A Benchmark Characterization of the EEMBC Benchmark Suite

Benchmark consumers expect benchmark suites to be complete, accurate, and consistent, and benchmark scores serve as relative measures of performance. However, it is important to understand how benchmarks stress the processors that they aim to test. This study explores the stress points of the EEMBC embedded benchmark suite using the benchmark characterization technique.

Jason A. Poovey
Thomas M. Conte
Georgia Institute of Technology
Markus Levy
Shay Gal-On
Embedded Microprocessor Benchmark Consortium

Benchmarking is as much an art as it is a science, and at times involves a leap of faith for the benchmark consumer. Most people trust benchmark suites to be complete, accurate, and consistent. By default, when used to compare one processor to another, benchmark scores serve as a relative measure of performance. Rarely do the benchmark consumers understand or verify the underlying stresses that the benchmarks place on the processors. This study examines the stress points of the Embedded Microprocessor Benchmark Consortium (EEMBC) suite, using the *benchmark characterization* technique.

Benchmark characterization involves finding a set of unique characteristics to classify benchmarks.[1] These characteristics should be predictive of performance on a wide variety of platforms, and should allow benchmark consumers to find good proxies for their own applications in the benchmark set. The technique aims to find just the right characteristics to reveal the benchmarks' required hardware capacities. Thus, system designers can use benchmark characterization to classify benchmarks and to discover if two benchmarks stress the system in similar or different ways.

Core concepts

Processor performance depends on the types of workloads used, and designers typically use benchmark suites with the initial assumption that the workloads they provide are representative of user programs. EEMBC benchmarks represent a wide variety of workloads and provide an industry standard tool for understanding processor performance. However, few people truly understand what these benchmarks are doing. EEMBC benchmarks attempt to represent user programs from several embedded disciplines. By understanding these benchmarks' inherent program behavior, system developers can make comparisons to their own applications.

Designers and researchers can use benchmark characterization to parameterize a workload so that it is quantifiable by several abstract attributes. They then can use the measured attributes to indicate program similarity. Using these characteristics, along with inherent knowledge of the actual application

Published by the IEEE Computer Society

Table 1. Example benchmarks from the EEMBC benchmark suites.

Suite	Benchmarks
Automotive	Finite and infinite impulse response (FIR and IIR) filters, tooth-to-spark tests, pulse-width modulation, matrix multiplication and shifting, table lookup, and fast Fourier transform (FFT)
Consumer	JPEG compression and decompression, high pass gray-scale filter, RGB to CMYK, and RGB to YIQ converter
Digital entertainment	JPEG compression and decompression, high pass gray-scale filter, RGB to CMYK, RGB to YIQ converter, Advanced Encryption Standard (AES), and Data Encryption Standard (DES)
Networking	Packet flow algorithms, Open Shortest Path First (OSPF), and route lookup
Networking v2.0	Packet check algorithms, OSPF, RSA, and network address translator (NAT)
Office automation	Dithering, rotate, and text
Telecommunications	Autocorrelation, FFT, and Viterbi decoder

target for a system, helps them select the most relevant benchmarks from the EEMBC suites.

Conte and Hwu argue that benchmark characterization should use metrics that reflect a benchmark's preferred architectural design point.[1] Other researchers have performed similar analyses using many different techniques.[2-6] Trace simulation, statistical analysis, binary instrumentation, and on-the-fly analysis all attempt to capture inherent program characteristics. For example, a designer can run simulations to find the natural cache size such that the cache encounters only compulsory misses (that is, misses that occur when the program first requests data), but never misses because of limited capacity. The instruction set also greatly affects benchmark behavior, so it is useful to determine the mix of function units in the processor pipeline that causes the benchmark to run most efficiently. In this article, we use several microarchitecture-dependent and independent attributes to characterize the EEMBC benchmark suites. Because microarchitecture-dependent and independent attributes are simply measurements of system events, our research also incorporates target-directed performance attributes. Thus, we characterize the EEMBC benchmarks not only on program behavior, but also on the hardware requirements needed to achieve specified performance targets. Using a combination of these elements better equips designers to make educated design choices when developing new systems.

The EEMBC benchmark suites analyzed

The EEMBC benchmark suites are a collection of algorithms and applications organized into various categories to target a broad range of users. We derive the details presented here from seven of these suites: automotive, consumer, digital entertainment, networking, networking v2.0, office automation, and telecommunication. Other suites not tested as of this writing include the recent office automation 2.0 as well as a recently released comprehensive suite for testing multicore platforms.

Table 1 shows examples of the types of workloads available in each suite. EEMBC has improved several suites by creating second versions (for example, the second versions of the networking and consumer suites are networking v2.0 and digital entertainment, respectively). In this study, we test both versions when possible.

Some benchmarks are included in multiple suites. Although the tasks performed in common benchmarks are similar, the code is not always identical. The benchmark suite designers place similar benchmarks in multiple categories because some algorithms are used in multiple embedded disciplines. For example, fast Fourier transforms are useful in both automotive and telecommunications applications, so both suites include

the FFT algorithm. Application developers might wish to test only the suite relevant to their application and avoid scouring unrelated suites for the most compatible workload.

Developing the methodology

To generalize the benchmark characterization results, we devised a method to describe application behavior independently of the underlying platform. We collected EEMBC benchmark characteristics from the MIPS, PowerPC, ARM, PISA, and x86 architectures. This provided a mix of instruction set architectures (ISAs) and different processor design methodologies (complex-instruction-set computing, reduced-instruction-set computing, low power, and small size) and several cross-compilers. For all compilers, we used the default optimization options (–O2 or equivalent) to level the field and not give an unfair advantage to the cross-compilers that support more optimizations. The resulting combination of metrics provides an accurate representation of the workload's activity. Thus, designers can determine which workloads are similar to their own, and then leverage the most relevant benchmarks as a proxy for architecture design. The characteristics themselves assist in this design by determining the minimum hardware requirements needed to achieve the target performance.

We achieved characterization primarily through trace-driven simulation. We used traces to collect data from cache design experiments and the dynamic instruction distribution. We performed all cache performance experiments using a single pass cache simulator that simultaneously evaluated the performance of multiple cache configurations using the least recently used (LRU) stack algorithm.[7] This method is similar to the cache analysis in many previous works.[7-9] We gathered function unit requirements by simulating an idealized MIPS superscalar machine. To remove all effects other than true dependencies, the idealized machine assumes perfect branch prediction, a perfect cache, and infinite issue width. We tracked usage throughout execution to determine the minimum number of function units necessary to satisfy the function unit needs 85 percent of the time (85 percent utilization).

It is sometimes difficult to understand which characteristics are inherent to the workload, and which are microarchitecture dependent (affected by the compiler, ISA, or hardware design). Some researchers look at characteristics that are common, assuming a specific ISA,[4] whereas others use analytical or statistical code analysis. We generated results using different compilers and architectures, and chose only characteristics that produced similar results on all four architectures and tool chains.

Figure 1 shows an example result summary from the cache simulation for a single benchmark. The *y*-axis denotes the minimum required cache size for a given set associativity and desired miss rate. The full results for the benchmark include any miss rate and associativity, but even this summary shows that including a set associativity of 2, 3, or 4 produces results that are clearly architecture or compiler dependent. In contrast, for this benchmark, fully associative (not shown) and direct-mapped results are similar for required miss rates of 1 or 0.1 percent. The full results therefore show that a miss rate of 1 percent or better gives the most architecture- and compiler-independent results.

Caches are defined in terms of block size, set associativity, and total size. Cache performance can vary significantly if any of these variables change. Therefore, performing an entire design space search is often costly because each cache configuration of interest requires additional simulation. However, a single-pass simulator simultaneously evaluates caches within a specified range of these three variables. Thus, rather than picking particular cache configurations, this tool isolates cache configurations that meet user-specified performance goals. We use the single-pass simulator to characterize workloads in terms of hardware requirements, rather than simply measuring the performance of a particular cache realization. We chose miss ratios of 1 and 0.1 percent as target performance goals to indentify cache sizes that would achieve the desired performance for level 1 and level 2 caches, respectively.[7]

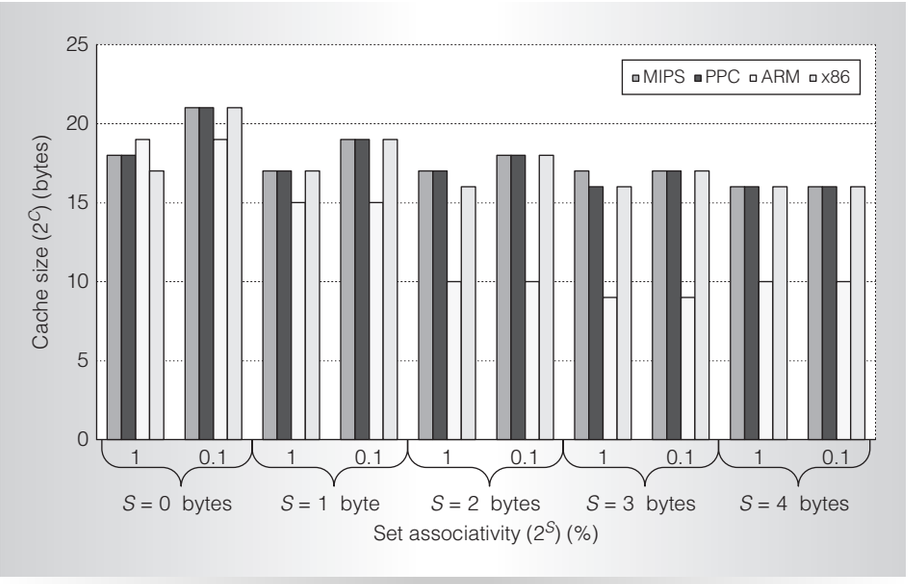

Figure 1. Cache summary example for the packet check 1-Mbyte benchmark.

If cold misses cause a cache configuration to have a higher miss rate than the user-specified target, we target the intrinsic miss ratio.

When designing a new system, designers must decide how many of each function unit type to include. Rather than iteratively modifying the number of function unit types and resimulating, we perform a single simulation to determine the optimal distribution. The function unit distribution simulates an idealized out-of-order machine of infinite width and a perfect branch predictor, thus making true dependencies between instructions the only bottleneck. At each clock tick, the simulator records the number of function units of each type required to service all in-flight instructions in the execute stage. In this study, the distribution results represent the number and type of function units necessary to meet workload demands for 85 percent of the execution time. Or, more simply, how many units of a given type are needed so that lack of those units will not limit workload execution 85 percent of the time. This target point allows for the best cost-benefit performance.[10] For this study, we assume five function unit types: integer ALUs, load/store units, multiply/divide units, branch units, and shift units.

The analysis

The function unit distributions demonstrate the hardware requirements needed to achieve near-maximum parallelism on an idealized machine. We visualize this data using Kiviat graphs. Kiviat graphs plot multiple variables on separate axes using a spider plot to provide a graphical representation of the simulation metrics. This makes it easier to visualize multivariable data in a way that reveals program behavior.[11] Figure 2 summarizes the function unit experiment results and groups them by EEMBC suite to show each suite's overall average tendencies. As the figure shows, each suite's behavior is unique, which is expected because they target different user audiences. This uniqueness is also significant because it illustrates each suite's application-specific nature and suggests that more than one suite must be executed to comprehend a particular processing platform's full capabilities.

The networking v2.0 suite has more function unit requirements than the first version, indicating that it offers greater instruction-level parallelism. The automotive suite exhibits similar strains on function units as the networking v2.0 suite; however, networking v2.0 has slightly higher branch instruction requirements. Therefore, computationally

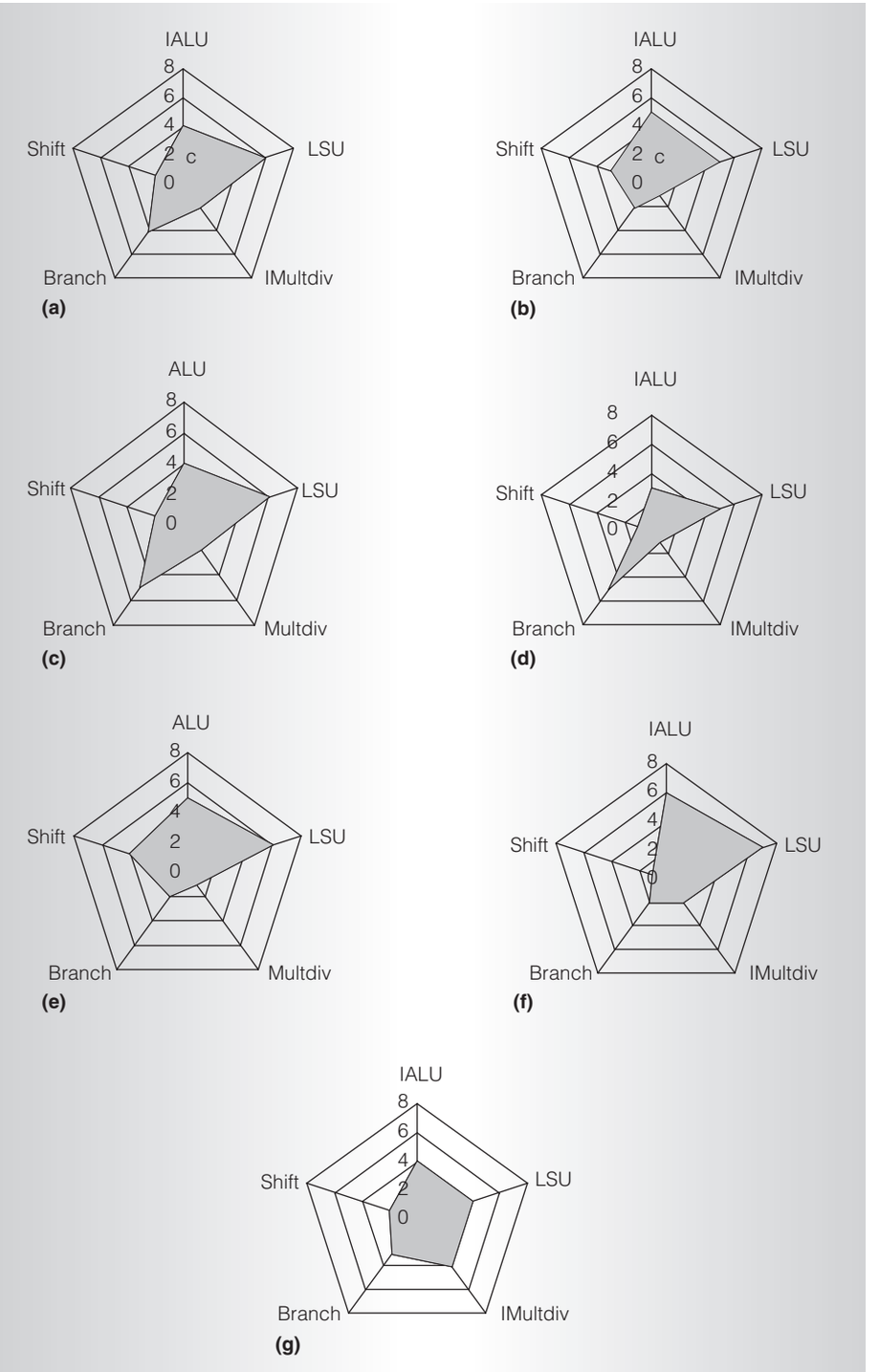

Figure 2. Kiviat graphs of function unit distribution for each benchmark suite at 85 percent utilization: automotive (a), consumer (b), digital entertainment (c), networking (d), networking v2.0 (e), office automation (f), and telecom (g).

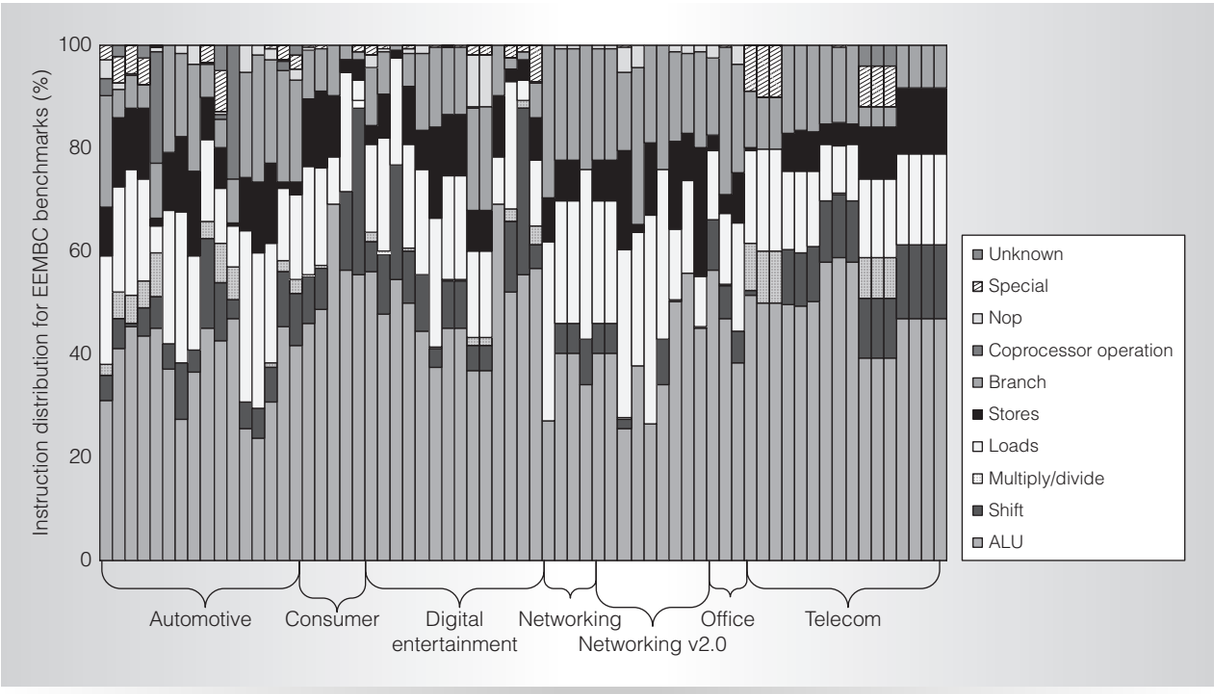

Figure 3. Dynamic instruction distribution.

the dependency chains are similar, but the branches occur more frequently in networking 2.0. This behavior could be due to tight loops for kernel processing, such as the tree traversal in the route lookup benchmark, as well as numerous conditional clauses, such as in the packet check benchmark.

Higher numbers of load/store function units benefit all suites but telecommunications. The lower requirement is not due to a lack of memory instructions, because as the instruction distribution data in Figure 3 indicates, an average number of memory instructions occur in the telecommunications suite execution stream. Rather, this number is due to the high data interdependence in array lookups. For example, the convolutional encoding algorithm uses a central shift register, which is both updated and read within the main algorithm loop, and the fxp bit allocation (fbital00) updates a threshold variable used for bit allocation repeatedly within the main loop.

Our analysis found that the digital entertainment benchmarks, which require four execution units, are the primary stressors of shift function unit requirements. Many digital entertainment benchmarks—such as JPEG compression and decompression and the Huffman decoder algorithm—use shifting heavily. Both the consumer and digital entertainment suites contain similar JPEG manipulation algorithms. Thus, the increase in shift requirements results primarily from the Huffman decoder benchmark. This benchmark uses shifting for traversal through the large input buffer, and for simplifying some calculations into bitwise operations instead of slower multiplication or division operations. The latter optimization creates increased parallelism opportunities. Part of calculating where to look in the buffer involves a bitwise AND with a binary value that is a power of two. To calculate these values, the program shifts the constant 1 left by the appropriate variable. These calculations do not depend on one another, and many values are constants. Therefore, these shifts should have no dependencies preventing instruction parallelism, and thus the Huffman decoder benchmark requires more function units to meet utilization goals.

The office automation suite exhibits a unique behavior, in which only the ALU and LSU exhibit a high degree of parallelism.

Office benchmarks such as bezier and dither use fixed-point math extensively, and the main loop has many calculations involving constants and a single variable. In addition, little interdependence exists among these statements, so the parallelism possibilities are higher. Therefore, we could attribute some of the ALU and LSU parallelism to the many parallel variable loads conjoined with the arithmetic calculation using a constant.

Only the telecom suite requires many multiply and divide units to achieve the utilization goals. The instruction distribution results in Figure 3 show that it is the only suite with a significant number of multiply and divide instructions for several benchmarks in the execution stream. These originate from the autocorrelation benchmark (three data sets) and the fft benchmark (three data sets). The autocorrelation benchmark, at the crux of the algorithm, consists of an integer multiplication of two array entries. Because these variables are unknown at compile time, the compiler cannot make any simplifications. In addition, because this multiplication is a crucial part of the algorithm, it results in a significant number of multiply instructions. Examination of the object file confirms the existence of a mult instruction in the autocorrelation main loop that could not be optimized. Both the fft and aifft benchmarks from the automotive suite use the calculation-heavy fft algorithm. As Figure 4 illustrates, these benchmarks are similar except for the branch misprediction ratio. The primary difference between the code in the two algorithms is the structure of the conditional clauses for the loops and if-then structures that control the algorithm's flow.

Figure 2 illustrates workload variety between the different benchmark suites, and further analysis shows significant variety internal to the particular workload categories. Figure 4 examines each benchmark individually by plotting their workload characteristics on Kiviat plots. As mentioned earlier, these plots allow for a better visualization of each benchmark's architectural characteristics. Table 2 denotes the meaning of each axis on the Kiviat plots.

Similar workloads have similar shapes in the Kiviat graphs. Within the figure, similar workloads' Kiviat graphs are grouped by visual similarity. Within each group, the similar shapes indicate that the relative emphasis among characteristics is similar; however, the emphasis's overall weight varies among benchmarks in the same group. As the figure shows, no single EEMBC suite exhibits completely homogeneous characteristics. Even the consumer suite, which targets applications with similar algorithms (image manipulation), spans two classification categories.

The most heterogeneous suite (automotive) covers four classification categories. For example, the FFT benchmark (aifft) has large cache requirements, whereas the infinite impulse response filter benchmark (iirflt) does not. The pointer chase benchmark (pntrch) exhibits a high percentage of memory accesses, but does not require a significant cache size to obtain the desired performance goals. This is unsurprising for the automotive suite because the algorithms and applications executing in the different parts of a vehicle vary greatly and would require different attributes for specialized processors.

The office automation suite is interesting in that it contains only three benchmarks, each classified into a different category. The rotate image benchmark is in a grouping (group 1) that is insensitive to increasing block sizes, but the cache size requirements are still similar to those of the other two office benchmarks. Therefore, the rotate benchmark stresses workloads with less spatial locality. The text benchmark is in a group with a low percentage of ALU computations and focuses more on variable movement. The main algorithm is simply a large nested switch statement that updates few variables, typically with constants or simple calculations. Finally, the dither benchmark is similar in instruction mix and instructions per cycle (IPC) to the rotate image benchmark but with higher sensitivity to spatial locality because increasing the block size reduces overall cache requirements. We can attribute this higher sensitivity to the algorithm repeatedly indexing into a table with address strides between lookups of only one or two entries.

Workloads from different suites exhibit similar characteristics. For example, the

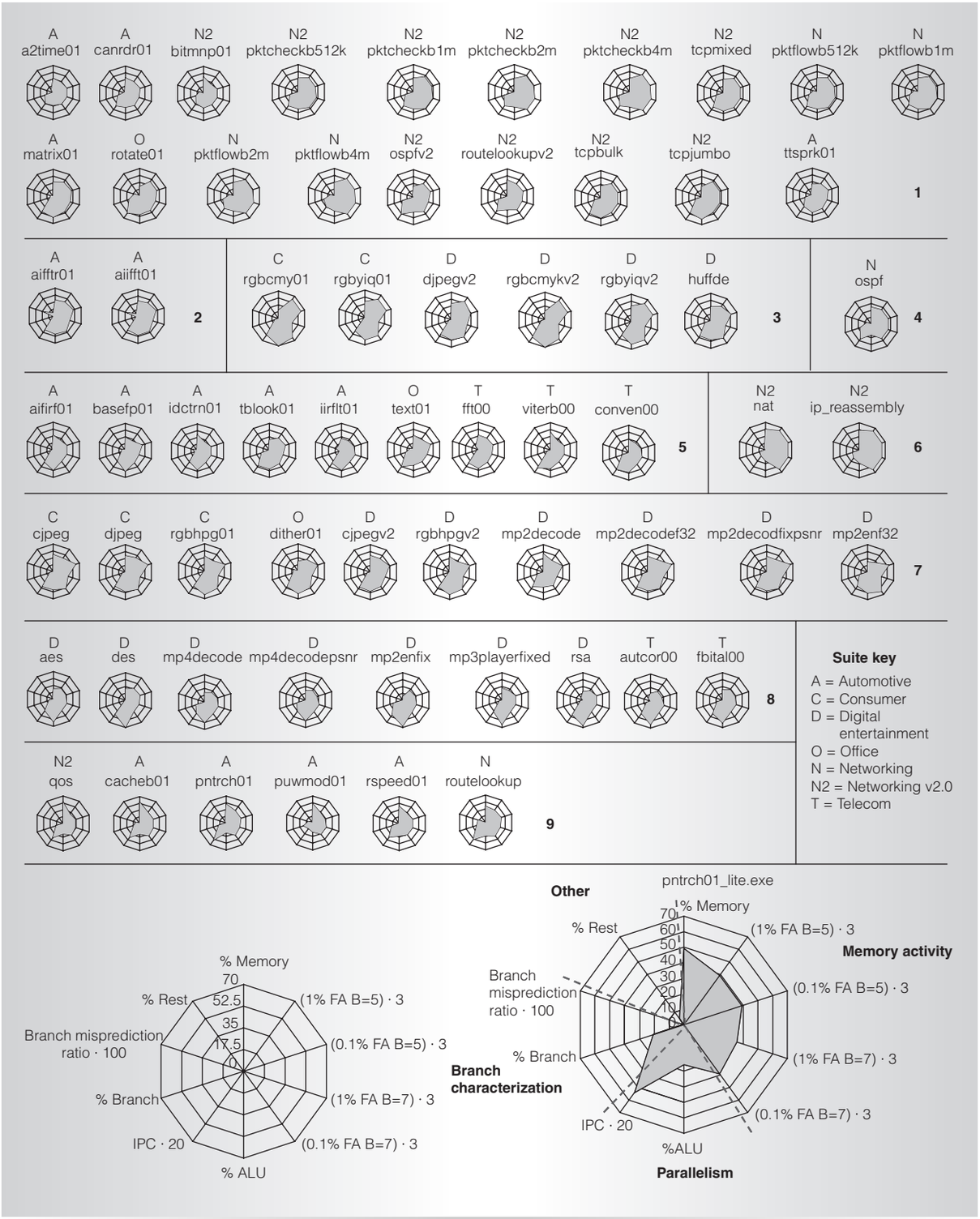

Figure 4. Kiviat plots of combined characteristics.

Table 2. Description of axes for Figure 4.

Axis label	Meaning
(1% FA B=5) · 3 (0.1% FA B=5) · 3 (1% FA B=7) · 3 (0.1% FA B=7) · 3	Log2(C), where C is the cache size in bytes required for a fully associative cache with a block size of 2^B targeting a (1%/0.1%) miss ratio. We multiply this value by 3 to normalize it with other metrics.
% ALU/% Memory/% Branch/% Rest	Dynamic instruction distribution of ALU, memory operations, branches, and all other instruction types.
Interprocess communication (IPC) · 20	The IPC calculated on a 4-wide superscalar processor using a bimodal branch predictor. We multiply this value by 20 to normalize it with other metrics.
Branch misprediction ratio · 100	Bimodal branch predictor miss ratio, which we multiply by 100 to normalize it with other metrics.

controller area network (CAN) remote data request benchmark (canrdr) takes a stream of three CAN messages and simulates their transmission and decoding. This process is similar to many networking benchmarks such as the packet flow (pktflow) and packet check (pktcheck) benchmarks, which deal with IP packet transmission and decoding.

Within the networking suites, we implemented the packet flow and packet check benchmarks using four different packet sizes. As expected, as packet size increases, the cache size requirements also increase because the working set is much larger. The Open Shortest Path First benchmark (ospf) demonstrates differences between the first and second versions. Networking v2.0 has greater ALU activity because it removes some of the superfluous debugging checks turned on by default in the first version. Both the algorithm and working set are identical; the only change between versions is code checking the consistency of the Dijkstra algorithm, which must be the source of the difference. Although the test harnesses between the two versions differ significantly, because we only collected data within the main benchmark execution (after data setup and before tear down), it did not affect our results.

As Figure 2 shows, the networking suite of benchmarks requires the fewest overall function units to achieve 85 percent utilization. This implies that this suite's inherent ILP is lower than that of the others. The memory units still exhibit a decent amount of parallelism, suggesting that memory dependencies are not the main cause of the ILP deterioration. One limiting factor is the high dependence chains caused by ALU instructions. Although the ALU instruction distribution is not as high as other suites, as Figure 3 shows, some benchmarks in the networking suite are still at least 40 percent ALU instructions. Thus, only one of the Kiviat graphs for the networking benchmarks resides in a group exhibiting high IPC (route lookup in group 9 in Figure 4). These benchmarks are much more memory intensive, with some benchmarks having up to 50 percent memory instructions in their overall execution stream.

Characterization group 3 consists entirely of benchmarks from the consumer and digital entertainment (that is, consumer v2.0) suites. These benchmarks show a low percentage of branch instructions and a high prediction ratio. Therefore, the average IPC is high. Additionally, the graphs exhibit sensitivity to an increased block size, especially when targeting a 1 percent miss ratio. This implies many calculations with few decision branches in large arrays. The block size effects indicate that spatial locality is significant for large portions of the data. Group 7 is similar to group 3, except the benchmarks in this group have more branch instructions, implying tighter loops and more if-then clauses. Together, the consumer and digital entertainment groups encompass all of the consumer benchmarks, showing that the earlier version contains many highly similar workloads. The digital entertainment suite encompasses a new characterization group (group 8) that emphasizes less cache stress and more ALU instructions.

Each grouping exhibits unique characteristics, which illustrates diversity in and among the EEMBC suites. For example, group 6 contains networking benchmarks that stress only the cache by reducing complex calculations and benchmarking large array lookups and data movement. However, all groups share a high misprediction rate using a simple bimodal predictor. Of course, this might not be an issue for most embedded applications because branch predictors often are not included or needed because the code is optimized to be fast without them.

Characterization group 8 encompasses benchmarks with higher ALU activity than other characterization metrics. Most of the benchmarks in this category also exhibit somewhat lower IPC values than other groupings, yet the branch misprediction ratio is consistently low. The cache size requirements are relatively small compared to other groupings because the working set is smaller for these benchmarks. Therefore, the implicit ILP must be limited to other factors such as a high degree of dependent instructions and calculations.

Characterization group 4 consists of a single benchmark, ospf. This benchmark is similar to the route lookup benchmark in group 9, except that the latter is less memory intensive and exhibits a higher branch misprediction ratio. Group 9 benchmarks respond to an increasing block size, whereas ospf does not. This is because ospf spends a large amount of time traversing a graph to solve Dijkstra's algorithm, in which the working set is larger and not sequentially stored, whereas benchmarks such as group 9's route lookup spend more time in computation through sequential structures.

Characterization group 5 is similar to group 1, except that it exhibits slightly different cache behavior. In this grouping, the cache requirements differ for 1 and 0.1 percent miss ratio targets. This implies that only a few conflict misses cause most of the cache misses for benchmarks in this category. Targeting a 0.1 percent miss ratio when the block size is 32 bytes requires a larger cache to map these conflict misses into separate blocks, whereas in group 1 the cache requirements for both cache sizes are similar. However, as benchmarks like the viterbi decoder benchmark (viterb00) illustrate, this behavior is not identical for the larger block size. In addition to this behavior, group 5 benchmarks exhibit relatively fewer cache requirements than the other metrics.

Group 1 exhibits the most balance among metrics. Benchmarks in this classification provide an overall equal stress on the cache and balanced instruction distribution, with varying IPC results. Most of the benchmarks place high stress on the cache, particularly those in the networking suites. One benchmark that is balanced overall but still different in this category is the automotive suite's angle-to-time conversion (a2time) benchmark. The cache requirements are smaller than those of other benchmarks, and the branch misprediction ratio is very low, yet the IPC is not very high. Most benchmarks with this cache behavior trend toward category 8 or 9, with either an imbalance in the instruction distribution or a higher branch misprediction ratio. Yet, this benchmark does not exhibit a spike for either characteristic. The benchmark is essentially a series of sequential calculations building on one another and causing true dependencies between instructions to limit ILP. Also, there is a sequence of if statements not grouped as if-then-else clauses, yet each one updates the same variable, resulting in output dependencies that can cause lower IPC.

In future work, we could use other tools and methodologies[4,5] to generate different characteristics and compare the results. In particular, Hoste and Eeckhout examine the delta change of measurements at fixed intervals to expose time-varying behavior.[4]

Our work has focused on characterizing the benchmarks, but it would also be interesting and practical to apply this methodology to real application code for validation. We could then devise a scheme that would compare workloads between the benchmarks and the real code, thereby letting the developer select benchmarks that most closely resemble their applications.

Another interesting approach is exploring the generation of workloads with specific characteristics to evaluate a particular architecture. Some work on the subject exists,[12-14]

but there appears to be no solution for generating portable C code with specific characteristics. Using a known set of kernels with various data sets and composing them into a benchmark based on characteristics as well as their relevance to a specific market segment can enable the creation of more accurate application-representative workloads. EEMBC is pursuing such a model for a future version of the automotive benchmarks.

You might ask, ''Why go through all the effort? Why don't developers just use their own application code to benchmark?'' The answer is simple: developers cannot always run their code on all platforms without considerable porting effort, and, for large applications, huge time commitments. The benchmarks serve as proxies for larger workloads that are representative, portable, and quick to execute. Additionally, the value of an industry standard and test input consistency allow a more uniform comparison. Results for EEMBC benchmarks are available from any silicon provider on request, and some are available publicly at www.eembc.org. Using these results and armed with specific benchmarks' characteristics, users can choose those results they find useful and select suitable systems or methodologies to use when designing new products, without spending time and effort to port and run the code on a platform. MICRO

References

1. T.M. Conte and W.W. Hwu, ''Benchmark Characterization,'' *Computer,* vol. 24, no. 1, Jan. 1991, pp. 48-56.
2. A. Joshi et al., ''Distilling the Essence of Proprietary Workloads into Miniature Benchmarks,'' *ACM Trans. Architecture and Code Optimization* (TACO), vol. 5, no. 1, 2008, article 10.
3. T.S. Karkhanis and J.E. Smith, ''Automated Design of Application Specific Superscalar Processors: An Analytical Approach,'' *Proc. 34th Ann. Int'l Symp. Computer Architecture* (ISCA 07), ACM Press, 2007, pp. 402-411.
4. K. Hoste and L. Eeckhout, ''Microarchitecture-Independent Workload Characterization,'' *IEEE Micro,* vol. 27, no. 3, May 2007, pp. 63-72.
5. A. Joshi et al., ''Measuring Benchmark Similarity Using Inherent Program Characteristics,'' *IEEE Trans. Computers,* vol. 55, no. 6, June 2006, pp.769-782.
6. K. Hoste et al., ''Performance Prediction Based on Inherent Program Similarity,'' *Proc. 15th Int'l Conf. Parallel Architectures and Compilation Techniques* (PACT 06), ACM Press, 2006, pp. 114-122.
7. T.M. Conte, M.A. Hirsch, and W.-M.W Hwu, ''Combining Trace Sampling with Single Pass Methods for Efficient Cache Simulation,'' *IEEE Trans. Computers,* vol. 47, no. 6, June 1998, pp. 714-720.
8. P. Viana et al., ''A Table-Based Method for Single-Pass Cache Optimization,'' *Proc. 18th ACM Great Lakes Symp. VLSI,* ACM Press, 2008, pp. 71-76.
9. A. Silva et al., ''Cache-Analyzer: Design Space Evaluation of Configurable-Caches in a Single-Pass,'' *Proc. 18th IEEE/IFIP Int'l Workshop Rapid System Prototyping* (RSP 07), IEEE CS Press, 2007, pp. 3-9.
10. T.M. Conte, ''Architectural Resource Requirements of Contemporary Benchmarks: A Wish List,'' *Proc. 26th Hawaii Int'l Conf. System Sciences* (HICSS 93), IEEE CS Press, 1993, pp. 517-529.
11. K.W. Kolence and P.J. Kiviat, ''Software Unit Profiles & Kiviat Figures,'' *ACM SIGMETRICS Performance Evaluation Rev.,* vol. 2, no. 3, 1973, pp. 2-12.
12. R. Bell Jr. and L. John, ''Improved Automatic Test Case Synthesis for Performance Model Validation,'' *Proc. Int'l Conf. Supercomputing* (ICS 05), ACM Press, 2005, pp. 111-120.
13. Z. Kurmas et al., ''Synthesizing Representative I/O Workloads Using Iterative Distillation,'' *Proc. Int'l Symp. Modeling, Analysis, and Simulation of Computer and Telecommunication Systems* (MASCOTS 03), IEEE CS Press, 2003, pp. 6-15.
14. A.M. Joshi, ''Automated Microprocessor Stressmark Generation,'' *Proc. IEEE 14th Int'l Symp. High Performance Computer Architecture* (HPCA 08), IEEE CS Press, 2008, pp. 229-239.

Jason A. Poovey is a PhD student in the College of Computing at the Georgia Institute of Technology. His research interests include performance modeling, workload characterization, and manycore/multi-

core technology. Poovey has a master's degree in computer engineering from North Carolina State University.

Markus Levy is the president and founder of the Embedded Microprocessor Benchmark Consortium (EEMBC), and founder of the Multicore Association. His research interests include microprocessor technology, processor benchmarking, and multicore programming. He received several patents while at Intel for his ideas related to flash memory architecture and usage as a disk drive alternative. Levy has a BS in electrical engineering from San Francisco State University.

Shay Gal-On is the director of software engineering at the Embedded Microprocessor Benchmark Consortium and leader of the EEMBC Technology Center. His interests include performance analysis, software security, and compilers. Gal-On received his degree from the Technion in Israel, following which he worked on compilers and performance tools for Intel, Improv Systems, PMC, and now EEMBC. Contact Gal-On at shay@eembc.org.

Thomas M. Conte is a professor in the College of Computing at the Georgia Institute of Technology. His research is in the areas of manycore/multicore architectures, microprocessor architectures, compiler code generation, architectural performance evaluation, and embedded computer systems. Conte has a PhD in electrical engineering from the University of Illinois at Urbana Champaign. He is associate editor of *ACM Transactions on Embedded Computer Systems, ACM Transactions on Architecture and Compiler Optimization, Computer*, and *IEEE Micro*.

Direct questions and comments about this article to Jason Poovey, College of Computing, Georgia Inst. of Technology, 266 Ferst Dr., KACB 2334, Atlanta, GA 30332; japoovey@gatech.edu.