Architectural timing verification of CMOS RISC processors

by P. Bose
S. Surya

We consider the problem of verification and testing of architectural timing models ("timers") coded to predict cycles-per-instruction (CPI) performance of advanced CMOS superscalar (RISC) processors. Such timers are used for pre-hardware performance analysis and prediction. As such, these software models play a vital role in processor performance tuning as well as application-based competitive analysis, years before actual product availability. One of the key problems facing a designer, modeler, or application analyst who uses such a tool is to understand how accurate the model is, in terms of the actual design. In contrast to functional simulators, there is no direct way of testing timers in the classical sense, since the "correct" execution time (in cycles) of a program on the machine model under test is not directly known or computable from equations, truth tables, or other formal specifications. Ultimate validation (or invalidation) of such models can be achieved after actual hardware availability, by direct comparisons against measured performance. However, deferring validation solely to that stage would do little to achieve the overall purpose of accurate pre-hardware analysis, tuning, and projection. We describe a multilevel validation method which has been used successfully to transform evolving timers into highly accurate pre-hardware models. In this paper, we focus primarily on the following aspects of the methodology: a) establishment of cause–effect relationships in terms of model defects and the associated fault signatures; b) derivation of application-based test loop kernels to verify steady-state (periodic) behavior of pipeline flow, against analytically predicted signatures; and c) derivation of synthetic test cases to verify the "core" parameters characterizing the pipeline-level machine organization as implemented in the timer model. The basic tenets of the theory and its application are described in the context of an example processor, comparable in complexity to an advanced member of the PowerPC™ 6XX processor family.

1. Introduction

The process of architectural simulation and verification forms a major focus within today's VLSI processor design programs. With increasing degrees of execution

0018-8646/95/$3.00 © 1995 IBM

concurrency and attendant levels of design complexity, the relative proportion of total effort devoted to pre- and post-fabrication test pattern generation and simulation-based verification is constantly on the rise. The ever-increasing level of integration afforded by custom and semicustom CMOS processor technology has led to the incorporation of deeper pipelines (leaner logic levels) and complex control logic to coordinate instruction and data flow through multiple (concurrent) functional units. Alternate, successive implementations of the same instruction-set architecture (e.g., POWER or PowerPC™ architectures [1, 2]), with varying levels of complexity (e.g., the PowerPC 601™, PowerPC 603™, PowerPC 604™, and PowerPC 620™ processors announced by the Somerset Design Center in Austin, Texas), help expose the need for a robust architectural verification methodology in a more vivid manner. *Functional* verification refers to validating an implementation with respect to an "expected," fault-free register or memory value-transition sequence, which would result in correct final register and memory states, for a given set of architectural test programs. In current industrial practice (e.g., [3]), random architectural test program generation and simulation methods are commonly used to "verify" the functional correctness of implemented instructions (applied singly or within a sequence of other operations). Classical fault-model-based functional test generation methods (e.g., [4, 5]) are occasionally used in essence, if not in full, to "bias" the architectural test sequence generation toward a few typical failure modes. Recent advances in formal verification techniques (e.g., [6–8]) seem to show promise.

With the current growth in superscalar and super-pipelined processors (RISC or CISC), the need for pre-hardware architectural *timing (or performance) verification* is of growing importance. Thus, for example, it is not enough to verify that the mapped (equivalent) test case for the assignment statement $A = B + C$ "works" correctly in the functional sense; we must also be able to verify the correctness of the exact cycle count for its execution. In initial generations of microprocessor design (especially for simple RISC processors such as the IBM 801 [9]), such timing or performance verification requirements were minimal because of the lack of concurrent (pipelined) structures; in a classical von Neumann machine, which uses single-cycle, nonpipelined instruction execution, the dynamic instruction count is necessarily a close correlator of the cycle count, irrespective of the nature of the test program. In modern VLSI processors, because of multiple dispatch modes and concurrent pipeline execution with out-of-order execution modes, the "science" of deriving performance-verification programs (PVPs) in addition to the traditional architectural verification programs (AVPs) (see [1], Section IV) is becoming increasingly important. In this paper, we attempt to present the theory and application of PVP generation as it is used in our current superscalar timing model validation methodology.

In prior work[1] [10–12], we briefly described alternate methods used in current industrial practice, for evaluating cycles-per-instruction (CPI) performance for superscalar machine models. A workload- or benchmark-driven *timer* is a cycle-by-cycle (timing) simulator of a candidate processor organization, with a program workload as the driving input. If this input is a *dynamic* execution trace, we refer to the tool as a *dynamic* timer. On the other hand, *static* timers evaluate program execution time by analyzing a static program listing (high-level, intermediate, or assembly/machine code). In either case, timers do not carry out actual functional simulation of the workload; only the cycle-by-cycle timing behavior of the concurrent pipeline structures is simulated. A *functional simulator*, on the other hand, is a sequential (i.e., one instruction at a time) simulation of an idealized von Neumann machine implementing the candidate (instruction set) architecture. One of the outputs of such a functional simulator is in fact a dynamic trace, which can be used to drive a dynamic timer. Clearly, in "timing" such a trace, the dynamic timer implicitly assumes the existence of a *compiler*, which created the original machine program, executed by the functional simulator. The (optimization) parameters used in the compilation process are fixed (i.e., unmodifiable), as far as the dynamic timer is concerned. The parameters which can be set and changed easily for such a timer are usually limited to processor organizational parameters, e.g., queue/buffer lengths, pipeline depths, cache latency parameters, branch prediction-related controls, bus/dispatch bandwidths, and various context-sensitive processing switches (on/off or Boolean flag parameters). In some recent, truly programmable timers (e.g., the BRAT timer in [13]), the instruction set architecture itself is made available to the user as a global parameter; a whole range of similar or related processor chips can be modeled by "programming" a range of such organizational parameters. By using such dynamic timers, the effect of alternate compiler optimization strategies on processor performance can be studied only by regenerating the trace for alternatively compiled modules. In the case of a static timer, on the other hand, the compiler optimization parameters can (at least in part) be made a subset of the timer parameters for the chip designer to experiment with. However, even in infinite cache mode, such static estimators tend to have less accuracy than trace-driven dynamic timers, though they are usually much faster[1] [10, 11].

Trace-driven dynamic timers can, in principle, be made as accurate as the underlying pipelined execution model, with designer-specified execution semantics. In this paper,

[1] J.-D. Wellman, "Cycles-per-Instruction Estimation Methods," IBM internal project report, September 1992, revised November 1993.

we restrict ourselves to such dynamic timers only; from this point onward, we omit the "dynamic" qualifier when referring to timers. In the early or intermediate stages of processor design, timers are useful for making design trade-offs and parameter sizing. Later, when the machine and timer models have stabilized, accurate pre-hardware projections are made using benchmark-driven timer runs. A crucial problem in this context is that of testing or validating the timer model against a "gold" processor model, i.e., one which is (conceptually) the exact, designer-specified execution model of the processor. Detailed execution semantics (using a formal hardware description language, such as the IBM-internal DSL, or the industry-standard VHDL) are available as part of the design, for purposes of simulation and synthesis. Formal timing verification schemes are therefore possible, in principle; but owing to complexities in global chip specification and lack of robust theories, we seldom see such an investment made in real, deadline-oriented processor development projects. Also, depending on the language formalism used, detailed, timer-like pipeline timing information is often not explicitly available through simulation models based on such formal descriptions. We have therefore taken the approach of devising robust performance (timing) validation strategies, applied to our fast timer models.

The sources of inaccuracies in a timer model are 1) modeling errors due to programming mistakes; 2) errors due to misinterpretation of informally specified execution semantics; and 3) data-sensitive execution semantics which are usually considered to be beyond the range of modeling capability of timers. An example of the third category of inaccuracies is a case in which alternate paths within a staged pipeline data path are followed, depending on the value range of the operand (register or memory). Such instances are usually infrequent enough that they may be ignored in analyzing performance behavior for realistic program benchmarks. We therefore limit our attention in this paper to instances 1 and 2. Since dynamic traces can be millions (and possibly billions) of instructions long, it is impractical for a designer to go over the entire cycle-by-cycle listing to identify such defects. A robust test/verification methodology, based on a tailored test case suite generated from higher-level application kernels (loops), is described in this paper. We present this method in terms of an example RISC superscalar processor, patterned after the RS/6000 [14], but with an added organizational feature: the instruction completion (reorder) buffer mechanism, to control out-of-order *finish* and in-order *completion* [15, 16]. We refer to this machine as the ERISC (extended RISC) machine. The machine organization for ERISC was defined specifically for the purposes of this paper; it is not based on an actual product. In terms of organizational complexity, ERISC is similar to an advanced member of the PowerPC 6XX family, such as the PowerPC 604 or the PowerPC 620 RISC microprocessors,[2] except that the ERISC has only one integer unit, which performs all load/store operations as well, like the original RS/6000 [14]. The trace-driven timer for ERISC was derived by extending the TRISC timer[3] used earlier[1] [10, 11] to study the accuracy of static timers. It is the same type of cycle timer used in the initial "research" superscalar timers [17]. In an earlier conference paper [18], a preliminary description of the performance fault models and levels of model validation used in our approach was provided. In this paper, we build on that theory and develop the practical application more fully. We develop a new notion of output behavioral testing which uses the cause–effect defect characterization principle in two ways: transient mode testing and steady-state parametric testing. In the latter part of Section 4, we illustrate the application of the experimental validation approach to *real* processors by presenting results obtained with the PowerPC 601 processor model and its timer.

2. The ERISC machine and its timer

• *ERISC core processor organization*

The example RISC machine (ERISC) [18] used for the purposes of this paper has a typical superscalar organization (**Figure 1**), implementing the POWER architecture [1]. All functional operations (FIX or FLT) are register-to-register, with two sources and one destination, explicitly specified. The instruction dispatch buffer can hold up to d instructions, which is a timer parameter. Every cycle up to three instructions can be dispatched, one to each of the functional units: BRN, FIX, and FLT. Floating-point load and store instructions are processed by FIX (the fixed-point unit) for address generation, prior to cache request. The instruction IDs for such instructions are dispatched to FLT (the floating-point unit) as well, to aid synchronization. FIX is capable of doing one agen (address generation) per cycle; this is matched by a single-port level-1 cache. The cache maintains a pending store queue (PSQ) to hold processed stores waiting to write into the data cache array. Actual synchronization is effected via register renaming. Both FIX and FLT (source and destination) registers are subject to dynamic renaming. FIX and FLT each have their own set of physical (rename) buffers, the sizes of which are timer parameters. The architected register file is updated during actual instruction completion. Instruction execution can be out of order, but instruction dispatch and actual completion is in order. The in-order dispatch and

[2] In this document, the terms "PowerPC 601 RISC microprocessor" and "601" are used to denote a microprocessor from the PowerPC Architecture™ family. Similarly, the terms "603," "604," "620," and "6XX" are used only as abbreviated notations for the corresponding microprocessor.

[3] P. Bose, "The TRISC Machine Architecture and Timer," IBM internal document, Thomas J. Watson Research Center, Yorktown Heights, NY, November 1991.

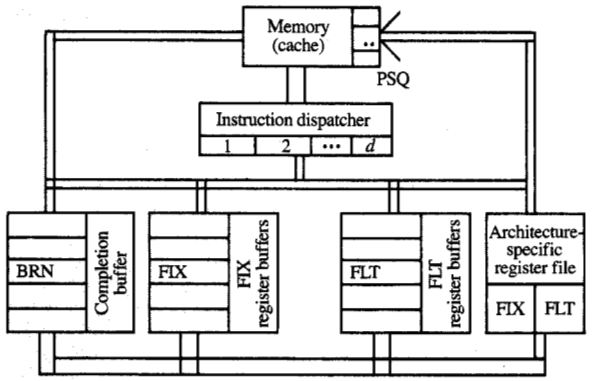


Figure 1

ERISC processor organization (timer model).

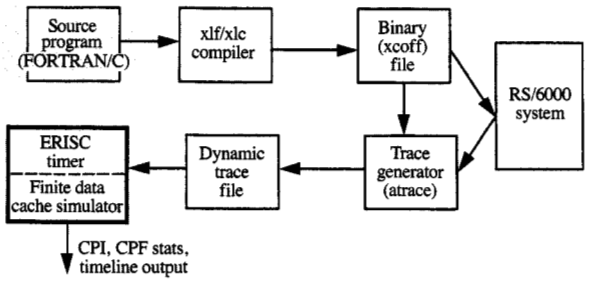


Figure 2

ERISC timer and associated tools and files complex.

completion mechanism [15] is managed using a completion buffer or queue, maintained and controlled by the branch-and-condition unit, BRN. This mechanism is similar to the completion buffer scheme described for the PowerPC 603 machine [16]; the in-order completion mechanism facilitates implementation of precise interrupts, with out-of-order execution modes present in the overall processor.

The organization parameters primarily considered in this paper are as follows:

1. Effective number of (one-cycle) pipeline stages, p_1 and p_2, respectively, in the FIX and FLT units. (The BRN unit has, effectively, a one-stage pipe.)
2. The sizes of the instruction queues q_{brn}, q_{fix}, and q_{flt}, associated with the three functional units; the size of the completion buffer (instruction sequencing table), q_{ist}, and the size of the pending store queue, q_{store}.
3. The cache access latency, c, in machine cycles. (Infinite cache model is assumed.)
4. The dependent bubble parameters, b_1 and b_2, respectively, where b_1 (b_2) is the number of pipeline bubbles in cycles caused by a consecutive dispatch sequence of two *dependent* fixed (floating)-point instructions, where the second instruction is data-dependent—via register interlocks—on the first one.
5. The number of FIX and FLT rename buffer registers, R_{fix} and R_{flt}.

• *ERISC timer*

As stated in Section 1, a *timer* is a cycle-by-cycle simulator of a candidate machine organization. Its main purpose is to print out an overall cycles-per-instruction (CPI) performance figure for a given instruction trace. As a side benefit, the detailed timer outputs are useful in identifying compiler deficiencies and organizational bottlenecks. In pre-hardware evaluations, dynamic instruction traces are usually generated by a separate *instruction set simulator*. **Figure 2** shows the software organization of the tools used to drive the ERISC timer. Since the instruction set architecture assumed is that of an RS/6000 [1, 14], we are able to use available compilers [19] and trace generators[4] for generating traces using an existing hardware platform. The actual ERISC timer program is written in Pascal (with an alternate implementation in C) and runs on an RS/6000 system. For the purposes of this study, we have used an *infinite cache* timer model, in which memory reference paths are pipelined, resulting *always* in cache hits, with a fixed latency of c cycles.

Example test case input and associated timer output

We present a specific loop test case, the daxpy test case with timer output, to introduce several concepts and definitions.

"Daxpy" is the key loop within the well-known floating-point benchmark of Linpack. The FORTRAN specification of daxpy is

```
do i = 1, n
  x(i) = x(i) + s · y(i)
enddo
```

where the one-dimensional arrays x, y and the scalar s are declared to be double-precision floating-point variables.

[4] "Atrace," IBM internal software, 1991; author: R. Nair, IBM Thomas J. Watson Research Center, Yorktown Heights, NY.

The corresponding compiled code, in mnemonic notation, per iteration, is as follows:

```
A:    lfd    fr1, r6, 0×8
B:    fma    fr1, fr0, fr2, fr1
C:    lfdu   fr2, r5, 0×8
D:    stfdu  fr1, r6, 0×8
E:    bc
```

The alphabetical labels A, B, C, ··· are assigned to successive instructions in the execution trace in alphabetical order, with Z being succeeded by a, b, c, ···, and z wrapping around again to A. Thus, in the actual trace, F stands again for the first load instruction (lfd), G for the fma, etc.

The ERISC cycle-by-cycle timer output for the first 40 cycles is shown in **Figure 3**. The labeled functional units or queues, with dashes (–) representing individual stages, are explained below (see also the subsection on the ERISC core processor organization):

PIB: Primary instruction buffer: nominally set to a size of 12 for this run.

IST: The instruction sequencing table or completion buffer (queue), nominally set to 16 for this run.

LSTQ: The load-store instruction queue, for holding load and store instructions for the fixed-point unit FIX; the size of this queue is determined by the q_{fix} timer parameter.

FPU_IQ: The floating-point instruction queue for the floating-point unit, FLT; the size of this queue is determined by the q_{flt} timer parameter.

CA: The on-chip, level-1 cache access pipe, nominally set to one stage for this run; this is equivalent to setting the timer parameter c to 1.

STQ: The pending store queue, which holds stores waiting for data, prior to writing in cache; the size of this queue is determined by the q_{store} timer parameter.

In cycle 1 of Figure 3, the first three instructions in the buffer (A, B, C, representing the lfd, fma, and lfdu) are dispatched, with the corresponding instruction IDs allotted to the completion buffer slots. A goes to the first stage ("decode") of FIX, while C is queued in LSTQ, and B goes to the first stage ("decode") of FLT. Instruction B (fma) must wait in "decode" for four cycles because it has to wait for one of its operands to be produced by A (lfd). It then advances along the FLT execution pipe (set to four stages for this run). The lfd A moves from "decode" to "address gen" to "request to cache" to the actual cache access stage (CA). Following this, A is "finished,"

Cycle	PIB	IST	LSTQ	FIX	FPU_IQ	FLT	CA	STQ
0	-------EDCBA	----------	----	-----	--------	-----	-	--------
1	----------ED	ABC-------	C---	A----	--------	B----	-	--------
2	-------JIHGF	ABCDE-----	D---	CA---	--------	B----	-	--------
3	----------JI	ABCDEFGH--	FH--	DC--A	G-------	B----	-	--------
4	-------ONMLK	ABCDEFGHIJ	HI--	FD--C	G-------	B----	A	--------
5	----------ON	-BCDEFGHIJKLM	IKM-	HF--D	L-------	GB---	C	--------
6	-------TSRQP	-BCDEFGHIJKLMNO-	KMN-	IH--F	L-------	G-B--	D	--------
7	---------TSR	QBCDEFGHIJKLMNOP	MNP-	KI--H	LQ------	G--B-	F	D-------
8	---------TSR	QBCDEFGHIJKLMNOP	NP--	MK--I	Q-------	LG--B	H	D-------
9	----------T	QRSDEFGHIJKLMNOP	PRS-	NM--K	Q-------	L-G--	I	D-------
10	----------T	QRSDEFGHIJKLMNOP	RS--	PN--M	Q-------	L--G-	D	I-------
11	-------YXWVU	QRST--GHIJKLMNOP	S---	RP--N	Q-------	L---G	K	I-------
12	----------YX	QRSTUVW-IJKLMNOP	UW--	SR--P	V-------	QL---	M	IN------
13	----------Y	QRSTUVWXIJKLMNOP	WX--	US--R	V-------	Q-L--	I	N-------
14	-------dcbaZ	QRSTUVWXY--LMNOP	X---	WU--S	V-------	Q--L-	P	N-------
15	---------dcb	QRSTUVWXYZaLMNOP	Z---	XW--U	a-------	VQ--L	R	NS------
16	----------d	QRSTUVWXYZabcNOP	bc--	ZX--W	a-------	V-Q--	U	NS------
17	----------d	QRSTUVWXYZabcNOP	c---	bZ--X	--------	aV-Q-	N	S-------
18	-------ihgfe	QRSTUVWXYZabcd--	----	cb--Z	--------	a-V-Q	W	SX------
19	----------ih	g-STUVWXYZabcdef	g---	ec--b	f-------	a--V-	Z	SX------
20	----------i	ghSTUVWXYZabcdef	h---	ge--c	--------	fa--V	S	X-------
21	-------nmlkj	ghi--VWXYZabcdef	----	hg--e	--------	f-a--	b	Xc------
22	----------nm	ghijkl-XYZabcdef	l---	jh--g	k-------	f--a-	e	Xc------
23	----------n	ghijklmXYZabcdef	m---	lj--h	--------	kf--a	X	c-------
24	-------srqpo	ghijklmn--abcdef	----	ml--j	--------	k-f--	g	ch------
25	----------sr	ghijklmnopq-cdef	q---	om--l	p-------	k--f-	j	ch------
26	----------s	ghijklmnopqrcdef	r---	qo--m	--------	pk--f	c	h-------
27	-------xwvut	ghijklmnopqrs--f	----	rq--o	--------	p-k--	l	hm------
28	----------xw	-hijklmnopqrstuv	v---	tr--q	u-------	p--k-	o	hm------
29	----------x	whijklmnopqrstuv	w---	vt--r	--------	up--k	h	m-------
30	-------CBAzy	wx--klmnopqrstuv	----	wv--t	--------	u-p--	q	mr------
31	----------CB	wxyzA-mnopqrstuv	A---	yw--v	z-------	u--p-	t	mr------
32	----------C	wxyzABmnopqrstuv	B---	Ay--w	--------	zu--p	m	r-------
33	-------HGFED	wxyzABC--pqrstuv	----	BA--y	--------	z-u--	v	rw------
34	----------HG	wxyzABCDEF-rstuv	F---	DB--A	E-------	z--u-	y	rw------
35	----------H	wxyzABCDEFGrstuv	G---	FD--B	--------	Ez--u	r	w-------
36	-------MLKJI	wxyzABCDEFGH--uv	----	GF--D	--------	E-z--	A	wB------
37	----------ML	wxyzABCDEFGHIJK-	K---	IG--F	J-------	E--z-	D	wB------
38	----------M	wxyzABCDEFGHIJKL	L---	KI--G	J-------	E---z	w	B-------
39	-------RQPON	M--zABCDEFGHIJKL	----	LK--I	--------	JE---	F	BG------
40	----------RQ	MNOP--CDEFGHIJKL	P---	NL--K	--------	QJE--	I	BG------

Figure 3

ERISC timer cycle-by-cycle output for daxpy test case.

followed by "completion," which is manifested by its disappearance from the completion buffer, or instruction sequencing table (IST), in cycle 5. As mentioned earlier, all instructions are dispatched in order, as evidenced by the appearance of instruction IDs in the completion buffer; also, the instructions are completed in order, as evidenced by the deletion of IDs from the head of the completion buffer. By monitoring the last stage of the FLT execution pipe, we can see that the pipe reaches steady state in about 20 cycles, beyond which one fma is produced every three cycles, implying a steady-state performance of 3/2 = 1.5 cycles per flop (floating-point operation), since an fma counts as two flops.

Statistics for timing analysis and verification

The following metrics are used in our architectural timing verification and test methodology:

$L(n)$: The length, in number of instructions, of a loop trace, obtained by executing the source loop over n iterations.

Table 1 Daxpy test case measurements: ERISC machine.

Iteration count, n	$L(n)$	$T(n)$	$CPI(n)$	$CPL(n)$	$CPF(n)$	$CPA(n)$	$CPB(n)$	$CPX(n)$
1	5	10	2.000	10.000	5.000	1.667	10.000	3.333
2	10	13	1.300	6.500	3.250	1.625	6.500	2.167
3	15	17	1.133	5.667	2.833	1.416	5.667	1.889
4	20	20	1.000	5.000	2.500	1.333	5.000	1.667
5	25	23	0.920	4.600	2.300	1.278	4.600	1.533
6	30	26	0.867	4.333	2.167	1.238	4.333	1.444
7	35	29	0.829	4.143	2.071	1.208	4.143	1.381
8	40	32	0.711	4.000	2.000	1.185	4.000	1.333
9	45	35	0.778	3.889	1.944	1.167	3.889	1.296
10	50	38	0.760	3.800	1.900	1.152	3.880	1.267

$T(n)$: The execution time, in cycles, of a given loop test kernel, traced for n iterations.

$CPI(n)$: The average number of executed cycles per instruction, on processing n loop iterations; thus, $CPI(n) = T(n)/L(n)$. If the cycle time (or clock frequency) is known, CPI can easily be translated to the actual MIPS performance of the machine.

IPC: The inverse of CPI; i.e., $IPC = 1/CPI$; sometimes IPC is more convenient to use, since *increase* in IPC correlates directly with *increase* in processor performance. (See Figures 5 and 6, shown later, and the related discussion in Section 5.)

$CPL(n)$: The average number of executed cycles per loop iteration; i.e., $CPL(n) = T(n)/n$.

P_{ss}: The steady-state period, in cycles, of the timer pipeline state-transition pattern.

N_{ss}: The cycle count which marks the onset of a steady-state cycle-by-cycle timer output pattern, for a given test loop trace; such onset is defined to begin at the end of an iteration completion.

CPI_{ss}: The steady-state CPI for a given loop trace, defined as

$$CPI_{ss} = \lim_{n\to\infty} [T(n)/L(n)].$$

CPF: The average number of executed cycles per floating-point operation; it is obtained by dividing the total number of cycles by the total number of floating-point operations (flops) in the trace. Each compound floating op [e.g., the fma (floating multiply–add) op] counts as two flops. CPF is related to the MFLOPS performance of the processor.

FPC: The inverse of CPF; i.e., $FPC = 1/CPF$.

CPF_{ss}: The steady-state CPF for a given loop trace (containing floating ops), defined as

$$CPF_{ss} = \lim_{n\to\infty} [T(n)/(n \cdot FPL)],$$

where FPL is the number of flops per loop iteration.

CPX: The average number of executed cycles per fixed-point operation.

CPB: The average number of executed cycles per encountered branch operation.

CPA: The average number of processor cycles per memory access (load/store). This metric measures the processor-memory (cache) traffic for a given workload.

Steady-state CPL, CPX, CPB, and CPA can be defined in a manner similar to the other steady-state metrics.

With reference to the daxpy test case example (Figure 3), we can measure the loop and performance metrics as shown in **Table 1**.

Since the number of instructions per loop iteration $IPL = 5$, clearly, $L(n) = IPL \cdot n = 5 \cdot n$. Iteration completion cycles are 10, 13, 17, 20, 23, $\cdots$, i.e., values of $T(n)$. These are obtained by noting the cycles at which the loop-ending branch instructions (i.e., E, J, O, T, Y, $\cdots$, etc.) *complete*, i.e., disappear from the IST. From the $T(n)$ column, it is clear that the onset of the steady-state period occurs after $T(n) = 20$, since after that $T(n)$ always increments uniformly by three cycles. Thus, $N_{ss} = 20$ and $P_{ss} = 3$. The calculation of the performance metrics follows from their definitions. Note that the number of floating-point operations per loop iteration, FPL, is 2 (since an fma counts as two flops); the number of accesses per loop iteration, APL, is 3 (two loads and one store); the number of fixed-point operations per loop iteration, XPL, is also 3 (loads/stores count as FXU operations); the number of branches per loop iteration, BPL, is 1.

The lemma below follows easily from the previous definitions and discussion.

Lemma 2.1 The following identities must hold:

1. $P_{ss} = CPL_{ss}$.
2. $CPI_{ss} = CPL_{ss}/IPL$.
3. $CPF_{ss} = CPL_{ss}/FPL$.
4. $T(n) = N_{ss} + (n - N_{ss}) \cdot P_{ss}$, for $n \geq N_{ss}$.
5. $CPL(n) = CPB(n)$, assuming simple loop test cases (i.e., with single loop-terminating branch).

Note that this is only a partial (example) list of valid relations; others may be stated in terms of the other metrics defined. Also, note that although $APL = XPL$ for floating-point loop test cases, $CPA(n)$ is always less than $CPX(n)$, because the load/store cache *access* process runs ahead of the actual load/store *completion* process. For the daxpy test case example, $CPI_{ss} = 0.6$, $CPL_{ss} = 3.0$, and $CPF_{ss} = 1.5$.

3. Model defects and their effects

A formal functional performance fault (defect) model for the ERISC class of timers was proposed in [18]. In this paper, we illustrate the effect of specific model defects on timer output, pointing the way to the test case application and validation procedure. We demonstrate the effects in the context of our running example of the daxpy test case.

- *Timeline states and state transition*

Each dash in the timeline output (Figure 3) represents either a pipeline stage or a queue/buffer entry. The FLT unit pipeline flow, for example, is depicted by the flow of the fma instructions (B, G, L, Q, $\cdots$, etc.). The IST unit depicts a circular buffer, with instructions added to the tail in a circular fashion, with completed instructions disappearing from the head. The other queues depict FIFO transitions, with the maximum queue shifts determined by various dispatch and finish/complete *bandwidths*. For example, the maximum number of new instruction identifiers (IIDs) issued into the instruction sequencing table (IST) is 3, because the maximum instruction-issue bandwidth for the shown ERISC run is 3. For a given cycle n, the pattern of instructions populating each of the eight units shown defines a *unit-level state* S_U and a *global state* S_G. Under fault-free conditions, the intercycle state transitions follow a set of deterministic *rules*, which in essence define the pipeline flow execution semantics as stipulated in the processor organization specification manual. Timer model defects manifest themselves as deviations from (or violations of) these state-transition rules. As an example, consider the instruction-issue logic of the fault-free ERISC machine. The formal specification [18] can be distilled into the following simple "English" rules:

1. On a given cycle, the maximum number of instructions *dispatchable* from the PIB is $DISP = \min(nPIB, bIST, ibw)$, where $nPIB$ is the number of consecutive instructions counting from the head of the PIB (position 0, PIB[0]), up to the first dispatchable branch op; $bIST$ is the number of blanks (i.e., available empty slots) in the IST, and ibw is the maximum instruction issue bandwidth.
2. On a given cycle, any unit-specific instruction within the block of dispatchable instructions is blocked from dispatch if the corresponding unit instruction queue (e.g., LSTQ or FPU_IQ in Figure 3) is full.
3. If a given instruction is blocked from dispatch, all following instructions within the block of dispatchable instructions are also blocked.

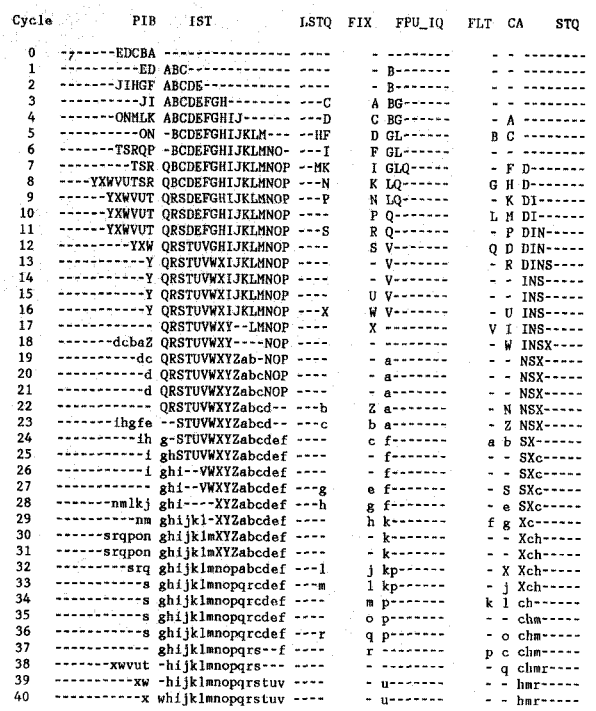
Similarly, rules defining instruction fetch (from cache) can be defined in terms of fetch bandwidth (fbw), slot availability in the PIB, and branch resolution time. For example, from Figure 3, note that instructions for the next iteration can be fetched into the PIB during the cycle *immediately after* the current iteration-ending branch has been dispatched. Rules for IST transition can be formulated in terms of completion bandwidth and the number and type of instructions marked "finished" on a given cycle. Pipeline stage transitions for the function units (FIX or FLT) are rather straightforward, with pipeline hazard conditions defining exceptions or disruptions to simple pipeline flow [20]. The full rule specification of the ERISC (timer) timeline state transition model is available in a more detailed technical report [21], which appears concurrently with the submission of this (revised) manuscript.

- *Rule-based state transition checking*

The following is a partial list of example assertions or rules, to check out the PIB unit-level state transitions (under infinite cache operation, loop test case input):

1. If [live_PIB(n) $\geq ibw$] and (queues_not_full), then [live_PIB($n + 1$) < live_PIB(n)].
2. If (ist_full) and (no_branches_scanned), then [live_PIB($n + 1$) − live_PIB(n)] $\leq fbw$.
3. If [PIB(n)[0].typ = brn_typ], then live_PIB(n) = 1.
4. If pib_full and dispatch_block, then state_pib(n) = state_pib($n + 1$); fetch_block = true.
5. If (fetch_block), then live_PIB($n + 1$) $\leq$ live_PIB(n).
6. If [for n = 1 to hang_limit, state_pib(n) = state_pib($n + 1$)], then timer_hang = true.

In the above notation, live_PIB(n) stands for the number of live instructions in the PIB at cycle n. The other predicates are self-explanatory. The first rule, for example, asserts that the number of live instructions in the PIB must decrease on state transition if dispatch conditions are enabled. As part of the overall validation methodology, we have implemented a robust state-transition checker program [21].



Cycle	PIB	IST	LSTQ	FIX	FPU_IQ	FLT	CA	STQ
0	-y-----EDCBA	----------	----	-	--------	-	-	--------
1	----------ED	ABC----------	----	-	B-------	-	-	--------
2	-------JIHGF	ABCDE-------	----	-	B-------	-	-	--------
3	----------JI	ABCDEFGH--------	---C	A	BG------	-	-	--------
4	-------ONMLK	ABCDEFGHIJ------	---D	C	BG------	-	A	--------
5	----------ON	-BCDEFGHIJKLM---	--HF	D	GL------	B	C	--------
6	-------TSRQP	-BCDEFGHIJKLMNO-	---I	F	GL------	-	-	--------
7	---------TSR	QBCDEFGHIJKLMNOP	--MK	I	GLQ-----	-	F	D-------
8	----YXWVUTSR	QBCDEFGHIJKLMNOP	---N	K	LQ------	G	H	D-------
9	------YXWVUT	QRSDEFGHIJKLMNOP	---P	N	LQ------	-	K	DI------
10	------YXWVUT	QRSDEFGHIJKLMNOP	----	P	Q-------	L	M	DI------
11	------YXWVUT	QRSDEFGHIJKLMNOP	---S	R	Q-------	-	P	DIN-----
12	---------YXW	QRSTUVGHIJKLMNOP	----	S	V-------	Q	D	DIN-----
13	----------Y	QRSTUVWXIJKLMNOP	----	-	V-------	-	R	DINS----
14	----------Y	QRSTUVWXIJKLMNOP	----	-	V-------	-	-	INS-----
15	----------Y	QRSTUVWXIJKLMNOP	----	U	V-------	-	-	INS-----
16	----------Y	QRSTUVWXIJKLMNOP	---X	W	V-------	-	U	INS-----
17	----------	QRSTUVWXY--LMNOP	----	X	--------	V	I	INS-----
18	-------dcbaZ	QRSTUVWXY----NOP	----	-	--------	-	W	INSX----
19	----------dc	QRSTUVWXYZab-NOP	----	-	a-------	-	-	NSX-----
20	----------d	QRSTUVWXYZabcNOP	----	-	a-------	-	-	NSX-----
21	----------d	QRSTUVWXYZabcNOP	----	-	a-------	-	-	NSX-----
22	----------	QRSTUVWXYZabcd--	---b	Z	a-------	-	N	NSX-----
23	-------ihgfe	--STUVWXYZabcd--	---c	b	a-------	-	Z	NSX-----
24	----------ih	g-STUVWXYZabcdef	----	c	f-------	a	b	SX------
25	----------i	ghSTUVWXYZabcdef	----	-	f-------	-	-	SXc-----
26	----------i	ghi--VWXYZabcdef	----	-	f-------	-	-	SXc-----
27	----------	ghi--VWXYZabcdef	---g	e	f-------	-	S	SXc-----
28	-------nmlkj	ghi----XYZabcdef	---h	g	f-------	-	e	SXc-----
29	----------nm	ghijkl-XYZabcdef	----	h	k-------	f	g	Xc------
30	------srqpon	ghijklmXYZabcdef	----	-	k-------	-	-	Xch-----
31	------srqpon	ghijklmXYZabcdef	----	-	k-------	-	-	Xch-----
32	----------srq	ghijklmnopabcdef	---l	j	kp------	-	X	Xch-----
33	----------s	ghijklmnopqrcdef	---m	l	kp------	-	j	Xch-----
34	----------s	ghijklmnopqrcdef	----	m	p-------	k	l	ch------
35	----------s	ghijklmnopqrcdef	----	o	p-------	-	-	chm-----
36	----------s	ghijklmnopqrcdef	---r	q	p-------	-	o	chm-----
37	----------	ghijklmnopqrs--f	----	r	--------	p	c	chm-----
38	-------xwvut	-hijklmnopqrs---	----	-	--------	-	q	chmr----
39	----------xw	-hijklmnopqrstuv	----	-	u-------	-	-	hmr-----
40	----------x	whijklmnopqrstuv	----	-	u-------	-	-	hmr-----

Figure 4

ERISC timer output (daxpy) with injected model defect.

An example timer model defect, injected by reducing the FIX and FLT pipe latencies and also removing a predicate from the instruction issue logic, causes the daxpy timeline output to change, as shown in **Figure 4.** Note that the steady-state period P_{ss} changes from 3 to 5 in the new timeline; also, N_{ss} is seen to increase significantly. Thus, an indirect way of inferring a subset of possible rule violations is to measure metric deviations from "expected" or "gold" values. This is the subject of the following subsection.

- *Loop-driven behavioral defect analysis*

The basic procedure described in this section is to drive the timer model to steady-state periodic behavior using simple loop test cases (like daxpy). Model defects are diagnosed from the behavior of the output, as manifested by metric measurement and characterization. The test cases are in the form of simple high-level FORTRAN, C, or assembler loops specifically designed to test for model defects (see Section 5). Both synthetic and application-based instruction test sequences are used. A given loop test case is traced over an increasing number of iterations and is fed to the timer to (progressively) drive it to a steady-state pattern.

Steady-state period analysis

By monitoring the steady-state pattern and the corresponding periodicity at the pipeline stages shown on the timer-generated timeline output, the various (steady-state) performance metrics (*CPI*, *CPF*, *CPA*, etc.) can be calculated easily. We explain this procedure with reference to the example timeline output shown in Figure 3.

It is to be noted that each unit or queue individually attains a steady-state pattern, with the same fundamental period of three cycles. Thus, beyond cycle $N_{ss} = 20$, the PIB exhibits a recurring pattern of 5–2–1, i.e., five instructions, followed by two, followed by one. This phenomenon of attainment of a uniform steady-state pattern across the modeled pipelined units when driven by an iterative loop trace is the characteristic signature of a *level-0* validated timer (see the definition in the following subsection).

In general, the steady-state pattern of an execution pipeline, such as the FLT pipeline, may not exhibit a *simple* periodic behavior of one fma produced every p cycles (implying a CPF_{ss} of $p/2$). For example, depending on the organizational parameter settings, the following steady-state pattern may emerge:

——— (fma finishes) · · · cycle n
—— (idle) · · · cycle $n + 1$
—— (idle) · · · cycle $n + 2$
—— (idle) ·
—— (idle) ·
——— (fma finishes) ·
—— (idle) ·
—— (idle) ·
——— (fma finishes) · · · cycle $n + 8$

The periodicity of the gap (idle) sequence in this case is 4–2, i.e., four gaps followed by two gaps. The gap sequence length (GSL) is defined to be the total number of distinct gap subpatterns within a period. Thus, for the example above, $GSL = 2$; (period is composed of two gap subpatterns: a four-gap subpattern and a two-gap subpattern). The steady-state CPF for the above case can be expressed as

$$CPF_{ss} = (4 + 2 + GSL)/(2GSL) = (4 + 2 + 2)/4 = 2.0.$$

Generalizing to the case for which the steady-state gap sequence exhibits the pattern $g(1)$–$g(2)$– · · · $g(m)$, we would have $GSL = m$, and

$$CPF_{ss} = \left[\sum_i g(i) + m\right] \Big/ 2m.$$

The above formulation effectively describes an algorithm to evaluate or infer a steady-state metric such as CPF_{ss} from a generated timeline output.

Levels of timer model validation

In the timer model evolution cycle, after the initiation of a CPU development project, the modeler and user typically go through distinct phases, characterized by increasing levels of confidence in the accuracy of the model. In the light of such experience, we have followed a multilevel model validation methodology. This approach has allowed us to plan the timer development schedule with meaningful status checkpoints and has enabled us to decide on the appropriate point at which the model can be used reliably for performing crucial design trade-off analysis.

We define three distinct levels of timer model validation, as follows.

Level 0 In this level of validation, all test loop traces applied result in a uniform, steady-state timer output pattern of *finite periodicity,* P_{ss}, attained within a *finite number of cycles,* N_{ss}. If the observed steady-state period is infinite (i.e., if the timer output pattern stabilizes to the exact same overall pipeline state for any cycle count greater than N_{ss}), then clearly the timer is in an illegal, deadlocked state; in such a case, the model is said to *fail* level-0 validation for the particular test loop trace.

The practical implication of attaining level-0 validation, as above, is as follows: The primary model defects encountered by the modeler during the initial testing stage are those which manifest themselves as timer "hang" problems. In effect, the cause behind such level-0 defects is logical errors in implementing the state-transition logic of pipelined instruction flow through the timer model. In rare cases, an actual design error in the machine logic specification is discovered in fixing such a level-0 validation problem.

The timer model is said to be weakly level-0-validated if it passes level-0 validation in the above-defined sense, across the test loop trace suite, for the specified design point characterized by an exact setting of organizational parameters. The model is said to be strongly level-0-validated if it passes level-0 validation for all legal (allowed or defined) combinations of organizational parameters.

Level 1 In this level of validation, for each test loop trace, the performance metrics (*CPI*, etc.) are observed to vary *monotonically* as a function of any given organizational parameter size (within its defined limits). (Only one parameter is varied at a time.) Also, for each loop test case, the *CPI* is expected to *decrease monotonically* toward a limiting *asymptote*, CPI_{ss}, as the number of loop iterations traced is increased; this behavior should be exhibited for every legal combination of organization parameters.

This phase of timer behavioral testing can be characterized by two different modes: transient-mode testing and steady-state parametric mode testing. At the end of level-0 validation, the modeler is able to project performance projection numbers for large benchmarks, such as the SPEC™ 92 suite, without encountering timer "hang" problems. However, the accuracy of the projected numbers is unknown. The level-1 validation procedure stresses the timer model by playing out the entire legal range of model parameters and observing the output metrics for behavioral consistency. (See the additional discussion of modes of Level-1 validation, later in this section.) Passing level-1 validation tests does not guarantee cycle-by-cycle or even cycle-count accuracy; however, at this stage, all or most of the logical errors in the individual units and inter-unit "glue" codes are typically eliminated. Beyond this stage, it is usually only a matter of validating the correct settings of the dozens of (often interacting) model parameters to ensure that the exact processor design point has been achieved. This is the objective of the next level of validation.

Level 2 In this level of validation, each test loop trace is verified to produce steady-state performance metrics (e.g., CPI_{ss} or CPF_{ss}) which are in agreement with deterministic (infinite-cache) static prediction formulae.

Level-2 validation exercises check for exact cycle-count accuracy across the test case repository. In essence, by comparing the measured metrics against analytic, bounds, and bandwidth-based formulae[1] [10–12, 22–25], the model parameters can be tuned to achieve the desired match. A typical example of a model bug caught in this stage is as follows: Suppose that the level-2 cache-hit latency parameter is intended to be seven cycles; hence, the corresponding parameter has been set to 7. However, in terms of the model internals, this setting actually causes an effective latency of (say) eight cycles as far as the actual designer's counting convention is concerned. This kind of error, caused typically by designer–modeler miscommunication, is detected and fixed during level-2 validation.

We state the following lemma without formal proof. It follows quite clearly from the definitions of Level-0/1/2 validation, above.

Lemma 3.1 Passing level-1 *and* level-2 validation for the generated loop-trace test case suite guarantees *strong* level-0 validation with respect to that test suite (but not vice versa), under the infinite cache execution semantics assumed for ERISC.

In addition to the above, in our verification methodology, we define a *level-3* validation step, in which

[1] J.-D. Wellman, "Cycles-per-Instruction Estimation Methods," IBM internal project report, September 1992, revised November 1993.

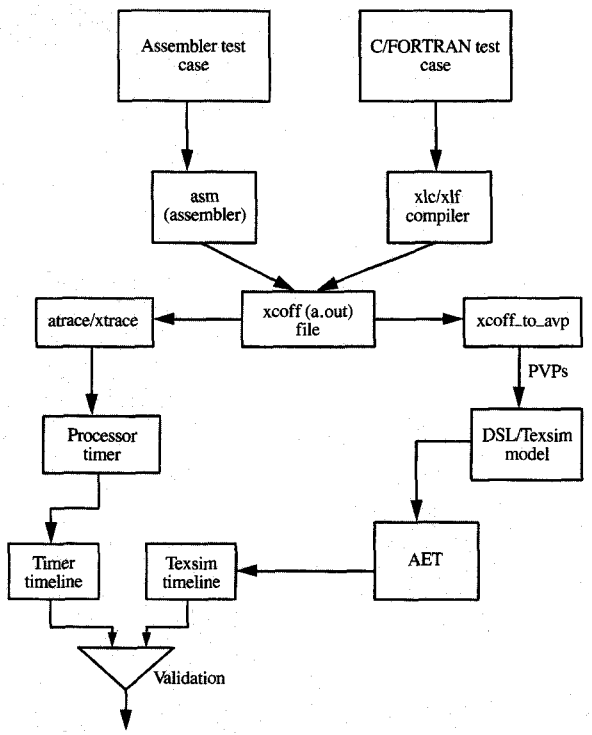


Figure 5

Level-3 validation procedure; PVP generation and use.

the cycle-by-cycle timer outputs are validated against the full-scale logic simulation (DSL/ Texsim) model of the processor under design. This level of validation is beyond the scope of this paper, and is therefore not discussed in detail here. The logical block diagram (flowchart) illustrating the level-3 validation procedure is shown in **Figure 5.** With minor variations, this basic methodology is used for timer–Texsim cross-validation by processor design and simulation groups within other IBM facilities, as well as for the PowerPC family of processors. A higher-level test case, usually an iterative loop structure, is coded either in assembler or in a high-level language (e.g., FORTRAN or C). The test case is translated into an xcoff (a.out) file and then traced via Atrace[4] [21] to feed the processor timer. The Xcoff-to-AVP utility[5] is used to generate an AVP, which, because of the intent of use, we may call a PVP (performance verification program).

[4] "Atrace," IBM internal software, 1991; author: R. Nair, IBM Thomas J. Watson Research Center, Yorktown Heights, NY.

[5] "Xcoff-to-AVP," IBM internal software, 1993; author: S. Hoxey, IBM Toronto, on assignment at Somerset Design Center, Austin, TX.

The PVP is equipped with the requisite setup and register/memory initialization code, and the proper instruction/data-card syntax, etc., so that the test program can be executed on the Texsim simulation model. The "all-events-trace" (AET) dump file generated by the Texsim simulation run is visualized in a timer-like timeline (see Figure 3), using another utility called Arch-I-Tex.[6] The timer-generated and Texsim-generated timeline outputs are compared for cross-validation. At the simplest level, cycle counts are compared; at the most complex level, cycle-by-cycle matching of pipeline states in the two timelines may be compared. There are intermediate levels of matching possible within this level-3 validation procedure. Detailed discussion of level-3 validation is deferred to a subsequent publication.

• *Modes of level-1 validation*

The following experimental level-1 validation modes, as mentioned before, are currently in use.

Transient CPI/CPF mode

In this mode, the cumulative CPI is measured and plotted against increasing iteration count for a given loop test case. The iteration count is varied in unit steps until the *CPI/CPF* metrics reach steady state. If the metrics do not approach an asymptotic steady state, a timer fault is flagged. A permanent timer hang situation, for instance, will cause *CPI* and *CPF* to increase without limit as the number of iterations is increased. Also, a nonmonotonic decrease of *CPI* signals faulty behavior. The following lemma is stated for use in the transient-mode timer model testing methodology.

Lemma 3.2 Under transient validation mode (as above), a necessary condition for deducing fault-free timer model behavior is that *CPI* and *CPF* measurements are observed to decrease, *monotonically* and in a nonlinear curve, toward an asymptote as the number of loop iterations traced and processed is increased.

Steady-state, parametric mode

In this mode, the steady-state *CPI* and *CPF* are measured for increasing or decreasing values of selected organizational parameters. Both single and multiple parametric testing may be performed, where "single" testing implies varying a single parameter at a time, and "multiple" means varying two or more parameters simultaneously in the same direction. The following lemmas are stated for use in the parametric-mode timer model testing methodology.

[6] "Arch-I-Tex," IBM internal program utility, 1993; author: R. Wasmuth, IBM Austin.

Lemma 3.3 Under fault-free conditions, the CPI_{ss} and CPF_{ss} for a given loop test trace should be monotonically nonincreasing as a given single queue-size, buffer-size, or bandwidth parameter is increased.

Lemma 3.4 Under fault-free conditions, the CPI_{ss} and CPF_{ss} for a given loop test trace should be monotonically nondecreasing as a given functional latency parameter (e.g., the number of stages in a pipeline) is increased.

Lemma 3.5 Under fault-free conditions, the period of onset of steady-state pattern N_{ss} should be monotonically nondecreasing (nonincreasing) as a given queue or buffer size parameter is increased (decreased).

Static loop execution time prediction

On the basis of prior work on static execution time estimation for loop structures[1] [10–12, 24, 25], we have formulated an exact algorithm for predicting the infinite-cache, steady-state period, and hence CPI_{ss}, as well as the cycle of onset N_{ss} of the steady-state pattern, in terms of the organizational parameters stated earlier (Section 2). We omit the discussion of that methodology in this paper. We give a simplified example of the kind of end result one can obtain using the theory. Consider the daxpy test case example, discussed earlier. The following "rules of thumb" may be used to roughly summarize the theoretical bounding methods for daxpy-like floating-point loops:

1. The loop-ending branch is fully overlapped with computation and takes "zero cycles" in the steady-state sense.
2. Let N_L be the number of loads needed per iteration. This is the number of elements newly accessed on a given loop iteration.
3. Let N_S be the number of stores needed per iteration. This is the number of target elements (to the left of an assignment) newly referenced on this iteration of the loop.
4. Let N_F be the number of functional arithmetic instructions (other than divides) needed for the computation.
5. Let N_D be the number of divides.

It should be remembered [22, 23] that for this RS/6000-like ERISC machine,

1. A store takes one cycle (pipelined) and cannot be overlapped with loads or fmas (floating-point multiply–add instructions).
2. A load takes one cycle (pipelined) but can be overlapped with fmas.
3. An fma costs one cycle if it is independent of the previous fma and two cycles if it is dependent.
4. A divide is assumed to take D cycles (nonpipelined); D = 16 to 19 cycles for the RS/6000.

The minimum number of execution cycles per iteration [22, 23], assuming independent fmas, perfect instruction overlap, no divides, and a terminating zero-cost branch, is then

$$T_{min} = N_S + \max(N_L + N_F).$$

If there are divides in the loop, N_F must be replaced by $N_F + D \cdot N_D$ in the above equation.

By substituting $N_S = 1$, $N_L = 2$, and $N_F = 1$ for the daxpy loop case, we obtain an analytically predicted steady-state cycles-per-iteration value of (1 + 2) = 3, which gives us a CPF of 1.5, identical to the CPF_{ss} obtained from the timer run (Figure 3).

4. Test case selection, generation, and application

Test cases used in our methodology are of two broad categories: a) real-application-based test kernels and b) specially designed synthetic test sequences. The first category of test cases is useful because of known techniques for evaluating application kernel performance, based on bounds and bandwidth analysis. With known target performance, timer-generated metrics can be directly checked for accuracy. The synthetic test cases are designed to validate core parameter settings in the timer model. **Figure 6** shows the classification and interrelationships of the two classes of test cases, and the source applications from which the type-a test cases are derived.

The synthetic test case generation methodology takes as input the basic organizational parameters of interest (Section 2) and optional "bias" parameters, to focus attention on a limited aspect of the defined performance (timing) fault model [18]. Once a complete suite of test loops is generated to cover all modeled faults across instruction classes, the level-x validation methodology (Section 3) is applied (x = 0, 1, 2). A few simple examples are given below.

- *Examples of test case generation*

Example 1: Consider the case of testing for correct behavior of floating-point load/store instructions. The high-level code segment used for generating compiled test sequences is clearly one or more instances of the assignment operation (i.e., $A = B$). To apply this repeatedly for a large number of consecutive load/store sequences, in order to force an asymptotic, steady-state CPI and hence a characteristic (repeating) timeline signature, the required test loop (in FORTRAN source) is as follows.

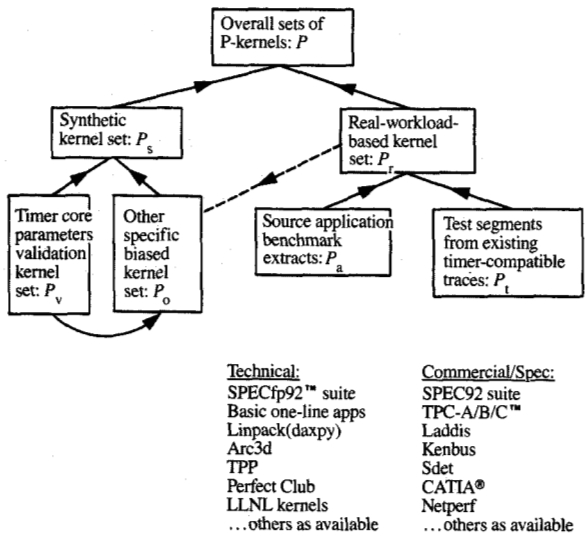


Figure 6

Performance test kernel selection.

Test kernel for load/store testing: The load/store test case

do $i = 1, n$
  $a(i) = b(i)$
enddo

The corresponding machine loop code sequence generated by the compiler is

```
lfdu   0, 5, 0×8
stfdu  0, 4, 0×8
bc
```

The types of functional timing faults [18] detected by this test case are a) dispatch decode, execution, or sequence fault: a decode fault causes the observed CPI_{ss} to be less than expected value; an execution fault causes the observed CPI_{ss} to be larger than expected; a sequence fault is detected via disagreement between total number of instructions dispatched and the actual trace length L; b) FIX execution or sequence fault: an execution fault in the finish-signal dispatch path causes CPI_{ss} to increase (possibly without limit, causing level-0 failure of validation); a sequence fault also causes a similar effect.

The fault-free CPI_{ss} for this loop trace is clearly seen to be 2/3 = 0.667 (see discussion on static loop execution prediction, Section 4).

Example 2: Consider the case of testing the basic mechanism of decoupled access–execute execution of superscalar processors such as ERISC. The fixed-point pipeline processes the floating-point loads and stores for computation performed by the FLT pipeline. The simplest test case for this purpose is a repetitive sequence of floating-point addition operations, generated by the following loop.

Test kernel for testing overlapped (decoupled) access–execute: The addition test case

do $i = 1, n$
  $c(i) = a(i) + b(i)$
enddo

The corresponding machine code sequence is

```
lfdu   0, 6, 0×8
lfdu   1, 5, 0×8
fa     0, 0, 1, 0, 0
stfdu  0, 4, 0×8
bc
```

Floating-point registers 0 and 1 are loaded by successive instructions prior to their use as operands by the floating-point add. There is a true dependency, which on first sight may appear to cause a dependence bubble in the FLT pipe. However, in pipelined mode, and because of the presence of register renaming, the throughput of completed adds should be determined solely by the three load/store instructions. Since we are dealing with a single-ported cache, the number of steady-state cycles per iteration should be 3; hence, CPF_{ss} should also be 3.0. Unrolling the loop will not further improve the CPF because of the limitation imposed by the single cache port.

Example 3: Consider the case of testing for peak dispatch, execute, and completion rate of three instructions per cycle, resulting in $CPI_{ss} = 0.333$. We could construct this case by using a sequence of lfd, fma, and bc, and iterating over the loop. The source loop to generate this would be as follows.

Test kernel for peak performance: The reduction test case

do $i = 1, n$
  $t = t + b \cdot a(i)$
enddo

The machine loop code sequence is

```
lfdu   1, 4, 0×8
fma    2, 0, 2, 1, 0
bc
```

The scalars t and b are loaded once outside the loop, causing the iteration loop to have the lfd, fma, bc sequence. Under adequate settings for rename buffers and queue sizes, this loop trace is expected to generate a CPI_{ss}

of 0.333 and a CPF_{ss} of 0.5. Deviation upward would generally point to execution or decode faults; downward deviation is not possible under modeled failures for this particular test case.

Example 4: The following loop is the key loop within the *linpack* scientific benchmark. Like the test case in Experiment 2, this is also a load/store-bound loop, but with two floating-point operations (because of the multiply–add) per cycle. This kernel is useful for testing access–execute overlap (such as Example 2). In addition, the basic limitation imposed by the load/store accesses (for the single-port cache implementation used in ERISC) can be tested, by parametric testing, with and without loop unrolling.

Test kernel for testing store-bound scientific computation: The daxpy test case

do $i = 1, n$
  $x(i) = x(i) + s \cdot y(i)$
enddo

The corresponding machine code sequence is

```
lfd    1, 6, 0×8
fma    1, 0, 2, 1
lfdu   2, 5, 0×8
stfdu  1, 6, 0×8
bc
```

As mentioned previously, this test case is expected to generate a CPF_{ss} of 1.5 for the ERISC machine. Deviation upward to 2.0 suggests a fault in the instruction completion logic (ICL), where the number of instructions completed per cycle may be erroneous, due either to a local (BRN) logic fault or to an execution fault (finish signal dispatch error) in FIX or FLT. Increases in *CPF* may also be caused by faults in the load/store priority logic in accessing the single-port cache. (This latter logic has not been specified in our earlier discussion.)

Single-instruction-cycle_count validation test cases
Synthetic-loop test cases, specially set up to measure and validate individual instruction timings, are easy to generate. Typically, a sequence of identical opcodes (with independent as well as dependent operands) is fed to the timer. The lifetime of each instruction is inferred from the throughput rate and the N_{ss} and P_{ss} metrics.

• *Experimental results with SPEC92 workload: ERISC timer*
Systematic loop test case generation coupled with level-x validation/verification enables the designer to detect modeled timing faults, which are otherwise hard to detect and diagnose from large benchmark run results alone. The effect of making the model progressively robust through level-x validation is seen in practice through changes in experimentally observed SPEC92 benchmark suite *CPI* numbers for ERISC, as we progressed from level 0 to level 2. **Table 2** shows the observed changes in *CPI*, over the course of verifying the machine timings, as embodied in the ERISC timer. In most cases, level-1 validation alone did not detect all the problems in the level-0-validated model. The overall difference in figures between the level-0 numbers and the level-2 numbers is quite significant, and it underlines the importance of performing systematic performance model verification/validation tests.

Table 2 Experimental ERISC timer validation data for SPECint92 benchmarks.

Benchmark	*CPI (level-0 model)*	*CPI (level-1 model)*	*CPI (level-2 model)*
compress	0.992	0.892	0.891
eqntott	0.801	0.763	0.773
espresso	1.035	1.031	0.998
gcc	1.243	1.241	1.141
li	0.989	0.989	1.119
sc	1.133	1.133	1.134

• *Transient and parametric mode experiments (level-1 validation): PowerPC 601 timer*
In this subsection we consider a timer model for a real superscalar machine: the PowerPC 601, and present experimental results illustrating the use of *transient* and *parametric* mode testing (see definitions in Section 3). The PowerPC 601 RISC microprocessor [26, 27] is the first implementation of the PowerPC architecture [2]. Pre-hardware performance modeling for most of the PowerPC microprocessor family (601, 603, 604, and 620) was performed using a simulation toolkit centered on a parameterized, dynamic (i.e., trace-driven) timer, called BRAT (Basic RISC Architecture Timer) [13, 16], developed and used by the PowerPC performance modeling team.

Figure 7 shows the transient-mode performance variations, as a function of the number of loop iterations, for each of the experimental test cases (Examples 1 through 4) described above. The current 601 timer model, with all parameters set and checked to match with the actual design, was used for the study. All of the graphs meet the condition stipulated by Lemma 3.2. In these graphs, the inverse of *CPI* and *CPF*, namely *IPC* and *FPC*, are plotted. In Figure 7(b), both the *IPC* and *FPC* variations are shown. **Figure 8** depicts the parametric (steady-state) *IPC*/*FPC* variations for two example parameters. In the first graph, the infinite-cache access latency (in processor cycles) is varied. The glitch in the *IPC* and *FPC* curves between latency 2 and 3 points to

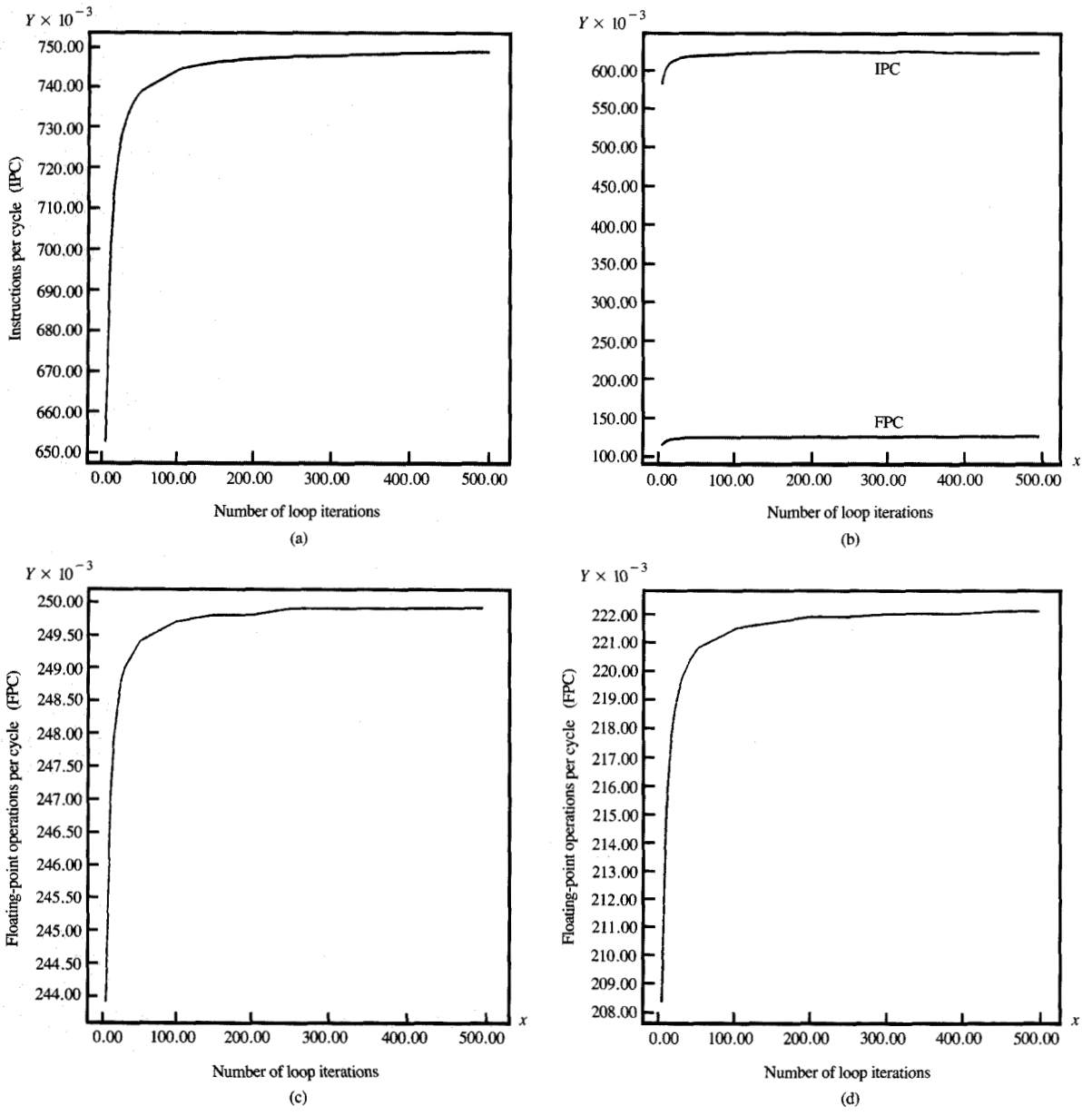


Figure 7

Transient-mode loop trace experiments: (a) Experiment 1—load store test case; (b) Experiment 2—addition test case; (c) Experiment 3—sum reduction test case; (d) Experiment 4—daxpy test case.

a timer model defect (see Lemma 3.4). This defect was logged and diagnosed to be real through code investigation. It has since been repaired, causing changes to some of the SPEC92 projections. The graph in Figure 8(b) shows the variation of *FPC* with the size of the instruction buffer (8 in the current design) for the daxpy test loop trace. Lemma 3.3 is satisfied, and the data indicate that for sizes of the instruction buffer above 3, there is no additional benefit. This is consistent with static (analytic) estimate equations [11], given the maximum number of instructions

which can be dispatched per cycle in this processor. Detailed experimental results and analysis for the 601 model are available in a technical report [28].

5. Conclusion

Architectural timing verification is rapidly becoming a critical part of the overall simulation and verification effort in advanced CMOS superscalar (pipelined) processor development projects. Timer models used in pre-hardware design trade-offs and performance projections require validation support to ensure accuracy. Execution cycle (performance) verification of the hardware description-language-based full logic simulation model is a closely related problem. A systematic methodology to address the above timing verification problems for current- and future-generation processor chips has been presented. This methodology is currently being used for testing, debugging, and validation of advanced superscalar timer models within the IBM Systems Technology and Architecture Division. The test loop trace-driven period analysis and related techniques may also be used to test the timing behavior of actual implemented hardware, or the full-scale logic simulation model, prior to final chip specification. In practice, we have found the described methodology to be very useful for augmenting the traditional architectural verification programs (AVPs) with performance verification programs (PVPs) to form a complete, robust verification suite. Our current timer validation methodology has helped us to project very accurate processor performance numbers, and to make accurate trade-off analyses, well before hardware implementation.

The goal of quantifying fault coverage in a manner analogous to hardware testing has not been directly addressed. In other words, test case generation followed by the standard validation experiments does not necessarily give us a measure of the percentage of faults covered. In ongoing research, we hope to address this issue in a more comprehensive manner. More theoretical work is needed to enable accurate prediction of performance deltas caused by model defects. Currently, we are building from existing theory on bounds-based estimation to quantify performance (e.g., *CPI*) deltas due to individual parametric variations. Completion of this theory will enable us to infer model defects from observed incorrect parametric-mode variations (see the subsection on loop-driven behavioral defect analysis).

The science (or art?) of timer model validation as reported here is acknowledged to be still in its infancy. However, the methods reported here have been used with success and have resulted in the removal of innumerable model defects. The goals of this paper were to present a real-life view of current timer model validation methods, as used in the current PowerPC 6XX development process; to publicize the critical need for robust performance verification strategies; and (we hope) to attract other performance analysts, modelers, and architects who will make future contributions in this area.

Some of the ideas on state-transition testing and validation of timer models mentioned in this paper have

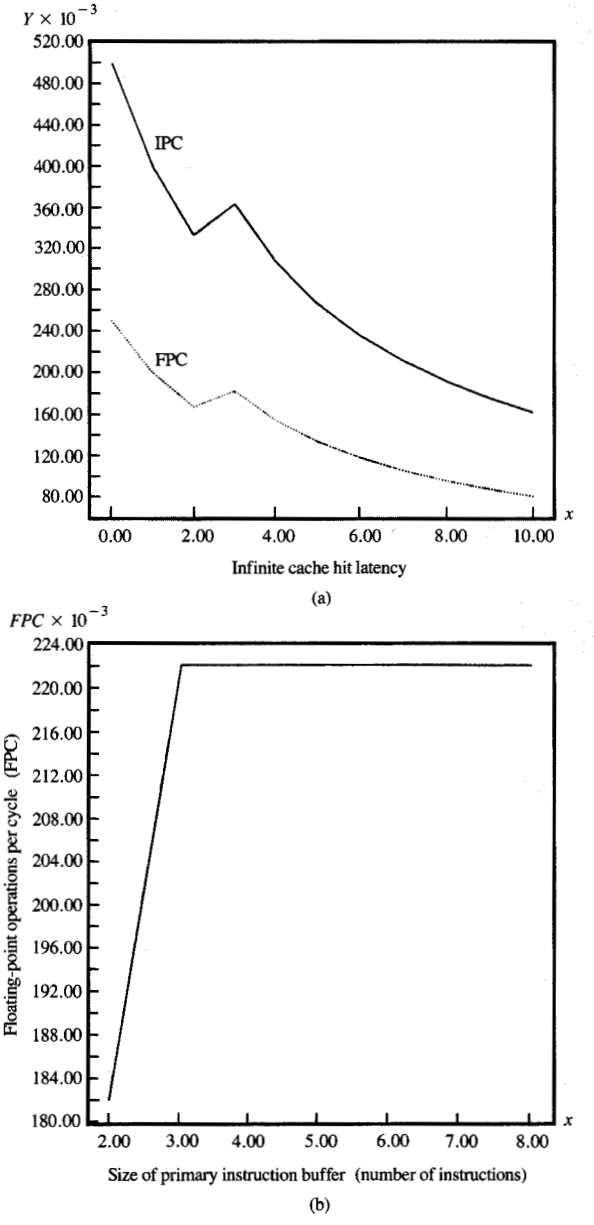


Figure 8

Parametric, steady-state-mode loop trace experiments: (a) Experiment 3—sum reduction test case; (b) Experiment 4—daxpy test case.

extended themselves into another application: design of self-checking architectural timers. A report on this topic, describing the methodology and toolset, is being planned for separate publication.

Acknowledgment

The authors are indebted to Danny Shieh and Lawrence Hannon for many fruitful discussions on floating-point loop timing assessment and measurement. They are also grateful to Mike Peters and Ali Poursepanj of the Somerset Design Center, Austin, for providing access to the BRAT 601 timer, which was used to try out some of the validation experiments reported in Section 5. One of the authors (PB) would like to acknowledge the help received from his IBM colleagues Hung Le, Charles Barton, and Robert Wasmuth in the form of many illuminating discussions. Many of the level-1 validation ideas reported here have also been applied to test and verify a future custom timer (primary coder: Charles Barton), which is currently under use in modeling an advanced, semicustom CMOS RISC processor timer at IBM Austin; the ERISC timer, which was used to develop the initial research ideas behind the validation methodology, is a separate, example (nonproprietary) model developed by one of the authors (PB). Finally, the authors would like to thank the referees for their valuable comments.

References

1. R. R. Oehler and R. D. Groves, "IBM RISC System/6000 Processor Architecture," *IBM J. Res. Develop.* **34,** No. 1, 23–36 (January 1990).
2. E. Silha, "PowerPC Architecture: A High-Performance Architecture with a History," *PowerPC and POWER2: Technical Aspects of the New IBM RISC System/6000*, pp. 73–79, Order No. SA23-2737-00; available through IBM branch offices; E. Silha and G. Paap, "PowerPC: A Performance Architecture," *Proceedings of IEEE COMPCON*, February 1993, pp. 104–108.
3. T. Brodnax, M. Schiffli, and F. Watson, "The PowerPC 601 Design Methodology," *Proceedings of the IEEE International Conference on Computer Design (ICCD)*, October 1993, pp. 248–252.
4. S. M. Thatte and J. A. Abraham, "A Methodology for Functional Level Testing of Microprocessors," *Proceedings of the 8th International Conference on Fault-Tolerant Computing*, Toulouse, France, IEEE Computer Society, June 1978, pp. 90–95.
5. D. Brahme and J. A. Abraham, "Functional Testing of Microprocessors," *IEEE Trans. Computers* **C-33,** 475–485 (June 1984).
6. S. Tahar and R. Kumar, "Towards a Methodology for the Formal Hierarchical Verification of RISC Processors," *Proceedings of the IEEE International Conference on Computer Design (ICCD)*, October 1993, pp. 58–62.
7. W. Hunt, "The Mechanical Verification of a Microprocessor Design," *From HDL Description to Guaranteed Correct Circuit Designs*, D. Borrione, Ed., North-Holland Publishing Co., New York, 1987.
8. M. Srivas and M. Brickford, "Verification of a Pipelined Microprocessor Using Clio," *Hardware Specification, Verification and Synthesis: Mathematical Aspects*, M. Leeser and G. Brown, Eds., Springer, New York, 1990.
9. G. Radin, "The 801 Minicomputer," *Proceedings of the Symposium on Arch Support for Programming Languages and Operating Systems*, March 1982, pp. 39–47.
10. P. Bose, "Early Performance Estimation of Super Scalar Machine Models," *Proceedings of the IEEE International Conference on Computer Design (ICCD)*, October 1991, pp. 388–392.
11. P. Bose and J.-D. Wellman, "MIPS-Driven Design and Estimation of VLSI CPUs," *Proceedings of IEEE VLSI Design '93*, Bombay, January 1993.
12. P. Bose, "Time Attributed Dependence Graph Scheme for Prediction of Execution Time for a Block of Assignment Statements with Looping," *IBM Tech. Disclosure Bull.* **36,** No. 09A, 621–622 (September 1993).
13. A. Poursepanj, D. Ogden, B. Burgess, S. Gary, C. Dietz, D. Lee, S. Surya, and M. Peters, "The PowerPC 603 Microprocessor: Performance Analysis and Design Trade-offs," *Proceedings of IEEE COMPCON*, 1994, pp. 316–323.
14. G. F. Grohoski, "Machine Organization of the IBM RISC System/6000 Processor," *IBM J. Res. Develop.* **34,** No. 1, 37–58 (January 1990).
15. J. S. Liptay, "Design of the IBM Enterprise System/9000 High-End Processor," *IBM J. Res. Develop.* **36,** No. 4, 713–731 (July 1992).
16. Papers in session on the PowerPC 603 Microprocessor, Charles Moore, session chair, *Proceedings of IEEE COMPCON*, 1994, pp. 300–323.
17. P. Bose, "The Cheetah/ELES Timer Document," IBM internal document (now declassified, available on request), Thomas J. Watson Research Center, Yorktown Heights, NY, May 1985.
18. P. Bose, "Architectural Timing Verification and Test for Super Scalar Processors," *Proceedings of FTCS-24*, June 1994, pp. 256–265.
19. *AIX Operating System, C Language Reference*, March 1991, Order No. SC23-2058-02; *AIX XL Fortran Compiler/6000, Language Reference, ver. 2.3*, September 1992, Order No. SC09-1353-02; available through IBM branch offices.
20. P. M. Kogge, *The Architecture of Pipelined Computers*, McGraw-Hill Book Co., Inc., New York, 1981.
21. P. Bose and S. Surya, "Architectural Timing Verification of CMOS RISC Processors," Technical Report, IBM Systems Technology and Architecture Division, Austin, TX, 1995, to appear.
22. IBM, International Technical Support Centers, "Predicting Execution Time on the IBM RISC System/6000," Order No. GG24-3711, July 1991; available through IBM branch offices.
23. IBM, International Technical Support Centers, "IBM RISC System/6000 NIC Tuning Guide for Fortran and C," Order No. GG24-3611-01, July 1991; available through IBM branch offices.
24. W. Mangione-Smith, S. G. Abraham, and E. S. Davidson, "A Performance Comparison of the IBM RS/6000 and the

Astronautics ZS-1," *IEEE Computer* **24,** 39–46 (January 1991).
25. W. Mangione-Smith, T.-P. Shieh, S. G. Abraham, and E. S. Davidson, "Approaching a Machine Application-Bound in Delivered Performance on Scientific Code," *Proc. IEEE* **81,** 1166–1178 (August 1993).
26. M. C. Becker, M. S. Allen, C. R. Moore, J. S. Muhich, and D. P. Tuttle, "The PowerPC 601 Microprocessor," *IEEE Micro* **13,** 54–68 (October 1993).
27. IBM Microelectronics and Motorola, *PowerPC 601 RISC Microprocessor User's Manual*, copyright Motorola, Inc., 1993.
28. S. Surya, P. Bose, and J. A. Abraham, "Architectural Timing Model Validation: PowerPC Processor Family," *Technical Report TR 51.084-0*, IBM Systems Technology and Architecture Division, Austin, TX, 1994.

Received May 24, 1994; accepted for publication October 28, 1994

Pradip Bose *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (BOSE at YKTVMV).* Dr. Bose is a research staff member at the IBM Thomas J. Watson Research Center; when the work reported in this paper was done, he was on assignment at the IBM Systems Technology and Architecture Division, Austin, Texas, where he led the performance tools and analysis support effort for an advanced superscalar processor development project. Dr. Bose received his M.S. and Ph.D. degrees in electrical and computer engineering from the University of Illinois, Urbana–Champaign, in 1981 and 1983, respectively. Since then, he has been with the IBM Corporation, where his initial assignment was in the area of performance modeling for one of the Research Division precursors to the IBM RS/6000 processor. His other research interests have been in the areas of high-performance computer architectures, parallel processing, compilers, VLSI testing, and expert systems. Dr. Bose spent a sabbatical year (1989–1990) at the Indian Statistical Institute, Calcutta, and has also taught as an Adjunct Professor at the New York University Courant Institute and at City University of New York (CUNY). He is a senior member of IEEE.

S. Surya *IBM Systems Technology and Architecture Division, 11400 Burnet Road, Austin, Texas 78758 (SURYA at AUSVM6).* Mr. Surya is a staff engineer at the IBM Somerset Design Center, Austin, Texas. He received a B.Tech degree in electrical engineering from the Indian Institute of Technology, Bombay, in 1985 and an M.S. in computer science from North Carolina State University, Raleigh, in 1987. Prior to joining IBM, Mr. Surya worked for Hewlett Packard and BBN Communications, in the areas of systems performance and network modeling/performance, respectively. At Somerset, he has been involved in processor performance modeling and validation activity for the PowerPC 6XX microprocessor family. Mr. Surya is also a Ph.D. candidate at the University of Texas; he is doing research in the area of architectural timing verification.