LA-UR- 99-303

Approved for public release; distribution is unlimited.

CONF-990606--

Title:

Instruction-level Performance Modeling and Characterization of Multimedia Applications

Author(s): Yong Luo
Kirk W. Cameron

Submitted to: International Conference on Supercomputing ICS99
Rhodes, Greece
June 20-25, 1999

Los Alamos
NATIONAL LABORATORY

Instruction-level Performance Modeling and Characterization of Multimedia Applications

Yong Luo[‡] Kirk W. Cameron[†‡]

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

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

Abstract

One of the challenges for characterizing and modeling realistic multimedia applications is the lack of access to source codes. On-chip performance counters effectively resolve this problem by monitoring run-time behaviors at the instruction-level. This paper presents a novel technique of characterizing and modeling workloads at the instruction level for realistic multimedia applications using hardware performance counters. A variety of instruction counts are collected from some multimedia applications, such as RealPlayer, GSM Vocoder, MPEG encoder/decoder, and speech synthesizer. These instruction counts can be used to form a set of abstract characteristic parameters directly related to a processor's architectural features. Based on microprocessor architectural constraints and these calculated abstract parameters, the architectural performance bottleneck for a specific application can be estimated. Meanwhile, the bottleneck estimation can provide suggestions about viable architectural/functional improvement for certain workloads. The biggest advantage of this new characterization technique is a better understanding of processor utilization efficiency and architectural bottleneck for each application. This technique also provides predictive insight of future architectural enhancements and their affect on current codes. In this paper we also attempt to model architectural effect on processor utilization without memory influence. We derive formulas for calculating CPI_0, CPI without memory effect, and we quantify utilization of architectural parameters. These equations are architecturally diagnostic and predictive in nature. Results provide promise in code characterization, and empirical/analytical modeling.

Keywords: *Instruction level parallelism, multimedia applications, analytical modeling, performance characterization, performance prediction*

Introduction

Instruction-level modeling is not new. Work has been accomplished via two approaches. Some have attempted to characterize code or architecture analytically such as [12] and more recently [13,14]. Still others focus on modeling using complex formulas to characterize underlying hardware such as [11]. [15] shows that analytical modeling of these types began in early 1980s. Each of these papers however suffers from a lack of simple formulated equations to quantify relationships. Some also suffer from their inability to be validated. All too often these types of methods rely on inherently slow simulators to provide validation of theory. In our presentation, we use real code and "on-chip" hardware counter measurements in support of our equations and theory. We present empirically derived diagnostic and predictive equations validated without the aid of simulations. We provide these equations in the context of a generic superscalar microprocessor allowing for theory application across platforms.

This paper presents a new technique of characterizing applications at the instruction level using hardware performance counters. Through a series of simplifying assumptions, we derive and validate equations to diagnose bottlenecks and predict architectural influence on existing codes without the aid of a simulator. This technique has the advantage of collecting instruction-level characteristics in a few runs virtually without overhead or slowdown. The biggest benefit of applying this technique to multimedia applications, which usually are present without source code, is the capability of gathering detailed performance behavior at runtime without accessing difficult-to-obtain source codes. A variety of instruction counts can be utilized to calculate some average abstract workload parameters corresponding to microprocessor pipelines or functional units. Based on the microprocessor architectural constraints and these calculated abstract parameters, the architectural performance bottleneck for a specific application can be estimated. Meanwhile, the bottleneck estimation can provide suggestions about viable architectural/functional improvement for certain workloads. Eventually, these abstract parameters can lead to the creation of a complete analytical microprocessor pipeline model and memory hierarchy model. 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 a SGI R10000-based system, using the SGI performance counter tool *perfex* and its associated libraries. While targeting a specific architecture for example and analysis, the applied technique is general in nature. Current research includes applications on other processors.

This paper consists of two parts. In the first part, we present 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. In this paper, we focus on 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 for several typical multimedia applications in the second part of the paper.

We utilize the abstract workload parameters and methodology previously defined and validated to analyze the characteristics of several sample multimedia applications. Herein we discuss analytically drawn conclusions and interesting observations. We conclude with a discussion of overall observations and directions for future work.

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.

Application Dependent Parameters

In any modeling study, defining a good set of application or workload parameters poses a significant challenge [10]. We use a set of instruction-level parameters as described in [16] 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 [2]. We define the terms in Figure 1 for those queues to be described in the general microprocessor model. 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_{L1} and λ_{L2} refer to the occurrence of L1 and L2 misses. These are inclusive and a subset of overall memory instructions.

$$\lambda_f = \frac{\text{\# graduated instructions}}{\text{\# graduated FP instructions}}$$

$$\lambda_i = \frac{\text{\# graduated instructions}}{\text{\# graduated INT instructions}}$$

$$\lambda_m = \frac{\text{\# graduated instructions}}{\text{\# graduated memory instructions}}$$

$$\lambda_{L1} = \frac{\text{\# graduated instructions}}{\text{\# L1 cache misses}}$$

$$\lambda_{L2} = \frac{\text{\# graduated instructions}}{\text{\# L2 cache misses}}$$

Figure 1 λ equations

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.

CPU model without memory influence

In Figure 2 we portray a simplified version of an arbitrary superscalar microprocessor. In this 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. What follows are detailed

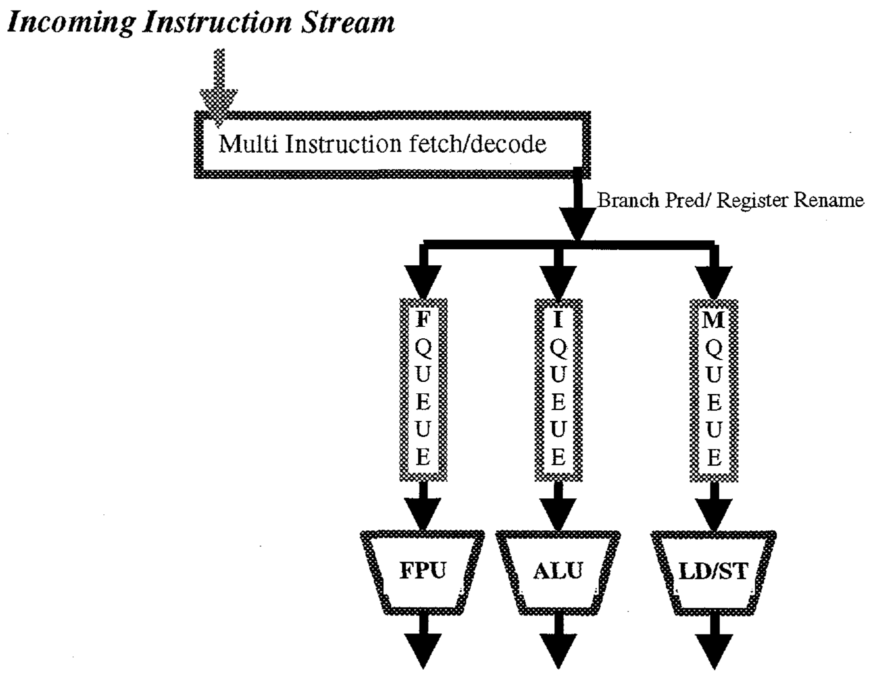
Incoming Instruction Stream
Multi Instruction fetch/decode
Branch Pred/ Register Rename
F QUEUE
I QUEUE
M QUEUE
FPU
ALU
LD/ST

Figure 2 General pipeline model for CPU only

explanations of the assumptions necessary to utilize the current model followed by derived equations. We will minimize these assumptions in future work. In the present case, for many applications, this modeling technique is very useful as will be shown when describing the analysis of the MIPS R10000.

Assumption 1 *Uniform distribution of instructions*

The λ values mentioned previously are conceptually determining a synthetic instruction stream based on measured values from a particular code. This stream is based on an average parameterization of the λ values over a uniformly distributed stream of instructions fed to the microprocessor. This assumption is necessary to simplify what would entail a modeling of the permutations of the instructions within the stream, an enormous amount of work and added complexity. We believe, and are supported by our results, that using streams in this manner produces a best-case scenario for instructions entering the microprocessor; and thus our equations for CPI_0 present a lower bound due to this assumption

Assumption 2 *λ values individually converge*

In [16] we show that for large problem sizes in scientific applications, λ values converge. This is intuitive if we consider application codes in general achieve a steady state of computation at some point in their execution. For multimedia applications, when source code is not readily available, we must vary input sizes instead of varying problem size within the code. Thus we cannot simply determine convergence of λ values by the previous method. Runtime profile data shows that I/O activities are negligible for all these experiments. By discounting startup effects, and I/O effects not related to the computational workload of a multimedia application, we can assume a steady state of computation is reached for a fairly large volume of input data. Thus, the values for λ used in our equations are the steady-state values for the code-application combination being modeled. In general, throughout this paper, when we discuss modeling of the processor, we refer to a modeling of this steady state.

Assumption 3 *Branch influence is negligible*

In a large amount of scientific codes, with particular exception to Monte Carlo based applications, processors are fairly successful at branch prediction. Branch modeling is a difficult endeavor. Luckily, if branches are predicted well (around 98% of the time), then we consider their effect as negligible. In truth, branch instructions have influence, but if they are small enough in proportion to the measured code that their effect is within counter tolerance, we chose to ignore them. This is the case with the modeled multimedia applications.

Assumption 4 *Icache effect is negligible*

With the size of today's icaches, it is no surprise that in our multimedia applications, and for well-structured codes in general, icache misses are limited to startup costs and thus negligible.

Assumption 5 *No data dependence*

It is fair to say that dependence modeling is an extremely difficult task due to the nature of dependencies. The goal in our modeling approach is to determine the stalls within a processor due to architectural constraints, not code dependencies. We also feel (again), that by modeling a CPI_0 without dependencies, we are in fact finding a lower bound. Further, if we can model all effects other than dependencies, then we will in fact be able to quantify dependencies themselves.

Assumption 6 *Infinite L1 cache (or no L1 misses)*

In this model, we wish to focus on contributions to stall that are not related to memory. We assume an infinite L1 cache so that no memory access can cause stalls within the processor, and we can model only the effect of the architectural constraints on performance.

Equation 1 *Conservation Equation*

$$1/\lambda_m + 1/\lambda_i + 1/\lambda_f = 1 \qquad (1)$$

In applications with a small percentage of branches and high branch prediction hit ratio, the execution of these applications can be virtually viewed as feeding integer, floating-point, and memory instructions into pipeline queues. At the end of each queue, functional units execute the instructions at a preset rate, e.g. two floating-point instructions per cycle, one memory instruction per cycle, etc. The out-of-order execution feature provides the ability of resolving data dependency within or between these pipeline queues. Therefore, the distributions or the mixture pattern of these three types of instructions in the application instruction flow shall essentially determine the instruction execution rate (*IPC*), ignoring the memory hierarchy effect. This is the well-known IPC_0 or CPI_0, which represents the effective application performance on a specific microprocessor without memory slowdown. The conservation equation mathematically defines this concept. The sum of the probabilities of each type of instruction must be 1. We should note that in the formulas for this model description we discuss in the context of one integer, one floating point, and one memory queue. The reasons for this will be obvious when we discuss the details of the MIPS R10000. For the sake of simplicity in presenting this work, we mention that these can easily be extended to model different numbers and sizes of queues provided the original assumptions are kept. For extension to other processors, these formulas may need to be syntactically modified, but will conceptually remain unchanged.

Equation 2 *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 graduate as well as the rate at which they are decoded giving:

$$G_x = \frac{\beta}{\lambda_x} - \Delta_x \qquad (2)$$

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 graduation 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. Informally, the growth rate for queue x is determined by the difference between the incoming rate of x instructions (β/λ_x) and the graduation rate of x instructions (Δ_x). We are interested in positive growth rates ($G_x > 0$) for each queue in question. This formula, along with our infinite L1 cache assumption, allows us to approach a lower bound for the widely discussed CPI_0 as we discuss in our next equation. 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 L1 cache). In particular, we use our growth rate 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.

$$\beta/\lambda_m + \beta/\lambda_i + \beta/\lambda_f = \beta$$

Let

$$\sum_x \Delta_x = \Delta_m + \Delta_i + \Delta_f \ .$$

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

$$G = \beta - \sum_x \Delta_x$$

Now we have three cases to discuss. For each of these cases, our modeling method could be extended for analysis, but we find this unnecessarily complicates formulas while not contributing to coverage of most types of processors.

G>0: If this case is found, this is a very inefficient processor. Here the decoding rate is greater than the peak graduation rate of the chip. This means as a steady state is reached, more instructions attempt to enter the processor's queues than could ideally be serviced - contributing directly to resource stalls within the chip. It should not be surprising that we do not know of any popular, current processors that perform in this manner. Hence we ignore this case in our modeling.

G=0: Now our decoding rate matches our peak graduation rate. This type of processor could at times suffer from the same inefficiencies as the previously discussed chip. The problem lies in instruction mix. In order to achieve no stalls here, the chip requires a perfect mix of instructions to match its functional units. Unfortunately this does not occur. Modeling this case is similar to the following case, so we do not need to ignore this case entirely.

G<0: Herein lies a quality of a good processor in today's market. The decoding rate is less than the graduation rate to provide more functional units than absolutely necessary. This provides for service of requests at an acceptable rate and implies the overall growth rate inside the processor is not positive.

Now, with our proper assumption, we can ease the analysis of queue limitations on processor efficiency:

if $\sum_x \Delta_x \geq \beta$ then $G_x \leq 0$ for at least one of x=m,i,f

Proof: From the conservation equation and previous definitions, $G_m + G_i + G_f \leq 0$. Generically, $G_{x1} + G_{x2} + G_{x3} \leq 0$. It follows that:

if $G_{x1}=G_{x2}=0$ then $G_{x3}\leq 0$

if $G_{x1}>0$, $G_{x2}=0$ then $G_{x3}\leq 0$

if $G_{x1}>0$, $G_{x2}>0$ then $G_{x3}\leq 0$

In these and all other cases, at least one growth rate is less than or equal to 0.

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.

Equation 3 *Lower bound for CPI_0*

Since we assume an infinite L1 cache, indicate no significant branching effect, and ignore 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. It is easy to describe this intuitively as the case where an infinite L1 cache is present. Following our previous assumptions, we give a formula to calculate this CPI_0 based on characteristics of the application and architecture under analysis. Once the *limiting factor* has been determined, we may proceed with this equation. Here, we briefly discuss its derivation.

We wish to derive this equation as generally as possible. Let us propose the concept of a period in cycles that is repeated by the processor for a particular code. This period is composed of two parts. Let C be this period. We define C as

$$C = C_{nostall} + C_{stall}\,.$$

Now, $C_{nostall}$ is the portion of this period in cycles during which no stalls occur due to "on-chip" resources. During this $C_{nostall}$, by our earlier assumptions and the underlying theory discussed so far, all non-*limiting factor* instructions are serviced at their preset graduation rates or incoming rates. But, we know we have a positive growth rate for instructions of type x. This means after the $C_{nostall}$ portion of our period, we require C_{stall} cycles to service the "left over" instructions dictated by the growth rate. So, C_{stall} depends on the length of $C_{nostall}$, and the growth and graduation rates of the queue dictated by the *limiting factor*. So, we can define C_{stall} with these terms:

$$C_{stall} = \frac{C_{nostall} * G_x}{\Delta_x}\,.$$

This gives a new equation for *C* that we simplify as

$$C = C_{nostall} + C_{nostall} * ({G_x}/{\Delta_x}) \tag{3a}$$

$$C = C_{nostall} * (1 + {G_x}/{\Delta_x})\,.$$

Now we must define the number of instructions that will graduate during C cycles. First let us consider the number of instructions to be serviced without stall effect. These are the instructions entering and graduating from the queues not associated with the *limiting factor*. We multiply the sum of these instructions by the number of cycles of no stall, $C_{nostall}$. This gives the first term of Equation 3b. The second term is given by the *limiting factor* instructions to be serviced during the entire period. These *limiting factor* instructions graduate at the hardware preset rate of Δ_x. Now we define and simplify N in equation form using j=i,m,f and x the *limiting factor* instruction i, m, or f:

$$N = \sum_{j \neq x} {\beta}/{\lambda_j} * C_{nostall} + \Delta_x * C\,, \tag{3b}$$

$$N = \sum_{j \neq x} \beta / \lambda_j * C_{nostall} + \Delta_x * (C_{nostall} + C_{nostall} * (G_x / \Delta_x)),$$

$$N = C_{nostall} * (\sum_{j \neq x} \beta / \lambda_j + \Delta_x * (1 + G_x / \Delta_x)).$$

We now have the number of cycles and instructions for a fixed repetitive period that is dependent upon both architecture and application. We define and simplify *CPI₀* now as cycles per instruction for this theoretically defined period:

$$CPI_0 = \frac{C_{nostall} * (1 + G_x / \Delta_x)}{C_{nostall} * (\sum_{j \neq x} \beta / \lambda_j + \Delta_x * (1 + G_x / \Delta_x))},$$

$$CPI_0 = \frac{1 + G_x / \Delta_x}{\sum_{j \neq x} \beta / \lambda_j + \Delta_x * (1 + G_x / \Delta_x)}.$$

Since $G_x = \beta / \lambda_x - \Delta_x$ from Equation 2,

$$CPI_0 = \frac{1 + \frac{\beta / \lambda_x - \Delta_x}{\Delta_x}}{\sum_{j \neq x} \beta / \lambda_j + (\Delta_x + \beta / \lambda_x - \Delta_x)},$$

$$CPI_0 = \frac{\Delta_x + \beta / \lambda_x - \Delta_x}{(\sum_{j=m,i,f} \beta / \lambda_j) * \Delta_x},$$

$$CPI_0 = \frac{\beta / \lambda_x}{\beta * \Delta_x},$$

$$CPI_0 = \frac{1}{\lambda_x \Delta_x}. \qquad (3)$$

This simplification is actually quite interesting as it shows *CPI₀* is dependent upon the product of probability of a *limiting factor* instruction and the associated graduation rate (in *CPI*) of the associated *limiting factor* queue. At this point, we stipulate that this is the formula for *CPI₀* when a *limiting factor* is present. If all growth rates are negative, there is no single *limiting factor*. *CPI₀* can then be calculated as 1/β, the ideal *CPI₀*. In the scenario of more than one positive growth rate, the derivation process of *CPI₀* is essentially the same and varies just slightly in details.

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. To validate our model, we chose to use real synthetic codes on real processors using hardware performance counters to provide necessary counts. 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. The assumptions are necessary for simplification, but not overly limiting.

The MIPS R10000 is a 4-way superscalar microprocessor with an two integer, two floating point and one memory functional unit. It supports up to 32 outstanding instructions across queues, and up to 16 instructions in each of its integer, floating point, and memory queues [8]. We use $\Delta_m=1$, $\Delta_i=2$, $\Delta_f=2$ for our experiments and according to [9] for ideal instruction mixes. In some special cases, $\Delta_f=1.5$ is used to compensate for instruction streams that vary from ideal. In particular, the R10000 can execute only one floating point multiply and one add simultaneously. So, for mixes of instructions that overwhelm (for example) the floating point multiply unit, we sustain a lower Δ_f as a result.

Our approach to validation is simple to discuss, but quite complicated to implement. We omit details of the implementation while presenting the important details of how our approach validates the model. We have created code that we can modify to ensure certain instruction streams are fed to the microprocessor. Our code has the

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_f	G_m	G_i						
fff_*+*	1.9578	-0.9945	-1.9819	f	1.0107	1.5000	0.6622	0.6596	0.40%
ifff_+*+	0.9761	-0.9959	-0.9942	f	1.3440	1.5000	0.5192	0.4960	4.47%
ii	-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%

following properties: any instruction mix is uniformly distributed, λ values are constant, branch influence is negligible, icache effect is negligible, there is no data dependence among instructions, there are no L1 cache misses. We use direct hardware counter measurements to ensure these assumptions are met. 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".

Table 1 shows our calculated and measured CPI_0 are within the tolerance of the counters themselves, implying they are quite accurate. Thus, with our assumptions, we are able to model CPI_0 with a great deal of accuracy. Since our theory is general in nature, we believe validation on other processors will support these findings.

Application Description

GSM (Global System for Mobile Communication) Vocoder is a set of standardized voice encoder/decoder for mobile communication. The code used in this study is a public domain C code package developed by The Communications and Operating Systems Research Group (KBS) at the Technische Universitaet Berlin. It's an implementation of the European GSM 06.10 provisional standard for full-rate speech transcoding, which uses RPE/LTP (residual pulse excitation/long term prediction) coding at 13 kbit/s. GSM 06.10 compresses frames of 160 13-bit samples (8 kHz sampling rate, i.e. a frame rate of 50 Hz) into 260 bits. For compatibility with typical UNIX applications, this implementation turns frames of 160 16-bit linear samples into 33-byte frames (1650 Bytes/s) [3].

MPEG-2 video decoder is the second application studied in this paper. MPEG-2 Video is a generic method for compressed representation of video sequences using a common coding syntax. It has been standardized by the International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC), in collaboration with the International Telecommunications Union (ITU). The MPEG-2 concept is similar to MPEG-1, but includes extensions to cover a wider range of applications. The primary application targeted during the MPEG-2 definition process was the all-digital transmission of interlaced broadcast TV quality video at coded bitrates between

4 and 9 Mbit/sec. However, the MPEG-2 syntax has been found to be efficient for other applications such as those at higher bit rates and sample rates (e.g. HDTV) [4].

MPEG audio encoder/decoder (Layer 2) is examined as a separate application in this paper. This software implements levels I and II psychophysical auditory models as described in the ISO 3-11171 rev 1 standard. The input/output audio data may either be headerless raw 16 bit data or alternatively an AIFF formatted file (Audio Interchange File Format) with certain limitations [5].

A public domain speech synthesizer rsynth is used in this experiment. This is a text to speech system produced by integrating various pieces of code and tables of data. The SGI port was developed by Nick Ing-Simmons of Texas Instruments. It claims to be the best public domain text to speech synthesizer. The SGI port package (rsynth 1.1) is obtained from Tom Benoist's "strange software page" [7].

The last multimedia application studied in our experiment is RealPlayer™ version 5.0 for SGI IRIX system. It is perhaps one of the most popular multimedia applications used.

Performance Characterization on R10000

In this section we discuss the present practicality of this instruction-level modeling/characterization technique. We concentrate on applying the newly developed methodology to analyzing performance characteristics of our sample multimedia applications. Efficiently measuring these instruction-level workload parameters is the key component to code characterization. On-chip hardware performance counters widely available on recent generation microprocessors become good candidates for extracting these functional-unit-related parameters.

For the MIPS R10000 and our associated multimedia 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 within 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. Therefore, the bottleneck analysis based on growth rate should still reflect one aspect of the real code characteristics.

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. We discuss these in the context of each code below. The key to this methodology is to estimate which instruction-associated λ value can cause stall of the microprocessor due to the limitation of architectural constraints.

GSM Vocoder

Table 2 lists the growth rates for GSM vocoder, under different input conditions. It is clear that no floating point operation is invoked in GSM decoding and the overall decoding performance (*CPI*) is slightly different from the encoding behavior. Their performances remain the same for different input data (E01, E02, E06). A single positive growth rate in the integer queue indicates the likely bottleneck for on-chip performance. The greater numbers in the decoding integer queue growth rates reflect that this queue may be more of a problem for decoding than for encoding. Other performance counter data also show the following for the GSM vocoder: nearly 100% L1 cache hit rate; branch ratio lower than 4%; Icache hit rate higher than 99.8%. These data guarantee that the growth-rate-based analysis leads to a good bottleneck estimate. Memory hierarchy effect apparently is insignificant in this application as nearly 100% of memory accesses can be satisfied by L1 cache data.

Table 2 GSM Vocoder Growth Rates

	Gf	Gm	Gi	CPI
E01_dec	N/A	-0.604	1.507	0.722
E02_dec	N/A	-0.624	1.529	0.714
E06_dec	N/A	-0.613	1.512	0.749
E01_enc	-1.619	-0.126	0.588	0.693
E02_enc	-1.619	-0.150	0.607	0.691
E06_enc	-1.621	-0.158	0.621	0.699

MPEG-2 Video Decoding

MPEG-2 video decoding growth rate data are exhibited in Table 3. Meanwhile, performance counter data demonstrate the following overall observations for this code: L1 hit rate >99.4%, Icache hit rate >99.9%, and branch rate < 9%. Again, it indicates a good sign for employing the growth-rate-based bottleneck analysis. Both MPEG1 and MPEG2 video decoding are performed in this experiment (IBM MPEG1 samples and Tektronix standard MPEG2 test streams [18] are respectively used for MPEG1 and MPEG2 decoding). Different video quality (B for Betascam, V for VHS, and D for Digital), different video formats (N for NTSC 525-line video, PAL for PAL 625-line video), and different bit rates (15 for 1.5Mbit/s, 30 for 3.0Mbit/s, ...) are compared in this MPEG video

Table 3 MPEG Video Decoding Growth Rates

	Gf	Gm	Gi	CPI
MPEG1(B-15)	-1.078	0.147	-0.411	0.612
MPEG1(B-30)	-1.141	0.179	-0.404	0.638
MPEG1(D-15)	-1.064	0.150	-0.434	0.622
MPEG1(V-15)	-1.086	0.135	-0.390	0.607
MPEG2(PAL-15)	-1.174	0.124	-0.281	0.609
MPEG2(PAL-40)	-1.164	0.102	-0.256	0.594
MPEG2(PAL-60)	-1.184	0.114	-0.261	0.608
MPEG2(N-15)	-1.170	0.137	-0.294	0.600
MPEG2(N-40)	-1.138	0.099	-0.280	0.597
MPEG2(N-60)	-1.166	0.122	-0.287	0.606

decoding process. It is interesting to see that the same application actually behaves in a slightly different way, probably due to the invoking of different algorithms incorporated in the code. However, the memory queue appears to be the likely architectural bottleneck for all situations tested, based on the positive growth rates shown in the table.

MPEG Audio

Table 4 lists the growth rates of MPEG audio encoding/decoding. In addition to this table, we've also noticed the following: L1 hit rate: encoding > 97%, decoding > 96%, Icache hit rate > 99.9% branch rate: encoding < 8%, decoding < 6.5%. We consider these Icache hit rates and branch rates not bad enough to prevent us from using the growth-rate-based analysis. The input data used in this experiment are also obtained from Tektronix standard test stream library [18]. Digital streams of 100Hz, 15kHz tonal tone signal, a synthetic multi-frequency sound, and a music stream are used here. It seems that MPEG audio encoding/decoding behavior only varies

Table 4 MPEG Audio Encoding/Decoding Growth Rates

	Gf	Gm	Gi	CPI
100Hz_enc	-1.261	0.242	-0.296	0.659
15k_enc	-1.298	0.260	-0.278	0.651
music_enc	-1.261	0.252	-0.309	0.659
sync_enc	-1.306	0.289	-0.283	0.640
100Hz_dec	-1.495	0.141	0.112	0.699
15k_dec	-1.530	0.191	0.092	0.673
music_dec	-1.519	0.129	0.133	0.687
sync_dec	-1.483	0.160	0.065	0.629

Table 5 Rsynth Growth Rates

	Gf	Gm	Gi	CPI
test1	-0.379	0.389	-1.526	0.683
test2	-0.424	0.300	-1.364	0.703

slightly on its input data. As for the likely architectural bottleneck, the memory queue is the only one for encoding. However, for decoding, it becomes a bit complex as it has both G_m and G_I positive. We do not see a general pattern for decoding from our simple experiment data, but it is equally likely that memory, integer and total "on-the-fly" instructions cause processor stalls during code execution.

rsynth and RealPlayer™

It turns out that the rsynth and RealPlayer™ performance are a little more complicated than the rest of the codes due to higher branch rates (as high as 12.9 % for rsynth and nearly 22% for RealAudio™ playback, 11% for RealVideo™ playback). However, we can treat correctly-predicted branches as NOPS instructions which do not have any impact on the functional queues but change the overall outgoing instruction rate. It would still be beneficial to list their growth rates (Table 5 for rsynth and Table 6 for RealPlayer™) to see their instruction distributions across functional units.

Both rsynth and RealPlayer™ show a high density of memory instructions as their G_m's confirm. On the other hand, RealPlayer™ seems to be experiencing more memory instructions and branches (twice as high as for video) for audio-only playback (bottom half of Table 6, _ra inputs). This is probably the main reason RealPlayer™ has over 50% higher CPI for audio-only playback.

Table 6 RealPlayer Growth Rates

	Gf	Gm	Gi	CPI
Frosty_rv	-1.841	0.809	-0.416	1.061
holynight_rv	-1.964	0.652	-0.093	1.011
soxmas_rv	-1.927	0.762	-0.254	1.014
sheavy_ra	-1.964	0.882	-0.773	1.592
sscloud_ra	-1.953	0.920	-0.824	1.646
theme_ra	-1.961	0.894	-0.803	1.603
wizISDN_ra	-1.928	0.814	-0.752	1.570

Conclusions

We have developed a growth-rate-based performance characterization/modeling method, using hardware performance counter data as input parameters. We have provided direct validation (with 5% errors) of an analytical model to calculate a lower bound of CPI_0 for a generic microprocessor. We showed the diagnostic qualities of these models when applied to the MIPS R10000 microprocessor. Unlike most analytical counterparts, this model is validated with real code on real machines using on-chip performance counters. We then applied this technique to some typical multimedia applications using hardware performance counters. These abstract workload parameters provide us some new insights about the performance behaviors of multimedia applications, which usually depend on the input of the application. Our simplistic queue modeling also provides a picture of the utilization of processor architectural features. We have made significant contributions in this research toward a new avenue of analytical modeling with the capability of direct validation without the use of simulators and have shown their application on multimedia codes. Dependencies need to be studied further using our techniques and additional techniques to quantify dependencies will be attempted. Latency and its influence on current models are also of interest to us. Modeling of branches should be incorporated in the model to extend its applicability. Overall, we will endeavor to provide useful formulas for those interested in quickly characterizing workloads against architectures with the hope of predicting the influence of architectural advances on current code.

References:

1. 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.
2. Bianchini, R., Lim, B., *Evaluating the Performance of Multithreading and Prefetching in Multiprocessors,* Journal of Parallel and Distributed Computing, N.37, p83-97, 1996.
3. Degener, J. and Bormann, C., *The README file of public domain GSM package,* ftp://ftp.cs.tu-berlin.de/
4. MPEG Software Simulation Group website: http://www.mpeg.org/MSSG/
5. Pan, D., *Working Draft of MPEG/Audio Technical Report,* http://www.iuma.com/audio_utils/converters/
6. Sun, X. H., Cameron, K. W., et al., A Hierarchical Statistic Methodology for Advanced Memory System Evaluation, submitted to IPPS'99, Sept. 1998.
7. Tom Benoist, "strange software page", http://www.webcom.com/ie/benoist/
8. Schwarzmeier, J. (SGI/Cray), *Private Communications,* Sept. 1997.
9. Turner, S. (SGI/Cray), *Private Communications,* Mar. 1998.
10. Sorin, D.J., Pai, V.S., Adve, S.V., Vernon, M.K., and Wood, D.A., Analytic Evaluation of Shared-Memory Systems with ILP Processors, 25th Annual International Symposium on Computer Architecture, June 1998.
11. Albonesi, 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.
12. 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.
13. De Gloria, A., Ancarani, F., Bellotti, F., and Olivieri, M., Instruction level analytic prediction of parallel CPU architecture performance, IIS '97, Dec. 1997.
14. 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.
15. MacDougall, M.H., *Instruction-Level Program and Processor Modeling*, IEEE Computer, July 1984
16. 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.
17. Martin, R., Chen, Y-C., and Yeager, K., *MIPS R10000 Microprocessor User's Manual*, Version 1.1, Jan. 1996.
18. Tektronix Standard MPEG test stream library, ftp://ftp.tek.com/tv/test/streams/Element/index.html