Code Coverage and Input Variability: Effects on Architecture and Compiler Research

Hillery C. Hunter and Wen-mei W. Hwu
Center for Reliable and High-Performance Computing
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
{hhunter, hwu}@crhc.uiuc.edu

ABSTRACT

Meaningful application benchmarking is crucial to processor design space exploration and compiler development. Many recent studies in the embedded processor domain have used reference telecommunications C programs which were originally intended to verify product compliance with a standard. In this paper, we demonstrate that telecommunications reference applications include a significant amount of superfluous code that skews a wide variety of experiments related to code size. A categorization of extra code is presented, and it is also demonstrated that benchmark inputs do not test broad usage patterns that might exist in a real system. It is shown that care must be taken in future studies to properly construct benchmarks which match the intended purpose of the target system and to provide inputs that more thoroughly exercise paths through telecommunications applications.

Categories and Subject Descriptors

B.8.2 [**Performance Analysis and Design Aids**]; B.1.4 [**Microprogram Design Aids**]: Languages and Compilers—*Optimization*; C.4 [**Performance of Systems**]

General Terms

Measurement, Performance, Design, Experimentation

Keywords

Benchmarks, DSP, Telecommunications, Architecture, Compiler, Code Coverage

1. INTRODUCTION

Academic interest in processor architecture and compilation for embedded systems is rising as consumer demands for wireless and handheld device functionality and performance continue to grow. The general-purpose architecture and compiler communities have long had SPEC [1], a recognized set of codes suited not only for performance comparison of marketed processors, but also for processor design. At the application level, the embedded community does not have an equivalent basis for evaluation of architectural trade-offs and compiler optimizations. EEMBC [2] and BDTi [3] have standardized DSP and embedded controller kernels for performance measurement. For application benchmarking, however, the community has turned to *reference* programs which were originally intended for verification of standard functionality. These programs were not originally intended for either performance or design evaluation. This paper explores properties of reference applications and their inputs for embedded and DSP benchmarking, and presents results which demonstrate that these may introduce inaccuracies which cloud experimental outcomes.

1.1 Kernel and reference application properties

Initial literature in the area of DSP processor design presented results based on the use of program kernels as benchmarks. These commonly included routines for the *Fast Fourier Transform* (FFT), the *Discrete Cosine Transform* (DCT), and filters. Such routines are essentially high iteration count loops that exhibit large amounts of parallelism. Kernels continue to be useful for tuning specialized processor performance, but are not representative of the code a higher-end embedded/DSP processor must execute, namely a complete application. As discussed in [4], full applications combine kernel loops with sections of control code and serial operations, and exhibit much less overall parallelism than kernels in isolation. For the six full applications and six kernels examined in [4], the kernels had an average instruction-level parallelism (ILP) of 3.48, while that of the full applications was only 2.11. The majority of kernel instructions were shown to originate from loops comprised of an operand fetch→address update→multiply-accumulate sequence, but significant amounts of integer arithmetic, logic, and control operations were indeed seen in applications. Because of these differing characteristics, an understanding of the interactions of kernel routines in the framework of a complete application is crucial to accurate development and analysis in this domain, particularly for the design of compilable architectures.

Unfortunately, the set of complete media and telecommunications applications available for study in C source code is limited for two primary reasons: (1) patent restrictions and

CASES 2002, October 8–11, 2002, Grenoble, France.

the proprietary nature of many commercial algorithmic implementations often prevent the release of product-quality codes into the public domain; and (2) historically, signal processing systems combined a MAC-centric DSP core with a controller or host processor, meaning that kernel computation and control code were split across two processors, so tuned industry implementations of unified control and kernel code were rare.

One of the first benchmark suites to include full media and telecommunications applications was MediaBench [5], a suite of programs compiled and made available to the academic community by the University of California at Los Angeles in response to a growing need of complete applications for design evaluation. Following MediaBench were the University of California at Berkeley's *Berkeley Multimedia Workload* [6], EEMBC's licensed suites, and a proposed public domain alternative to EEMBC, the University of Michigan's MiBench [7]. Though they term their programs "applications," the EEMBC and MiBench suites are primarily kernels. These are meant to test the ability of an architecture and compiler to handle code segments which are pieces of larger applications such as modem standards or cellular phone codecs. For example, *fbital00* from EEMBC and the FFT kernel provided in both EEMBC and MiBench are key procedures for *ADSL* (Asymmetric Digital Subscriber Line) modems.

Applications included in benchmark suites generally originate from standards bodies such as the ITU [8] and ETSI [9] as *reference* codes, meaning that their functionality exactly meets the specifications of a given standard. The intention of the ITU, ETSI, and other standards bodies is to provide C code against which products may be verified. Until recently, product code was written by hand, so this simply meant providing C functions which demonstrated the underlying algorithms, and input/output pairs for checking assembly programs. As such, reference codes are valid even if inflated by case handling for optional functionalities and slowed by factors such as emulation of common DSP features (*e.g.* multiply-accumulate and saturating arithmetic). Subsequent sections examine and categorize this code inflation in reference telecommunications programs, show the pitfalls these properties expose to design and optimization experiments, and describe the invariability of reference application inputs.

2. EXPERIMENTAL ENVIRONMENT

2.1 Benchmarks

Table 1 lists the telecommunications applications and kernels used for this study. The eight applications from MediaBench and the ETSI are reference implementations of the standards they represent, and have been used in previous embedded and DSP studies, including [10], [11], [12], [13], and [14]. The five kernels comprise the EEMBC telecommunications suite used for performance benchmarking of embedded and DSP processors.

2.2 Code generation

Code generation and performance results presented in this paper were obtained using the IMPACT compiler framework [17]. Discussions will refer to two types of code: *CLO* (Classically-optimized) *code*, to which only traditional compiler optimizations have been applied; and *ILPO code*, which has been aggressively control transformed to enhance instruction-level parallelism. ILPO code has been formed into superblocks [18] and hyperblocks [19] (single entry, multiple exit, predicated regions). Although full predication has not yet been implemented on a DSP processor, it is assumed in this work that a predication mechanism such as that proposed in [20] could be used. The IMPACT compiler also uses intrinsic operations to represent DSP instructions (*e.g.* saturating arithmetic) specified in the ETSI's *g724* benchmarks.

3. CODE COVERAGE

The first motivation for analyzing the suitability of current benchmarks for embedded architecture research was the observation that a significant amount of telecommunications application code was not being touched during execution of reference inputs. Code usage was tallied at the control block level (a single entry, multiple exit region—a basic block, hyperblock, or superblock), and for the eight applications listed in Table 1, it was found that less than half the operations (43.9% of CLO code and 48.7% of ILPO) belonged to control blocks used during program execution. This low code utilization implies that either the applications contain superfluous code, or their inputs do not exercise many of the programs' control paths. Reference applications therefore appear not to meet the key embedded requirement that programs be tightly coded to minimize memory size, power, and cost.

3.1 Categorization of unused code

To understand the low instruction coverage observed in reference codes, untouched functions of the eight applications were examined by hand. Profile information assigned by the compiler was used to mark functions which were not executed during a sample input run, and each of these functions was then placed into one of four categories: Extra, Opposite End, Legitimate, or Necessary. These categories are defined in Table 2, and the results of this categorization are shown in Figure 1.

Functions categorized as Extra do not have any call sites in the benchmarks, and would not be present in an actual application implementation. Among the Extra functions found in this application set were procedures which printed help and version information (in the *gsm*s); utility functions for converting between various formats (*g721*s); functions for related, but separate, standards (*g721*s); and arithmetic emulation functions either not used or already in-lined by the programmer (*gsm*s).

Opposite End functions are an artifact of the bundling of encoder/decoder pairs in reference programs, and were found in all applications. Reference programs are generally packaged for distribution as a single bundle which contains both decoder and encoder functionality. Benchmark suites, however, define codec ends as separate applications, and differences in computational complexity mean that results are almost always presented for each end application separately. In the IMPACT framework, each application is separated into its own directory with source code, input and output sets, and administrative information which specifies benchmark set-up and execution parameters. While codec sides may share some functions, the clear benchmark partitioning in the IMPACT environment made it apparent that care should be taken when each end of a codec is used as a separate application, because the standard distribution of

Table 1: Telecommunications codes studied.

Applications	Source	Description
adpcmdec *adpcmenc*	MediaBench	Intel/DVI ADPCM codec
gsmdec *gsmenc*	MediaBench	Lossy sound compression according to the GSM 6.10 RPE-LTE standard [15]
g721dec *g721enc*	MediaBench	Voice compression according to the CCITT G.721 standard
g724dec *g724enc*	ETSI	GSM 06.60 EFR speech transcoding [16], state-of-the-art digital cellular communications
Kernels		
autcor00	EEMBC	Autocorrelation: Code-Excited Linear Prediction (CELP) filter transfer function matching
fbital00	EEMBC	Bit allocation: data distribution into ADSL frequency bins
fft00	EEMBC	Fast Fourier Transform: 256-point complex decimation in time algorithm
viterb00	EEMBC	Viterbi decoder: embedded IS-136 channel coding
conven00	EEMBC	Convolutional encoder: embedded V.xx modem output data stream encoding to enable error detection/correction

Table 2: Code coverage categories.

	Name	Description
Superfluous	Extra	No call sites in the benchmark
	Opposite End	Functions used only on the opposite codec side
	Legitimate	Needed for support of options not included in the definition of benchmark functionality
Required	Other Untouched	Operations not used during a sample benchmark run, but residing in touched functions
	Necessary	Non-superfluous functions; necessary for handling legitimate input possibilities or corner cases possible for the defined benchmark suite options
	Touched Code	Used during benchmark execution

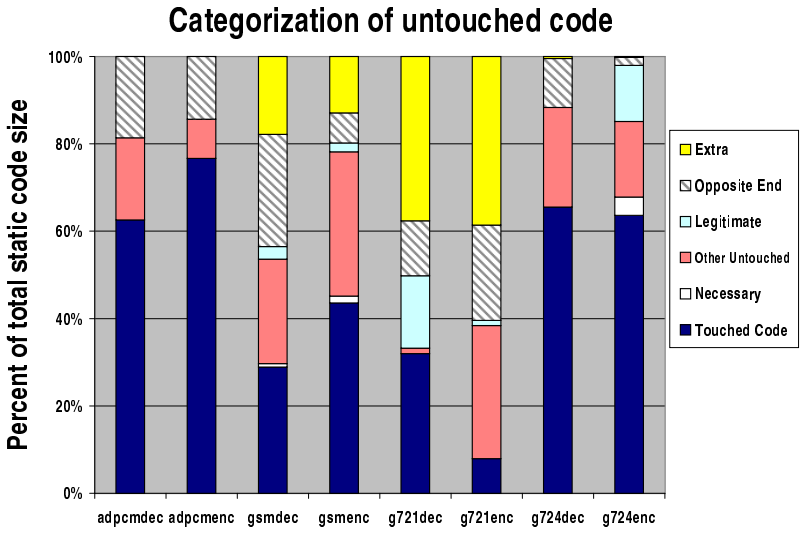

Figure 1: Categorization of untouched telecommunications reference code.

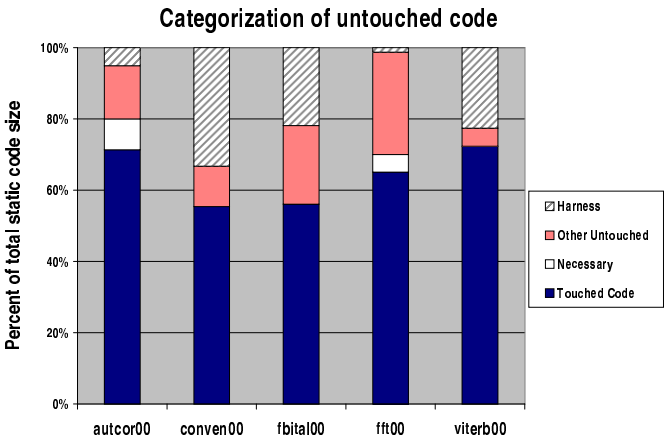

Figure 2: Categorization of untouched EEMBC benchmark code.

combined code results in significant waste. Opposite End functions were found to account for as much as 26% of original static code size (*gsmdec*), and their removal resulted in noticeable speedups in application compilation time. A compiler might be able to skip optimization and code generation for Opposite End and Extra functions by performing call site analysis and marking functions which are not present in the application call graph.

In contrast to Extra and Opposite End functions, which are simply out of place in programs used for processor design and compiler analysis, Legitimate functions correctly perform algorithmic calculations, but result from the wide set of profiles and operating conditions regulated by telecommunications standards. Embedded and DSP devices are generally intended to handle particular subsets of standard functionality, so it is acceptable that a benchmark is defined to handle a particular input type or profile. However, care must be taken then to strip the benchmark of legitimate code which is not related to the functionality being tested. In the studied applications, Legitimate functions primarily handle different input encoding types (*g721*s and *gsm*s), and various frame types (*g724*s). Each Legitimate function is a valid part of the reference code, but not part of the benchmark functionality as defined by its suite, *e.g.* specification of *g721* as using linear coding, as opposed to A- or μ-law coding. The Legitimate functions to handle A- and μ-law coding were therefore superfluous to the *g721* code when used as a benchmark.

Necessary functions, combined with Other Untouched operations and Touched functions, comprise *required* benchmark code. Necessary functions are those which were not exercised for the sample input, but are essential to correct coding of the standard and operation of the benchmark. They are not superfluous, and for the experiments in later sections, were not removed from the applications' code base. Other Untouched is an operation-level categorization, which encompasses all operations from control blocks not issued during the sample application run, but not members of a superfluous function. The Touched category in Figure 1 indicates the percentage of total static operations which were part of control blocks touched during an application's execution. On average, 47.7% of static operations were used. The untouched code in EEMBC telecommunications kernels was also examined, and it was found that the vast majority resulted from harness set-up in the IMPACT environment. Figure 2 shows the distribution of Harness, Other Untouched, Necessary, and Touched code in the EEMBC kernels. Designed for industrial processor evaluation, the EEMBC kernels are carefully written, and the kernels themselves were not found to contain the superfluous code present in the reference applications studied. Rather, C utilitity procedures used to supply a kernel with input values and check its output (the "harness") were originally replicated for each kernel. Individual kernels do not require use of all harness functions, thus leaving some harness functions untouched for each benchmark. Removing unnecessary harness code meant changing the harness to be specific to each benchmark, *i.e.* taking into account kernel-specific use of utility procedures and other functions for receiving inputs and creating and checking output values. This left an average of 14% untouched operations, a value significantly lower than for the untouched reference application code.

3.2 Consequences of low code coverage

The low code coverage and corresponding causes discussed in the previous section make it clear that reference applications currently used for embedded architecture research are not suited for studies which seek to measure absolute code size. This section will demonstrate that even fairly innocuous experiments may be significantly skewed by the properties of current telecommunications benchmarks. Experiments in this section were conducted using a relatively generic eight-wide VLIW machine model loosely corresponding to a Texas Instruments C6x series processor. Results are not closely tied to the machine model, and the same trends were observed when other target architectures were used for code scheduling.

As seen in Figure 3, hyperblock formation and predicate optimization can realize significant speedups for reference media and telecommunications codes. On average, optimizations achieved an execution speedup of 1.92 (ratio of ILPO cycle time to CLO cycle time). The penalty of many such performance-enhancing optimizations is evaluated in terms of incurred code growth, and if appropriate, resultant cache degradation. One side effect of the amount of superfluous code in reference benchmarks is a dilution of these apparent code size expansion penalties. Shown in Figure 4 are two sets of code growth results for ILPO code: (a) relative to total original benchmark code (*Original Code*); and (b) calculated using applications stripped of the superfluous code described in Section 3.1 (*Reduced Code*). The difference between the original results and the reduced result is 41.3%, which is a significant change. On the one hand, this indicates that the compiler has correctly targetted its optimizations at the frequently traversed code paths, but it also results in under-representation of the code size penalty of these optimizations.

The original results under-represent the code size penalty of hyperblock optimization by an average of 41.3%, which is a significant misestimation.

The next two sets of results focus on compiler techniques which have become common-place as DSP and embedded architectures have evolved to meet the performance demands of new applications. First, as the issue width of embedded processors increases, loop unrolling is often necessary to expose sufficient ILP to utilize the processor's functional units.

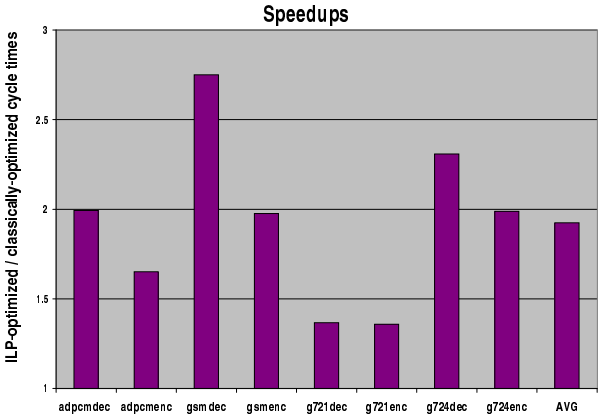

Figure 3: Speedup of ILP-optimized over traditionally-optimized code.

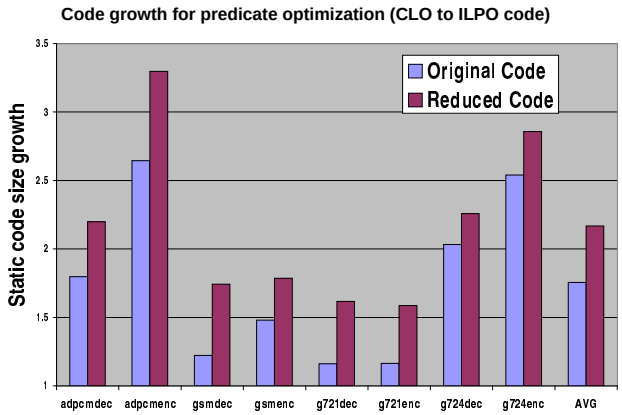

Figure 4: Under-estimation of code expansion penalties for ILP optimization.

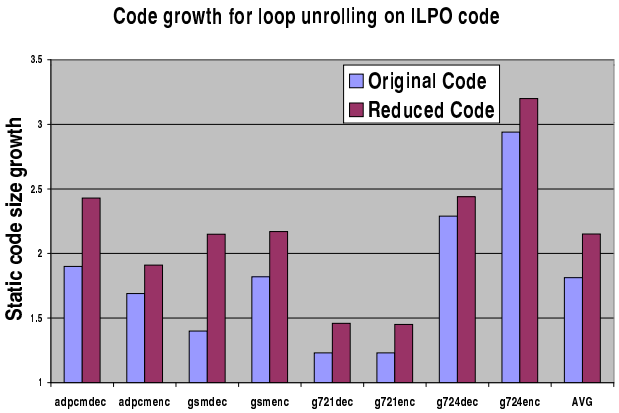

Figure 5: Code growth for loop unrolling optimization.

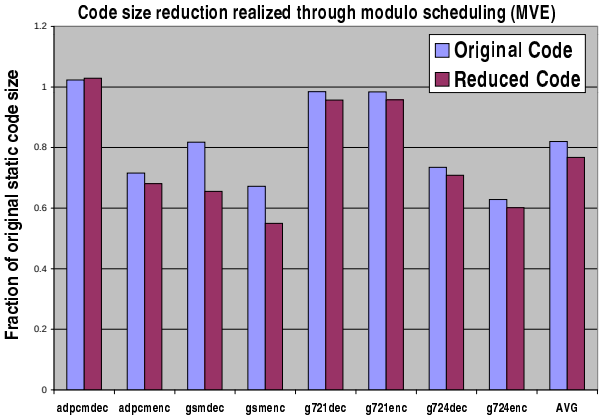

Figure 6: Code size reduction from modulo scheduling.

This once again enhances performance, but results in code size expansion which is of great concern for DSP and embedded processors with limited memory budgets. Figure 5 shows that the penalty of loop unrolling may be misunderstood when viewed only in comparison to total application code size. The average difference between results reported for original applications and pared-down benchmarks is 34%. Essentially, the larger (superfluous) instruction base for the original benchmarks provides a greater denominator for the code size expansion results, and causes the code size penalty of loop unrolling to be under-estimated.

Lastly, a technique which commonly realizes significant static code size reduction while maintaining high function unit untilization for telecommunications codes is *modulo scheduling* [21], a form of software pipelining. Many recent DSP architecture/compiler papers have been written on software pipelining techniques, but Figure 6 shows that yet again, current application benchmarks may skew experimental results. The programs were modulo scheduled with modulo variable expansion (MVE) and their static code size was compared to versions compiled with aggressive loop unrolling. Bars in the figure depict the ratio of unrolled operation count to that with modulo scheduling, for both the original applications and their corresponding reduced versions. For all applications except *adpcmdec*, which suffered due to modulo variable expansion, the code size reduction achieved via modulo scheduling was greater for the reduced application than for the original program. The presence of superfluous code in the unedited applications caused the benefit of modulo scheduling to be *underestimated* with respect to the actual benefit achieved on the benchmarks containing only required functions. The average difference between these two ratios was 0.053, which is a meaningful boost to the benefit of the technique.

Unguarded use of telecommunications reference codes for benchmarking may thus cause *either* inflation or deflation of results. Differences between outcomes for original applications and their reduced counterparts were as high as 41%, a value which is unacceptable, and indicates that fair judgement of certain instruction set architecture (ISA) and compiler features is very difficult if standard reference codes are used for evaluation and design.

4. INVARIABILITY OF INPUT SETS

Care must also be taken when specifying inputs for telecommunications applications because these inputs will be used by a profiling compiler to define the application's intended usage patterns. This section evaluates the effectiveness of current benchmark suite inputs in testing varied paths through application code. Code coverage at the instruction and control block level is examined, and it is shown that additional inputs often introduce very little variation in touched code or control flow patterns.

For the following experiments, code in superfluous categories (Extra, Other End, and Legitimate) was removed from the telecommunications applications. ILP-optimized versions of the benchmarks were exercised with three input sets provided either by a standards body or packaged benchmark suite. The inputs used for each benchmark are listed in Table 3. EEMBC provides a fourth input for *viterb00*, but it was not used in these experiments.

When measured by touched operation counts, the average reduced application code coverage for a single input

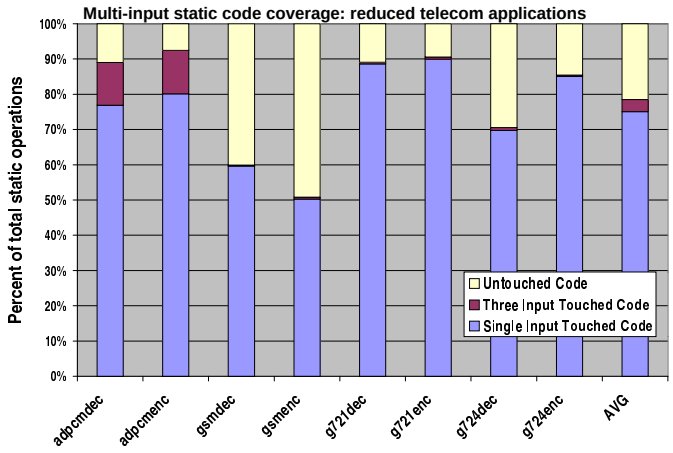

Figure 8: Percentage of static code touched: telecom applications.

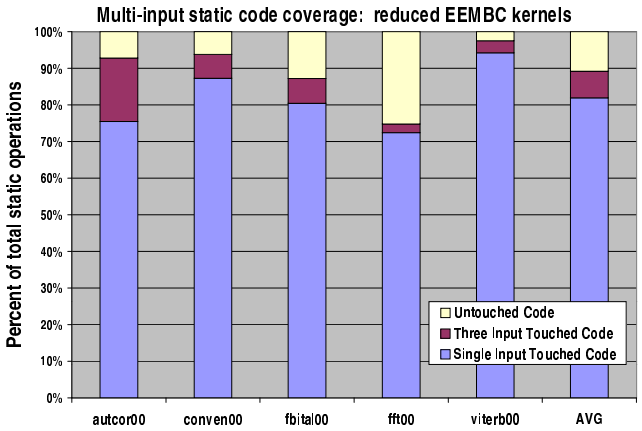

Figure 9: Percentage of static code touched: EEMBC telecom kernels.

(75.0%) was found to be significantly better than the original Touched proportion from Figure 1 (47.7%), but still leaves room for improvement. Figure 8 shows that an additional two inputs increase average instruction coverage by just 3.4% over a single input. Only the additional *adpcm* inputs contribute significantly to application code coverage.

Multiple-input code coverage of EEMBC kernels was also measured, and it was found that 89.2% of the reduced kernel operations were members of touched control blocks when three inputs were used. Not only is this value an improvement over the kernel coverage before customizing the code harnesses (Figure 2), but it is also greater than the coverage of the eight reduced applications in Figure 8. The breakdown of kernel static operation coverage is pictured in Figure 9. Hyperblock formation, loop peeling, and other ILP optimizations result in side exits and control blocks designed to handle error conditions and block traversals not seen for the profile input used to guide optimization. Because the EEMBC kernels are carefully written and their inputs are chosen to effectively evaluate product performance, the 10% of operations belonging to blocks not touched dur-

Table 3: Input sets.

Applications	Input1	Input2	Input3
adpcm{*dec*|*enc*}	MediaBench default: clinton.pcm	MiBench training input: small.pcm	MiBench evaluation input: large.pcm
gsm{*dec*|*enc*}	MediaBench default: clinton.pcm	MiBench training input: small.au	MiBench evaluation input: large.au
g721{*dec*|*enc*}	MediaBench default: clinton.pcm	MediaBench alternate input: S_16_44.pcm	[none found as part of a suite]
g724{*dec*|*enc*}	285 frames of a synthetic harmonic signal; pitch delay varies slowly from 144 to 18.5 samples	224 frames of male speech, active speech level: -18.7dBov	363 frames of female speech with ambient noise, active speech level: -25.0 dBov
Kernels			
autcor00	xpulset.dat	xsinet.dat	sxpeecht.dat
conven00	xk5r2d.dat	xk4r2d.dat	xk3r2d.dat
fbital00	xtypSNRt.dat	xstepSNRt.dat	xpentSNRt.dat
fft00	tpulset256.dat	xspn256t.dat	xsine256t.dat
viterb00	gett.dat	togglet.dat	onest.dat

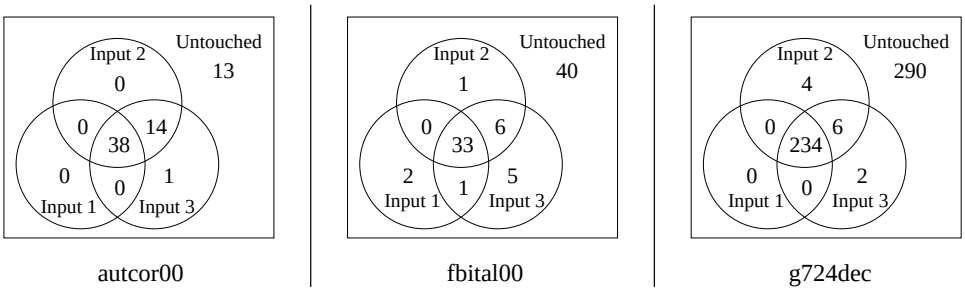

Figure 7: Venn diagrams of control block usage.

ing EEMBC kernel execution may be viewed as an average lower bound on ILPO touched code.

The control blocks used for each input's execution were examined, and Figure 7 shows Venn diagrams derived for two EEMBC kernels (*autcor00* and *fbital00*) and one ETSI application, *g724dec*. The values shown within each circle represent control block counts for individual inputs, and the total untouched cb count is given outside the input regions. There is significant overlap between each input's control block (cb) coverage, and it should be noted that in many cases, additional cb coverage results from alternate peeled loop or hyperblock exits, which do not represent substantive differences in application control flow.

While each input provided little additional code coverage, it was also found that the relative control block execution frequencies for each application input were very similar. This indicates that not only did each input exercise very little additional code, but it resulted in traversal of paths nearly identical to those for other inputs. To demonstrate this, Figures 10 and 11 show comparisons of control block contributions to total cycle count for two applications whose additional inputs resulted in very little change in touched code: *g721enc* and *gsmdec*. Control blocks are listed along the x-axis, sorted by greatest percentage contribution to program execution, *i.e.* bar 1 is the cb which accounts for the greatest percentage of the application's execution time. Each bar represents a difference in a control block's percentage contribution to total application cycle count with respect to that of Input1. All control blocks which account for at least 0.5% of the application's execution time are shown on the graphs. A clear correlation between relative cb importance across input sets is seen in these figures, since the maximum difference between any single cb's contribution to overall application execution time is 1.2%, and the differences for *gsmdec* are all under 0.005%.

The previous figures have examined invariability of input sets from three perspectives: covered instructions, used cb counts, and control block traversal. The lack of variability in the *gsm*, *g721*, and *g724dec* application inputs is most pronounced, while the EEMBC kernel inputs consistently provide noticeable additional code coverage.

5. CONCLUSION

The use of reference applications for embedded processor and compiler research has become fairly common in the academic community. This study has shown that the presence of superfluous functions in these programs makes them unsuited for development and evaluation of techniques whose impact is measured in terms of absolute or relative code size. In addition, inputs were shown to lack sufficient variation in static instruction usage and program control path traversal. Higher percentages of code coverage, and thus higher experimental accuracy, are realizable if reference codes are edited. Benchmarks must be carefully designed so that they include all code relevant to a full application (more than that provided by kernels), but are yet limited to all meaningful code (less than that of reference programs) so that realistic analyses can be made. Embedded and processor architecture and compiler developers can be helped by knowledge of the

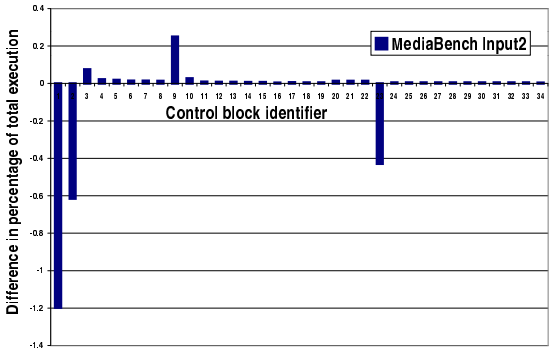

Figure 10: Percentage cb contribution to execution cycles; difference from Input1: g721enc.

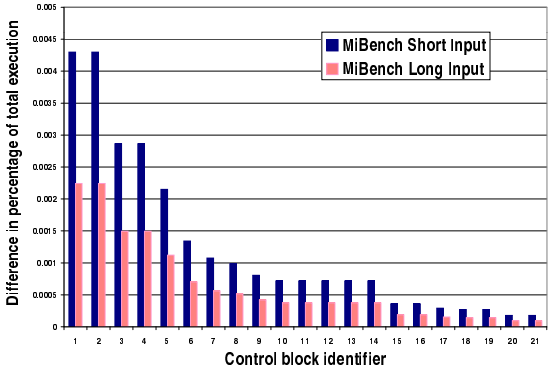

Figure 11: Percentage cb contribution to execution cycles; differences from Input1: gsmdec.

properties brought to light in this study and more carefully consider their impact when evaluating experimental results.

6. ACKNOWLEDGMENTS

This work was supported by the Semiconductor Research Corporation under the grant "Memory Efficient EPIC/VLIW Architecture (ID 785)." Hillery Hunter was supported by a fellowship from the National Science Foundation. The authors would like to thank Jaime Moreno and Matthew Merten for their contributions, and the anonymous reviewers for their comments.

7. REFERENCES

[1] Standard Performance Evaluation Corporation (SPEC). www.spec.org.

[2] Embedded Microprocessor Benchmark Consortium (EEMBC). www.eembc.org.

[3] Berkeley Design Technology, Inc., "Evaluating DSP processor performance." White paper, 2000.

[4] M. Saghir, P. Chow, and C. Lee, "Application-driven design of DSP architectures and compilers," in *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing 1994 (ICASSP '94)*, pp. 437–440, Apr. 1994.

[5] C. Lee, M. Potkonjak, and W. Mangione-Smith, "MediaBench: A tool for evaluating and synthesizing multimedia and communications systems," in *Proceedings of the 30th Annual International Symposium on Microarchitecture (MICRO-30)*, pp. 330–335, Dec. 1997.

[6] N. Slingerland and A. Smith, "Design and characterization of the Berkeley Multimedia Workload," Tech. Rep. CSD-00-1122, University of California at Berkeley, 2000.

[7] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Proceedings of the 4th Workshop on Workload Characterization*, pp. 1–12, Dec. 2001.

[8] International Telecommunication Union (ITU). www.itu.int.

[9] European Telecommunications Standards Institute (ETSI). www.etsi.org.

[10] J. Fritts, W. Wolf, and B. Liu, "Understanding multimedia application characteristics for designing programmable media processors," in *Proceedings of the SPIE Photonics West Media Processors Conference*, pp. 2–13, 1999.

[11] S. Debray, W. Evans, R. Muth, and B. De Sutter, "Compiler techniques for code compaction," *ACM Transactions on Programming Languages and Systems*, vol. 22, no. 2, pp. 378–415, 2000.

[12] C. Hughes, P. Kaul, S. Adve, R. Jain, C. Park, and J. Srinivasan, "Variability in the execution of multimedia applications and implications for architecture," in *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pp. 254–265, 2001.

[13] E. Witchel, S. Larsen, C. Ananian, and K. Asanovic, "Direct addressed caches for reduced power consumption," in *Proceedings of the 34th Annual*

International Symposium on Microarchitecture (MICRO-34), pp. 124–133, Dec. 2001.

[14] O. Unsal, I. Koren, C. Krishna, and C. Moritz, "The Minimax Cache: an energy-efficient framework for media processors," in *Proc. of the 8th IEEE Symposium on High-Performance Computer Architecture (HPCA-8)*, pp. 131–140, Feb. 2002.

[15] ETSI TC-SMG, "GSM full rate speech transcoding (GSM 06.10)," Tech. Rep. Version 3.2.0, Release 92, Phase 1, European Telecommunications Standards Institute, Feb. 1992.

[16] ETSI TC-SMG, "Digital cellular communications system; Enhanced Full Rate (EFR) speech transcoding (GSM 06.60)," Tech. Rep. ETS 300 726, European Telecommunications Standards Institute, Mar. 1997.

[17] W. Hwu, *et. al.*, "Compiler technology for future microprocessors," *Proceedings of the IEEE*, vol. 83, pp. 1625–1995, Dec. 1995.

[18] W. W. Hwu, *et. al.*, "The superblock: An effective technique for VLIW and superscalar compilation," *The Journal of Supercomputing*, vol. 7, pp. 229–248, Jan. 1993.

[19] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, R. A. Bringmann, and W. W. Hwu, "Effective compiler support for predicated execution using the hyperblock," in *Proceedings of the 25th Annual International Symposium on Microarchitecture (MICRO-25)*, pp. 45–54, Dec. 1992.

[20] J. Sias, H. Hunter, and W. Hwu, "Enhancing loop buffering of media and telecommunications applications using low-overhead predication," in *Proceedings of the 34th Annual International Symposium on Microarchitecture (MICRO-34)*, pp. 262–273, Dec. 2001.

[21] B. Rau, "Iterative modulo scheduling: An algorithm for software pipelining loops," in *Proceedings of the 27th Annual International Symposium on Microarchitecture (MICRO-27)*, pp. 63–74, Dec. 1994.