POWER2 CPU-Intensive Workload Performance

Sohel R. Saiyed, J. Michael O'Connor, and Maurice Franklin

Introduction

The POWER2 processor provides industry-leading performance across a broad range of applications in the workstation and server markets. One significant market segment consists of applications commonly characterized as *CPU-intensive workloads*. The Systems Performance Evaluation Corporation's (SPEC) integer (CINT92) and floating-point (CFP92) benchmarks represent these workloads. The POWER2 processor has the highest performance results on both benchmarks (as of the announcement of the POWER2 products). This achievement is possible due to the significant performance improvement of POWER2 over its predecessor, POWER. The SPECfp92 and SPECint92 ratings for POWER2 exceed those of the highest performing POWER implementation by factors of 1.9 and 1.7, respectively. (When models are significant to the discussion, we use the IBM RS/6000 Model 990 to represent the POWER2 processor and the Model 980 to represent the high-end POWER implementation.) These speedups result from a combination of compiler and hardware design enhancements.

First, we describe the performance improvements provided by the latest versions of the compilers (C Set ++ 2.1 for C code and XLF 3.1 for Fortran code) that implement new optimizations to exploit the new features of the POWER2. Next, we discuss the performance gains obtained from various POWER2 architectural and implementation improvements. These improvements include a faster clock, additional functional units, improved caches, and new instructions [1,2,3,4].

The SPEC Benchmark Suites

Throughout this paper, the SPEC integer and floating-point benchmark suites are the basis for evaluating the POWER2 performance improvements. These SPEC suites are widely accepted measures of workstation performance, especially among computer system users engaged in computationally intensive engineering and scientific work. The integer suite (CINT92) consists of six programs; the floating-point suite (CFP92) has fourteen programs. Tables 1 and 2 list some characteristics of these programs [5].

Compiler Improvements

To achieve the full potential of the POWER2 architectural features, new compilers include enhancements such as performing more aggressive high-order transformations, scheduling instructions to take maximum advantage of the dual integer (Fixed-Point Units, or FXUs) and dual Floating-Point Units (FPUs), and exploiting the new POWER2 instructions.

Loop Unrolling

Of the high-order transformations in the new compilers, loop unrolling is the most important optimization for the POWER2. Unrolling consists of replicating the body of a loop by some factor and reducing the iteration count by an equivalent factor. For example, the following simple loop is shown, both before and after being unrolled by a factor of four:

```
Before Unrolling:
DO J=1,1000
   SUM(J)  = OFFSET + X(J) * Y(J)
ENDDO


After Unrolling:
DO J=1,1000,4
   SUM(J)   = OFFSET + X(J)   * Y(J)
   SUM(J+1) = OFFSET + X(J+1) * Y(J+1)
   SUM(J+2) = OFFSET + X(J+2) * Y(J+2)
   SUM(J+3) = OFFSET + X(J+3) * Y(J+3)
ENDDO
```

Program	Lang.	Description
008.espresso	C	Logic optimizer
022.li	C	Lisp interpreter
023.eqntott	C	Truth table generator
026.compress	C	Lempel-Ziv compression
072.sc	C	Spreadsheet
085.gcc	C	C compiler

Table 1 SPEC CINT92 Programs

On some architectures, unrolling avoids the branch penalty overhead associated with each loop iteration. This justification for loop unrolling is not generally valid on POWER and POWER2 since both often achieve zero-cycle branches.

Unrolling does provide several other benefits on POWER2. First, unrolling provides an opportunity to expose the parallelism between successive loop iterations by creating a substantially larger basic block (the sequence of nonbranch instructions between branches) for the body of the loop. The larger basic block permits the compiler's instruction scheduler to make more efficient use of the multiple functional units because there are more opportunities to schedule instructions in otherwise "dead" cycles. In the preceding code example, four independent floating-point multiply-add (fma) instructions will keep both FPUs busy. Hence, unrolling enables the compiler to expose greater instruction-level parallelism to the POWER2 hardware.

Second, unrolling often creates opportunities to utilize quad-word storage reference instructions. These new POWER2 operations load (or store) two 64-bit floating-point operands into registers in a single memory access. In cases where successive iterations of a loop access sequential elements of an array, unrolling the loop by a factor of two or more allows two instances of a Load Double instruction (64-bit load) from successive iterations to be replaced by a single Load Quad (128-bit load). For floating-point codes dominated by storage reference instructions, this can result in a substantial performance improvement.

Program	Language	Precision	Description
013.spice2g6	Fortran	Double	Analog circuit simulation
015.doduc	Fortran	Double	Monte-Carlo simulation of nuclear reactor
034.mdljdp2	Fortran	Double	Atomic motion equation solver
039.wave5	Fortran	Single	Solves particle and Maxwell's equations
047.tomcatv	Fortran	Double	Vectorized mesh generation
048.ora	Fortran	Double	Ray tracing
052.alvinn	C	Single	Neural network training
056.ear	C	Single	Simulation of the human ear using FFTs
077.mdljsp2	Fortran	Single	Single-precision version of 034.mdljdp2
078.swm256	Fortran	Single	Solves shallow water equations
089.su2cor	Fortran	Double	Computes Quark Gluon theory particle masses
090.hydro2d	Fortran	Double	Solves hydrodynamical Navier Stokes equations
093.nasa7	Fortran	Double	Seven kernels often used in NASA applications
094.fpppp	Fortran	Double	Calculates multi-electron integral derivatives

Table 2 SPEC CFP92 Programs

A third advantage of unrolling is that it exposes parallelism between long-latency instructions (divide and square-root) across loop iterations. In programs dominated by long-latency instructions, the parallelism among such instructions significantly affects performance [6]. For instance, in a floating-point loop where a scalar is divided by each element of an array, the loop spends most of its time performing the 17-cycle divide. Unrolling often makes it easier for the compiler to expose parallelism of two independent divide (or square root) operations. In this case, unrolling can result in effectively 8.5 cycles per floating-point divide. In loops with long latency instructions, unrolling is a particularly useful technique.

Although loop unrolling increases code size, this does not greatly impact the SPEC performance on POWER2 because the POWER2 instruction cache is large with respect to the SPEC programs, which have small code footprints (the set of unique instruction cache lines touched during a program's execution). The more serious drawback to unrolling is an increase in register use. An unrolled loop has more unique variables, increasing the number of registers needed to retain these variables. This increased register usage might increase the amount of *spill code*–instructions that save and later restore the values of registers to or from memory, making the registers available for other variables. Because this spill code can often adversely affect performance, the compiler applies heuristics to determine how much unrolling should be applied to a given loop. For a more thorough discussion of loop unrolling and other high-order transforms, see [7].

To illustrate the importance of loop unrolling on actual code, consider an inner loop from the SPEC floating-point benchmark 052.alvinn:

```
for (hu = 0; hu < (30+1); hu++)
{
  psum_array[hu] += delta[ou] *
          h_o_weights[ou][hu];
  h_o_w_ch_sum_array[ou][hu] +=
          delta[ou] * hidden_act[hu];
}
```

Without any unrolling, the compiler might generate the following unoptimized code sequence for the main body of the loop. The Load Double with Update (lfdu) instructions are loading the successive elements of the *h_o_weights* and *hidden_act* arrays. The Load Double (lfd) instructions are loading the successive elements of the *h_o_w_ch_sum_array* and *psum_array* arrays while the Store Double with Update (stfdu) instructions store back the results. The floating-point multiply-add (fma) instructions perform the required arithmetic operations. Because storage references dominate this loop, the FXU's ability to process loads and stores will limit performance:

```
CL.54:
lfdu  fp5,gr3=hidden_act(gr3,8)
lfd   fp4=h_o_w_ch_sum_array(gr7,8)
lfdu  fp3,gr6=h_o_weights(gr6,8)
lfd   fp2=psum_array(gr4,8)
fma   fp4=fp4,fp1,fp5,fcr
fma   fp2=fp2,fp1,fp3,fcr
stfdu gr7,h_o_w_ch_sum_array(gr7,8)=fp4
stfdu gr4,psum_array(gr4,8)=fp2
bc    ctr=CL.54
```

Invoked with an appropriate optimization level, the compiler might unroll the loop by a factor of two and generate the code that follows. (In reality, the compiler or preprocessors will typically unroll by a factor of four or more. For the sake of simplicity, this example used human unrolling.)

```
CL.54:
lfqu  fp4,fp5,gr6=hidden_act(gr6,16)
lfq   fp8,fp9=h_o_w_ch_sum_array(gr4,8)
lfqu  fp2,fp3,gr7=h_o_weights(gr7,16)
lfq   fp6,fp7=psum_array(gr3,16)
fma   fp4=fp8,fp0,fp4,fcr
fma   fp5=fp9,fp0,fp5,fcr
fma   fp2=fp6,fp0,fp2,fcr
fma   fp3=fp7,fp0,fp3,fcr
stfqu gr4,h_o_w_ch_sum_array(gr4,16)=
                              fp4,fp5
stfqu gr3,psum_array(gr3,16)=fp2,fp3
bc    ctr=CL.54
```

The Load Quad (lfq) instruction loads two successive array elements from each of the *psum_array* and *h_o_w_ch_sum_array* arrays; the Store Quad with Update (stfqu) instructions write back these array elements. Finally, the Load Quad with Update (lfqu) instructions load successive pairs of elements from the *h_o_weights* and the *hidden_act* arrays. The unrolled loop requires three cycles for every two iterations while the original loop requires three cycles for only a single iteration. Thus, unrolling improves the performance of this storage reference limited loop by a factor of two.

Dual Instruction Unit Scheduling

Integer programs typically have less exploitable parallelism than floating-point programs [8]. Short basic blocks with frequent branches characterize integer programs. For the SPEC integer benchmarks on POWER2, the average number of instructions between branches ranges from 2.6 to 4.0 per benchmark with an overall average of 3.4 for the suite. These frequent branches tend to limit the exploitable parallelism. Often the compiler can not reorder instructions around branches, severely limiting its ability to find independent instructions that fully utilize POWER2's dual integer units.

In contrast, the floating-point benchmarks have many instructions between branches. In the SPEC CFP92 suite, compiled for POWER2, the average number of instructions between branches ranges from 5.1 to 34.7 per benchmark with an overall average of 12.3 for the suite. Many of the branches in these benchmarks are loop-closing branch-on-count instructions, which behave like unconditional branches, rarely limiting hardware parallelism. As shown earlier, work from adjacent iterations often provides independent operations after unrolling. Furthermore, because these benchmarks consist of both fixed-point and floating-point instructions, it is possible to keep all functional units busy. Thus, floating-point benchmarks provide more opportunity to exploit instruction-level parallelism. Therefore, one expects gains from compiler improvements to be greater on POWER2 for the floating-point programs than for the integer programs, as described in the following section.

Performance Enhancements

We define the performance contribution provided by compiler enhancements as the improvement in the execution times of programs compiled with the new POWER2 optimized compilers versus programs compiled with the older compilers (XLC 1.2.1/XLF 2.3) that target the POWER implementation. The process of measuring the contributions consists of using both the new and old compilers to compile the benchmark programs and then running both executables on a POWER2 system. A comparison of run times highlights the improvements in the new compilers, as the effects of the hardware are minimized by running the test on the same hardware platform.

On the SPEC integer benchmarks, the new compilers provide up to 15.7% improvement, and the geometric mean of the improvements is 5.3%. These gains are small because only limited opportunities exist in which to schedule the fixed-point instructions to specifically take advantage of the second FXU. There are no new integer instructions for the compiler to generate, and the integer benchmarks are not typically good candidates for improvements through high-order loop transformations. Due to these factors, the SPEC CINT92 suite and other integer workloads show only minor gains from POWER2 compiler enhancements; rather, the SPECint92 gain for POWER2 is largely due to hardware improvements, discussed in the following section.

The performance improvement seen by using the latest compilers on the floating-point suite is much larger than the gain seen on the integer suite, providing a 53.5% maximum improvement while the geometric mean is 17.6%. This improvement is partially explained by the fact that the compilers can take advantage of the new floating-point instructions, such as quad-word storage references and Square Root. Also, the high-order loop transformations, such as loop unrolling, are generally much more effective on the highly regular and repetitive code of scientific programs common in the SPEC floating-point suite. Finally, the compiler is able to schedule the code so that the second FPU and second FXU can be efficiently exploited, in large part due to the long basic blocks inherent in the floating-point benchmarks. These factors combine to give the latest compilers a substantial performance advantage on POWER2 over the earlier versions of the compilers for floating-point code, such as the CFP92 benchmarks.

Hardware Improvements

Gains due to hardware improvements in POWER2 account for another large portion of the performance improvements. These improvements primarily come from a combination of a faster clock, more functional units, improved caches, and new instructions. In this paper, the POWER2 performance monitor facility [9] provided the measurements to evaluate the impact of the various POWER2 hardware features. Special-purpose hardware [9] provided the data for a POWER system.

Faster Clock

One hardware component of the performance improvement in POWER2 over POWER is a faster clock rate. The fastest POWER system, a Model 980, operates at 62.5 MHz; the POWER2 used in our discussion, a Model 990, operates at 71.5 MHz, a 14% gain. However, this contributes only a small portion of the overall performance improvement realized by POWER2, as other architectural and implementation improvements provide the bulk of the gains.

More Functional Units

POWER2's additional functional units provide significant benefits. POWER2 has twice as many functional units as POWER; two integer units and two floating-point units. Thus, POWER2 hardware is able to exploit the instruction-level parallelism exposed by the compiler to a substantially greater degree.

One measure of the exploited instruction-level parallelism is the instructions-per-cycle (IPC) ratio. This is the average number of instructions that are executed per clock cycle. To examine the effects of additional functional units alone, we ignore the penalty cycles associated with cache misses. Using the same POWER-targeted executables, Tables 3 and 4 compare the exploited parallelism of the Models 980 and 990, based on infinite-cache IPC. (The infinite-cache IPC is the IPC the workload would exhibit if the caches were infinite in size, thereby satisfying all memory references from cache.) As shown, the POWER2 achieves a marked improvement for this performance component.

The POWER2 IPC metric for 048.ora is the only one shown that is lower than POWER, which would appear to be counter intuitive given that 048.ora has significantly higher overall performance on POWER2. This situation results from the fact that 048.ora is a very square-root intensive benchmark. On the POWER processor, a 54-instruction library routine performs the square root operation in roughly 50–55 cycles. In contrast, on POWER2, a single hardware Square Root instruction requires about 26 cycles. Thus, a single instruction executing in 26 cycles replaces 54 instructions executing in 50–55 cycles. This causes a substantial drop in IPC while boosting net performance. To a lesser degree, a similar effect results from replacing two Load Double operations with a single Load Quad, causing the IPC to slightly underestimate the overall performance gain.

Another way to assess the benefit of the additional functional units is to examine how often a given functional unit is busy. Tables 5 and 6 show the percentage of cycles that a given functional unit was busy. The tables also contain the percentage of time that the second functional unit was busy when there were instructions of the appropriate type (fixed-point or floating-point) available to execute. This utilization number is a good indicator of how much parallelism is exploited. Ideally, if the utilization was 100%, then the second unit would be busy whenever the first functional unit is busy. However, dependencies among the instructions make this an unlikely prospect.

Although 013.spice2g6 is a member of the floating-point suite, it is included in the FXU-limited table because it consists of only 5.9% floating-point instructions; thus, FXU utilization primarily determines its performance. This data shows that the FXU-limited benchmarks make efficient use of the two integer units. FXU1 is busy between 48% and 64% of the time that fixed-point instructions are available. Although not shown in Table 6, the FPU-limited benchmarks also take advantage of the second FXU, often keeping it utilized as much as the FXU-limited benchmarks.

The FPU-limited benchmarks generally make good use of the second FPU. For instance,

Integer Benchmark	Model 980	Model 990	Percent Change
008.espresso	1.00	1.27	35%
022.li	0.85	1.26	48%
023.eqntott	1.19	1.89	59%
026.compress	0.91	1.32	45%
072.sc	0.84	1.21	44%
085.gcc	0.83	1.13	36%

Table 3 SPEC CINT92 Infinite Cache IPC

Floating-Point Benchmark	Model 980	Model 990	Percent Change
013.spice2g6	0.89	1.23	38%
015.doduc	0.83	1.08	30%
034.mdljdp2	0.81	0.96	19%
039.wave5	1.04	1.62	56%
047.tomcatv	1.21	2.66	120%
048.ora	0.85	0.54	–37%
052.alvinn	1.44	4.60	219%
056.ear	1.15	1.68	46%
077.mdljsp2	0.69	0.83	20%
078.swm256	1.23	2.92	137%
089.su2cor	1.19	1.70	43%
090.hydro2d	0.84	1.09	30%
093.nasa7	0.92	2.43	164%
094.fpppp	0.93	1.57	69%

Table 4 SPEC CFP92 Infinite Cache IPC

078.swm256 makes the best use of the additional floating-point unit, keeping it busy 90% of the time that floating-point instructions are available for execution.

048.ora exhibits the unusual behavior of FPU1 utilization being significantly higher than FPU0. One explanation of this involves 048.ora's heavy usage (about 30% of CPU time) of Square Root instructions. If, during the first of several iterations, a single long-running instruction (in this case Square Root) enters one pipeline (in this case FPU1), it may allow other pipelines to drain.

If the other pipelines drain while this instruction executes, there is an effective synchronization of subsequent instructions and pipelines; instruction issuance beyond the square root will occur in a deterministic manner. If the same instruction again goes to the same pipeline on the second iteration, then the synchronization repeats and the same pipeline should process the square root operation each iteration. Apparently in the case of 048.ora, FPU1 always executes the square root. Adding one instruction between instances of the Square Root instruction may cause it to shift to FPU0 or even to alternate between pipelines.

Increased Data Cache Capacity

The POWER2 data cache is four times larger than that on high-end POWER implementations, while the instruction cache is the same size. In addition, the POWER2 data and instruction caches have longer lines (the unit of transfer between memory and the cache). Table 7 summarizes these changes. (Sizes are in bytes). Finally, the width of the bus from cache to main memory is 32 bytes on POWER2, twice that of POWER systems.

Based upon measurement alone, it is difficult to separate the impact of line-size and cache-size improvements. Because the instruction cache miss rates for the workloads under study are very small (generally less than 0.1%), the impact of these misses is negligible. (See "POWER2 Commercial Workload Performance" [10] for a discussion of the effects of longer I-cache

FXU-Limited Benchmark	Percentage of Cycles Busy		Util. 2nd FXU
	FXU0	FXU1	
008.espresso	64%	31%	49%
013.spice2g6	61%	29%	48%
022.li	65%	33%	51%
023.eqntott	66%	55%	83%
026.compress	53%	29%	55%
072.sc	54%	31%	58%
085.gcc	51%	27%	53%

Table 5 Utilization of FXUs

FPU-Limited Benchmark	Percentage of Cycles Busy		Util. 2nd FPU
	FPU0	FPU1	
015.doduc	73%	25%	29%
034.mdljdp2	72%	52%	44%
039.wave5	69%	47%	63%
047.tomcatv	81%	68%	84%
048.ora	52%	91%	45%
052.alvinn	66%	58%	88%
056.ear	75%	50%	66%
077.mdljsp2	76%	16%	20%
078.swm256	91%	87%	92%
089.su2cor	73%	43%	53%
090.hydro2d	71%	39%	49%
093.nasa7	63%	37%	58%
094.fpppp	66%	33%	46%

Table 6 Utilization of FPUs

lines on other workloads.) As for the D-cache, its twice-as-long cache lines will dramatically reduce the miss rates of programs that have a high frequency of stride-one storage accesses (such as tomcatv, alvinn, and compress). Programs sequentially accessing successive memory locations will only cause a cache miss due to "striding off the end" of a cache line once every 64 words rather than once every 32 words. (Since POWER2 also doubles the bus bandwidth, the memory subsystem refills a cache line as fast on POWER2 as on POWER, despite POWER2's longer lines.) Most of these programs also should benefit significantly from the quadrupled capacity of the POWER2 D-cache. Tables 8 and 9 show that D-cache miss per reference rates of the various SPEC benchmarks significantly decrease (improve) due to increased line length and cache capacity.

Note that the data presented here is for the first-level (L1) data cache. None of the performance numbers in this paper include the effects of a second-level (L2) cache. Many other workstation vendors include large L2 caches. For instance, the DEC AXP 10000 Model 610 includes a 4MB L2 cache. Because POWER2 has a large L1 cache, the performance gain from an L2 cache is small for many workloads. An L2 cache can be more beneficial for programs that have relatively high data cache miss rates, such as compress, spice2g6, tomcatv, or nasa7, and workloads with large instruction footprints (and high instruction cache miss rates), such as database applications.

New Instructions

POWER2 adds several new instructions. For performance on the SPEC benchmarks, the most important are quad-word storage references and Square Root. The 048.ora benchmark clearly shows the dramatic impact the POWER2's hardware square root can provide for square root intensive code. The execution time of 048.ora drops by 25% when this new instruction is generated by the compiler.

Evaluating the effect of quad-word storage instructions (with a prototype compiler that prevents Load Quad instructions from being generated) shows a modest overall gain of 2.2 percent. As the preceding unrolling example shows, the Load Quad instruction provides the greatest improvements on programs that perform many stride-one accesses. Such a program, 052.alvinn, has a gain of 23%, the largest Load Quad gain seen in the SPEC suite. Other benchmarks, such as Linpack, show even greater gains from Load Quad. Furthermore, compiler technology should continue to provide additional quad-word storage reference improvements.

Model	Cache Type	Cache Size	Line Size	Assoc.
Model 980	Instr.	32k	64	2-way
	Data	64k	128	4-way
Model 990	Instr.	32k	128	2-way
	Data	256k	256	4-way

Table 7 980 and 990 Cache Attributes

Integer Benchmark	Miss Rate	
	POWER	POWER2
008.espresso	0.26%	0.06%
022.li	0.13%	0.004%
023.eqntott	3.33%	1.05%
026.compress	7.28%	1.96%
072.sc	1.31%	0.19%
085.gcc	0.61%	0.07%

Table 8 SPEC CINT92 D-Cache Miss Rates

Floating-Point Benchmark	Miss Rate	
	POWER	POWER2
013.spice2g6	6.66%	1.49%
015.doduc	0.43%	0.00%
034.mdljdp2	0.55%	0.01%
039.wave5	0.34%	0.09%
047.tomcatv	2.44%	1.39%
048.ora	0.01%	0.002%
052.alvinn	1.30%	2.30%
056.ear	0.01%	0.007%
077.mdljsp2	0.78%	0.02%
078.swm256	1.79%	1.01%
089.su2cor	2.11%	0.51%
090.hydro2d	3.85%	0.61%
093.nasa7	4.21%	2.86%
094.fpppp	0.06%	0.00%

Table 9 SPEC CFP92 D-Cache Miss Rates

Conclusions

The architectural, implementation, and compiler improvements of POWER2 systems combine to provide industry-leading performance. At 71.5 MHz, POWER2 has a SPECint92 of 126.0 and a SPECfp92 of 260.4. This gives POWER2 the best SPECint92 and SPECfp92 in the industry at the time of its announcement.

Acknowledgements

We would like to thank Da-Gung Lu for his help with compiler option selection and Kate Stewart for answering compiler related questions. We would like to particularly thank Ed Welbon and Chris Chan-Nui for their invaluable assistance in making hardware measurements.

References

1. Steven W. White and Sudhir Dhawan, "POWER2: Next Generation of the RISC System/6000 Family," *PowerPC and POWER2: Technical Aspects of the New IBM RISC System/6000*, pp. 8–18.
2. Baba Arimilli, Jama Barreh, Robert Golla, and Paul Jordan, "POWER2 Instruction Cache Unit," *PowerPC and POWER2: Technical Aspects of the New IBM RISC System/6000*, pp. 19–28.
3. D. J. Shippy, T. W. Griffith, and Geordie Braceras, "POWER2 Fixed Point, Data Cache, and Storage Control Units," *PowerPC and POWER2: Technical Aspects of the New IBM RISC System/6000*, pp. 29–44.
4. Troy N. Hicks, Richard E. Fry, and Paul E. Harvey, "POWER2 Floating-Point Unit: Architecture and Implementation," *PowerPC and POWER2: Technical Aspects of the New IBM RISC System/6000*, pp. 45–54.
5. "CINT92 & CFP92 Benchmark Descriptions," *SPEC Newsletter*, Volume 4, Issue 4, December 1992; Standard Performance Evaluation Corporation, p. 9.
6. Norman P. Jouppi, "The Nonuniform Distribution of Instruction-Level and Machine Parallelism and Its Effect on Performance," *IEEE Transactions on Computers*, Vol. C-38, No. 12, December 1989, pp. 1645–1658.
7. *AIX Version 3.2 for RISC System/6000 Optimization and Tuning Guide for XL Fortran, XL C, and XL C ++*, SC09-1705, IBM Corporation.
8. Monica S. Lam and Robert P. Wilson, "Limits of Control Flow on Parallelism," *Proc. of the 19th Annual International Symposium on Computer Architecture*, May 1992, pp. 46–57.
9. E.H. Welbon, C.C. Chan-Nui, D.J. Shippy, and D.A. Hicks, "POWER2 Performance Monitor," *PowerPC and POWER2: Technical Aspects of the New IBM RISC System/6000*, pp. 55–63.
10. Maurice Franklin, William Alexander, Rajiv Jauhari, Ann Marie Grizzaffi Maynard, and Bret Olszewski, "POWER2 Commercial Workload Performance," *PowerPC and POWER2: Technical Aspects of the New IBM RISC System/6000*, pp. 137–144.