when calculating Q'_o for the Origin 2000 are given in Table 6; PowerChallenge results are identical. If the number of maximum outstandings utilized by a code is less than 4, the outstanding misses are not fully utilized. From Table 6, we can see the maximum number of misses that could be utilized for a given code instruction stream, assuming that it is uniformly distributed. These results and the outstanding miss utilization formula generally serve to complement the “on-chip” cpi formula providing code-machine characteristics for bottleneck analysis.

5 Conclusions

This multi-level procedure based on real-time code measurements can provide both analysis of current performance and evaluation of possible gains/losses of simple architectural changes such as increasing queue length, increasing number of outstandings, or increasing cache size. Our outstanding miss utilization equation showed the bounds for architecture code combinations allowing us to determine whether an increase in outstanding misses would serve a useful purpose for our codes on future generations of processors of this type. Further research will include incorporating latency, branching, and dependencies in the current method.

Table 5. Cache miss distances for ASCII codes on MIPS R10000

	Sweep		Dsweep		Heat		Hydro		Hydro-t	
	λ_{L1}	λ_{L2}	λ_{L1}	λ_{L2}	λ_{L1}	λ_{L2}	λ_{L1}	λ_{L2}	λ_{L1}	λ_{L2}
PowerChallenge	26.6	112.8	12.5	34.6	15.5	62.2	13.5	78.8	30.3	274.2
Origin 2000	24.9	122.9	12.7	38.0	15.5	62.6	13.4	219.4	30.3	290.1

Table 6. Outstanding miss utilization, Q'_o .

	Sweep	Dsweep	Heat	Hydro	Hydro-t
Origin 2000	1.8	2.7	2.9	4.4	2.0

References:

1. Albonese, D.H., and Koren, I., A Mean Value Analysis Multiprocessor Model Incorporating Superscalar Processors and Latency Tolerating Techniques, *International Journal of Parallel Programming*, Vol. 24, No. 3, 1996.
2. Emma, P.G., and Davidson, E.S., Characterization of Branch and Data Dependencies in Programs for Evaluating Pipeline Performance, *IEEE Transactions on Computers*, Vol. C-36, No. 7, July 1987.
3. De Gloria, A., Ancarani, F., Bellotti, F., and Olivieri, M., Instruction level analytic prediction of parallel CPU architecture performance, *IIS '97*, Dec. 1997.
4. Migliardi, M., and Maresca, M., Modelling Instruction Level Parallel Architectures Efficiency in Image Processing Applications, *International Conference on High Performance Computing and Networking*, Vienna, Austria, April 1997.
5. MacDougall, M.H., Instruction-Level Program and Processor Modeling, *IEEE Computer*, July 1984
6. Lubeck, O.M, Luo, Y., and Wasserman, H.J. et al, An Empirical Hierarchical Memory Model Based on Hardware Performance Counters, *PDPTA'98*, Las Vegas, July 13-16, 1998.
7. Sun, X. H., Cameron, K. W., et al., A Hierarchical Statistic Methodology for Advanced Memory System Evaluation, accepted by *IPPS'99*, Sept. 1998.
8. Luo, Y. and Cameron, K.W., Instruction-level Characterization of Scientific Computing Application using Hardware Performance Counters, *Workshop on Workload Characterization at Micro-31*, Nov. 1998.
9. Bianchini, R., Lim, B., Evaluating the Performance of Multithreading and Prefetching in Multiprocessors, *Journal of Parallel and Distributed Computing*, N.37, p83-97, 1996.
10. Luo, Y., and Cameron, K. Instruction-level Performance Modeling and Characterization of Multimedia Applications, *Los Alamos Unclassified Technical Report #99-303*, Jan. 1999.
11. Hennessy, J.L., and Patterson, D.A., *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, Inc., p35-37, 1996.
12. Turner, S. (SGI/Cray), Private Communications, Mar. 1998.