A Study on the Loop Behavior of Embedded Programs

Jason Villarreal, Roman Lysecky, Susan Cotterell, and Frank Vahid
Department of Computer Science and Engineering
University of California, Riverside

Technical Report UCR-CSE-01-03
December 2001

ABSTRACT

Software executing on a microprocessor contributes to much of the overall power and performance of an embedded system. A general rule-of-thumb for the behavior of both desktop and embedded systems has been that most execution time is spent in a small fraction of the software. We studied the behavior of 16 embedded system programs from the Powerstone benchmarks, with a focus specifically on those programs' loop behavior. We examined such behavior for a popular 32-bit embedded microprocessor (MIPS) as well as a popular 8-bit mmicroprocessor (8051).

Keywords

Embedded software, dynamic loop behavior, loop cache, loop analysis, hardware/software partitioning, architecture synthesis.

1. Introduction

A common aspect of numerous research efforts in low power and high performance embedded systems focus on the most frequently-executed software regions. Those regions may be translated into custom instructions, partitioned for execution on a coprocessor, compressed, or cached. A general rule-of-thumb is that software tends to spend most of its time in a small percentage of code. The desktop software community has utilized this rule to develop profile-guided compilers [5][16] that focus their optimization efforts on the most critical software regions. Even hardware-assisted runtime optimization has been proposed [1]. Most profile-guided efforts from the software community have focused on high performance. Recently, however, embedded system design automation has begun looking at the power savings as well [4].

The most critical software regions tend to exist within loops. Thus, previous researchers working in the desktop computing domain have investigated the dynamic behavior of loops. Kobayashi performed an early study of dynamic loop behavior for IBM System/370 applications, showing that more than half of a program's executed instructions lie within loops [12]. Several recent efforts focus on dynamic loop detection for use in speculative execution, in particular, on exposing more instruction-level or thread-level parallelism to a superscalar or multi-threaded processor (e.g., [18]).

Embedded software is generally thought to have different behavior than desktop applications. The software tends to be written in a leaner manner, and may spend more time in very small loops [14]. Furthermore, embedded microprocessors tend to focus on low power rather than just high performance, meaning their architectures do not support the large scale instruction-level parallelism of today's popular desktop processors, which in turn means that the compilers for embedded processors may emit code quite different than those for desktop processors. An analysis of MediaBench, a benchmark suite focusing on multimedia and communication applications was performed recently [3]. The results from this analysis focused on the instruction mix, branch prediction accuracy, cache hits, memory use, and integer bit utilization, but not on loops.

Motivated by the need for a better understanding of the loop behavior of embedded software, we decided to conduct a study on such loop behavior. We present the results of that study in this report.

2. Method

2.1 Benchmarks

For this study we sought to contrast the results for a popular 32-bit processor with those for a popular 8-bit processor. We used Motorola's PowerStone benchmark suite as our set of software applications [15]. Table 1 shows the benchmarks we used, a short description of each, and their code size in lines of C code excluding comments and whitespace.

There are several additional programs included in the PowerStone benchmarks. However, we excluded some due to their small size or small dynamic instruction count. Additionally, we did not include a few because they would not execute on one of our simulators, for reasons we are investigating. Initially, we were also considering investigating the loop behavior of MediaBench, but we chose PowerStone for these experiments because most benchmarks from the former do not apply to small embedded processors. These benchmarks can be viewed as either small embedded programs or computation kernels that might be found in larger embedded programs.

Each PowerStone benchmark comes with its own example input and expected output. For instance, *g3fax* contains sample fax data within the benchmark. In addition, each program has a main loop that has an iteration number that can be set to 1 or more. For our analysis, we set the iteration number to 1.

Table 1: Benchmark Description and Code Size.

Benchmark	Lines of C Code	Description
adpcm*	501	Voice Encoding
bcnt	90	Bit Manipulation
binary	67	Binary Insertion
blit	94	Graphics Application
brev	72	Shifting and Or Operations
compress*	943	Data Compression Program
crc	84	Cyclic Redundancy Check
des*	745	Data Encryption Standard
engine*	276	Engine Controller
fir*	173	FIR Filtering
g3fax	639	Group Three Fax Decode
jpeg*	540	JPEG Compression
matmul	42	Matrix Multiplication
summin	74	Handwriting Recognition
ucbqsort	209	U.C.B Quick Sort
v42*	553	Modem Encoding/Decoding

* MIPS benchmark only

2.2 LOOAN Tool

The benchmarks were compiled for the MIPS 32-bit microprocessor using LCC [7]. For the 8051 8-bit microcontroller, the Keil C compiler was used with the NOOVERLAY flag, which assures that data segments and code segments remain separate. We ran the MIPS programs on a MIPS simulator that we modified to emit assembly code with addresses, a map file, and an instruction trace. The map file simply provides a listing of the functions in the program along with their start and end addresses. The 8051 programs were run on an instruction set simulator also modified to output an instruction address trace. The map files for the 8051 assembly code were generated by the Keil compiler.

We implemented the loop analysis with a C++ program that represents the loop structure of a given MIPS or 8051 program. The program reads a benchmark's assembly file, map file, and instruction trace and creates a directed acyclic graph (DAG) representation in which the root of the DAG has children that correspond to all of the routines in the code, e.g., main, printf, etc. Each routine node has children nodes that correspond to that routine's loops, which are automatically numbered beginning with 1. Likewise, each loop node has children nodes that correspond to that loop's sub-loops. Finally, when a node (loop or routine) has a call to a function, a special function call node is created that links to the routine being called. This is done to enable us the keep track of statistics for both the individual links to function calls as well as statistics for all calls to the function.

After the DAG is created, the loop analysis program will parse the instruction trace and update each node with the required information. After we have processed the entire instruction trace, we calculate certain statistical data and output the information to a file. We will discuss these statistics later.

Collectively, we refer to this set of tools as *LOOAN* (LOOp ANalysis).

We chose the above approach over a binary instrumentation approach for several reasons. One was that we could easily update our analysis program to keep additional statistics. A second is because the above approach yields no change in program behavior. The disadvantages compared to instrumentation are the slower execution and the need to generate large trace files.

The MIPS simulator and the Keil compiler run under Windows NT. The other tools we created were run on a Pentium-based Linux workstation but were written in standard C++ which could easily be ported to other platforms.

2.3 Generating Loop Behavior Data using LOOAN

When using the *LOOAN* environment, to generate data for an 8051 program, we first compile it with the Keil compiler setting the NOOVERLAY flag and generate both the assembly file (in HEX format) and the map file (which is created by the Keil compiler during linking). Then, the compiled program is simulated using the 8051 instruction set simulator to generate a trace file. This usually takes less than two seconds for small programs. However, the trace file generated can be very large (the 8051 *summin* trace file was 256 MB). Finally, to generate the loop analysis data, the assembly file, map file, and trace file are run through our loop analysis tool.

In order to generate data for the MIPS processor, we first use LCC coupled with the modified MIPS simulator to generate an assembly file, a trace file, and a map file. These outputs are then used by the loop analysis tool to generate the loop analysis results. Executing the *jpeg* benchmark on the MIPS simulator took 49 seconds on a 400 MHz Pentium II processor and generated a 36 MB trace file.

3. MIPS Results

Figure 3 and Figure 4 present the loop analysis results for benchmarks run on the MIPS processor. In the figures, *Region* is the name of the loop, which begins with the name of the subroutine in which the loop is found. Loops are numbered in the static order they appear in the assembly code of that subroutine. A nested loop creates another level of numbering. Thus, a loop named *main.5* corresponds to the fifth loop encountered in the main routine of a program. A loop named *main.5.1* corresponds to *main.5*'s first sub-loop. For conciseness, we only list loops that contribute to at least 5% of the overall dynamic instruction count, thus you may notice gaps in the numbering of loops in the table. *Size* indicates the static size of each loop computed as the end address minus the start address plus 1.

We also show subroutines themselves in the table. They appear as a name without a loop number following them. The entire program is reflected by '.'.

We define a single *iteration* of a loop as a pass through the body of the loop followed by a jump to the loop beginning. We define an *execution* of a loop as the situation of entering the loop from outside the loop, during which the loop may iterate many times before it finally exits. A subroutine, on the other hand, always iterates exactly once during each execution. In the table, *dynamic instructions per iteration* indicates how many

Figure 1: Percentage of time spent in small loops.

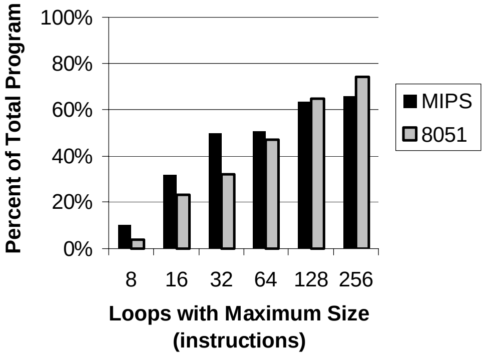
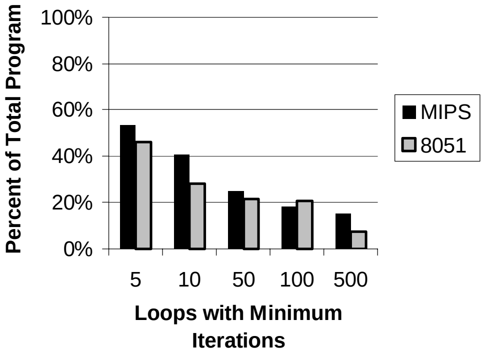


Figure 2: Percentage of time spent in highly-iterating loops.



instructions are executed for a single iteration of the loop. *Iterations per execution* indicates the number of iterations each time we enter the loop. *Number of executions* indicates the total executions of this loop or subroutine after a complete run of the benchmark. *Total dynamic instructions* indicates the total number of instructions executed by this loop during the complete run of the benchmark. Finally, *%* represents the percentage of total dynamic instructions that this loop or subroutine accounts for. For convenience of readability, we indent the % depending on the loop's nesting level. We sum % for each example. So the first % column represents time spent in subroutines, the second in first level loops, the third column in second level loops, etc.

The first observation we can make from the data gathered is that the time spent in loops by these programs, as seen at the bottom of the *%* data of each example, is large. Some of the programs spend over 90% of its time in loops. The average across all the examples is roughly 66%. The average is computed from the total percentage from each example. Thus, by not combining the raw numbers first, all examples are weighted equally. The number is actually about 70% if we include loops that contribute to less than 5% of the total dynamic instructions.

Another observation we can make is that in many examples, a significant percentage of time is spent in rather small loops. To illustrate this concept, Figure 1 plots the percentage of time spent in loops of size 8 or less, 16 or less, 32 or less, 64 or less, 128 or less, and 256 or less, averaged across all the examples. In obtaining the values for this plot, care was taken not to double-count nested loops. Nearly all time spent in loops (66% of total time) is spent in loops of size 256 or less. However, also note that most of this time (77% of it) is spent in loops of size 32 or less, accounting for 51% of the total time. In other words, half of the time is spent in what many would consider very small loops.

We also look at the percentage of time spent in highly iterating loops. Figure 2 shows the percentage of time spent in loops with at least 5, 10, 50, 100, and 500 iterations. 53% of the time is spent in loops that iterate at least 5 times. Notice that this is a significant drop from the 66% for all loops. This means that many loops iterate only once or just a few times. However, 41% of time is spent in loops that iterate at least 10 times.

4. 8051 Results

Figure 5 and Figure 6 shows the loop analysis data for the benchmarks run on the 8051. The data is presented in the same manner as the MIPS data.

From the loop analysis data, we can see that many of the 8051 benchmark applications spend over 90% of their execution time in loops, with an average across all the examples of roughly 77%. Furthermore, as seen in figure Figure 1, on average 74% of total time is spent in loops of maximum size 256. However, of this time 64% is spent in loops of size 64 or less, accounting for 47% of the total execution time. This indicates that approximately half of the programs execution time is spent within small loops.

We also look at the percentage of time spent in highly iterating loops. Figure 2 shows the percentage of time spent in loops with at least 5, 10, 50, 100, and 500 iterations. For the 8051 benchmarks, almost half of the time (46%) is spent in loops that iterate at least 5 times. Furthermore, 36% of all loops iterate at least 10 times, and account for 28% of total execution time. Another interesting observation is that almost all loops that iterate at least 50 times actually iterate for more than 100 times, and roughly one third of these loops iterate greater than 500 times.

5. Further Analysis

While some of the results for the 8051 are quite similar to those of the MIPS, there are certain aspects that mark some notable differences. As seen in Figure 1, most of the execution time for the 8051 programs was spent within loops of no greater than 256 instructions (74%). Compared with the MIPS applications, it is approximately 12% more of the total time. Additionally, the MIPS spent 50% of it time in loops of no greater than 32 instructions, while the 8051 only spends 32% in the loops of the same size. This difference can be accounted for by observing that in order to achieve the same task more code will be required on the 8051. This is mainly due to the fact that the 8051 is an 8-bit processor and lacks the ability to perform native 32-bit integer operations and native floating point operations. Thus, the size of the loops will contain more instructions then the equivalent MIPS code.

Furthermore, as seen in Figure 2, in general both MIPS and 8051 applications follow the same trend with regard to the number of iterations a loop executes. However, the 8051 applications have a large percentage of loops that execute greater then 100 times. As mentioned earlier, almost all loops that executed 50 iterations also executed 100 iterations. To determine the cause for this behavior we looked at which loops executed at least 100 iterations and determined that they mainly correspond to the 8051's startup code.

6. Conclusions

Studying the loop behavior of programs can yield many insights as to what architectural features and optimization techniques can be utilized in a system architecture. We presented the *LOOAN* environment for performing loop analysis and provided details of a study on the loop and subroutine behavior of a set of embedded programs.

7. References

[1] Bala, V., E. Duesterwald, and S. Banerjia. Dynamo: A Transparent Dynamic Optimization System. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), June 2000.

[2] Bellas, N.; Hajj, I.; Polychronopoulos, C.; Stamoulis, G. Energy and Performance Improvements in Microprocessor Design Using a Loop Cache. International Conference on Computer Design, pp. 378-383, 1999.

[3] Bishop, B., T.P. Kelliher, and M.J. Irwin. A Detailed Analysis of MediaBench. IEEE Workshop on Signal Processing Systems, pp.448-455, 1999.

[4] Chung, E.Y., L. Benini and G. De Micheli. Automatic Source Code Specialization for Energy Reduction. International Symposium on Low Power Electronics and Design, 2001.

[5] Diniz, P. and M. Rinard. Dynamic Feedback: An Effective Technique for Adaptive Computing. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), June 1997.

[6] Fisher, J. Customized Instruction Sets for Embedded Processors. Design Automation Conf. (DAC), 1999.

[7] Fraser, Christopher. A Retargetable C Compiler: Design and Implementation. Addison-Wesley, January 1995.

[8] Gajski, D., F. Vahid, S. Narayan and J. Gong. Specification and Design of Embedded Systems. Prentice Hall, 1994.

[9] Govindarajan, S.C., G. Ramaswamy, and M. Mehendale. Area and Power Reduction of Embedded DSP Systems using Instruction Compression and Re-configurable Encoding. International Conference on Computer Aided Design, 2001.

[10] Henkel, J. A Low Power Hardware/Software Partitioning Approach for Core-Based Embedded Systems. Design Automation Conference, pp. 122-127, 1999.

[11] Ishihara, T., H. Yasuura. A Power Reduction Technique with Object Code Merging for Application Specific Embedded Processors. Design Automation and Test in Europe, March 2000.

[12] Kobayashi, M. Dynamic Characteristics of Loops. IEEE Transactions on Computers, vol C-33 (no. 2), Feb 1984, pp. 125-132.

[13] Lakshminarayana, G., A. Raghunathan, K.S. Khouri, N.K.Jha, and S. Dey. Common-Case Computation: A High-Level Technique for Power and Performance Optimization. Design Automation Conference (DAC), pp. 1-5, 1999.

[14] Lee, L.H., B. Moyer and J. Arends. Instruction Fetch Energy Reduction Using Loop Caches For Embedded Applications with Small Tight Loops. International Symposium on Low-Power Electronics and Design, San Diego CA, 1999, pp. 267-269.

[15] Malik, A.; Moyer B.; Cermak D. A Lower power unified cache architecture providing power and performance flexibility. International Symposium on Low Power Electronics and Design. June. 2000.

[16] Pettis, K. and R.C. Hansen. Profile Guided Code Positioning. ACM SIGPLAN 90 Conference on Programming Language Design and Implementation (PLDI), June 1990.

[17] Semiconductor Industry Association. International Technology Roadmap for Semiconductors: 1999 edition. Austin, TX: International SEMATECH, 1999.

[18] Tubella, J and A. Gonzalez. Control Speculation in Multithreaded Processors through Dynamic Loop Detection. High Performance Computer Architecture, Las Vegas, 1998.

[19] Vahid, F. and A. Gordon-Ross. A Self-Optimizing Microprocessor Using a Loop Table for Low Power, International Symposium on Low Power Electronics and Design, 2001.

[20] Virtual Socket Interface Association, Architecture Document, 1997.

Figure 3: Loop statistics for MIPS (*adpcm*, *blit*, *compress*, *crc*, *des*, *engine*, *fir*, and *g3fax*).

Region	Start	End	Static Size	Dynamic Instrs per Iteration				Iter per Exec.				Total Execs	Total Dynamic Instrs	%		
				avg	min	max	stddev	avg	min	max	stddev					
adpcm																
.	2	1911	1910	63891	63891	63891	0	1	1	1	0	1	63891	100%		
..decode	1236	1489	254	1237.38	1237	1238	0.49	1	1	1	0	50	29800	47%		
..upzero	1710	1766	57	122.5	93	152	29.5	1	1	1	0	100	12250	19%		
..decode.1	1414	1435	22	220	220	220	0	10	10	10	0	50	11000		17%	
..filtez	1571	1600	30	82	82	82	0	1	1	1	0	100	8200	13%		
..decode.2	1459	1474	16	160	160	160	0	10	10	10	0	50	8000		13%	
..upzero.2	1730	1752	23	132	132	132	0	6	6	6	0	50	6600		10%	
..filtez.1	1583	1595	13	65	65	65	0	5	5	5	0	100	6500		10%	
..uppol2	1767	1806	40	36	36	36	0	1	1	1	0	100	3600	6%		
..upzero.1	1716	1727	12	72	72	72	0	6	6	6	0	50	3600		6%	
														84%	56%	
blit																
.	2	1044	1043	22845	22845	22845	0	1	1	1	0	1	22845	100%		
..blit	867	1016	150	11062.5	11062	11063	0.5	1	1	1	0	2	22125	97%		
..blit.1	906	916	11	11003	11003	11003	0	1001	1001	1001	0	1	11003		48%	
..blit.2	945	955	11	11003	11003	11003	0	1001	1001	1001	0	1	11003		48%	
														97%	96%	
compress																
.	2	1869	1868	138573	138573	138573	0	1	1	1	0	1	138573	100%		
..getcode	1620	1748	129	85.3	52	332	84.02	1	1	1	0	465	39665	29%		
..compress	1162	1361	200	71810	71810	71810	0	1	1	1	0	1	35863	26%		
..compress.2	1244	1327	84	64882	64882	64882	0	800	800	800	0	1	35738		26%	
..output	1362	1503	142	63.06	28	157	31.5	1	1	1	0	465	29323	21%		
..decompress	1504	1619	116	65677	65677	65677	0	1	1	1	0	1	26012	19%		
..decompress.2	1543	1610	68	63805	63805	63805	0	464	464	464	0	1	24436		18%	
..getcode.1	1668	1694	27	236	17	254	31	10	2	11	1	59	13949		10%	
														94%	53%	
crc																
.	2	1061	1060	37650	37650	37650	0	1	1	1	0	1	37650	100%		
..icrc1	867	898	32	111	95	127	6	1	1	1	0	256	28416	75%		
..icrc1.1	876	892	17	96	80	112	6	8	8	8	0	256	24576		65%	
..icrc	899	1030	132	18484	1095	35873	17389	1	1	1	0	2	8552	23%		
..icrc.1	923	947	25	34820	34820	34820	0	257	257	257	0	1	6404		17%	
														98%	82%	
des																
.	2	1530	1529	122214	122214	122214	0	1	1	1	0	1	122214	100%		
..des_set_key	867	1072	206	1456	1456	1456	0	1	1	1	0	47	68432	56%		
..des_set_key.1	974	1063	90	1340	1340	1340	0	16	16	16	0	47	62980		52%	
..des_encrypt	1176	1476	301	913	913	913	0	1	1	1	0	47	42911	35%		
..des_encrypt.1	1225	1326	102	816	816	816	0	8	8	8	0	47	38352		31%	
														91%	83%	
engine																
.	2	1109	1108	410607	410607	410607	0	1	1	1	0	1	410607	100%		
..interpolate	932	1045	114	138	68	199	35	1	1	1	0	1742	240876	59%		
..engine	867	931	65	409812	409812	409812	0	1	1	1	0	1	71384	17%		
..engine.1	874	924	51	409798	409798	409798	0	26	26	26	0	1	71370		17%	
..engine.1.1	877	910	34	15744	11063	18950	2263	68	68	68	0	26	70928			17%
..interpolate.2	973	980	8	33	5	61	16	4	1	8	2	1742	57358		14%	
..interpolate.1	935	942	8	32	5	61	16	4	1	8	2	1742	56102		14%	
..edge_to_rpm	1046	1073	28	56	56	56	0	1	1	1	0	1742	48776	12%		
..fdiv_func	1074	1087	14	14	14	14	0	1	1	1	0	3484	48776	12%		
..engine.1.1.1	886	889	4	17	7	31	8	4	2	8	2	1742	29042			
														100%	45%	17%
fir																
.	2	1057	1056	16211	16211	16211	0	1	1	1	0	1	16211	100%		
..fir_filter	869	915	47	529	529	529	0	1	1	1	0	10	5290	33%		
..fir_filter.1	889	903	15	497	497	497	0	34	34	34	0	10	4970		31%	
..sqrtd	548	597	50	561	561	561	0	1	1	1	0	10	3520	22%		
..sqrtd.1	568	586	19	532	532	532	0	19	19	19	0	10	3230		20%	
..fabsd	395	406	12	11	10	11	0	1	1	1	0	284	3082	19%		
..sind	407	468	62	161	55	227	53	1	1	1	0	20	2232	14%		
														87%	51%	
g3fax																
.	2	1095	1094	1128023	1128023	1128023	0	1	1	1	0	1	1128023	100%		
..main	932	1095	164	1127913	1127913	1127913	0	1	1	1	0	1	550587	49%		
..main.1	956	1075	120	1126855	1126855	1126855	0	35	35	35	0	1	550546		49%	
..main.1.1	975	1068	94	22680	4780	32124	8675	238	12	447	168	34	549660			49%
..main.1.1.1	1028	1033	6	135	10	10372	684	23	2	1729	114	2622	354534			
..rowout	912	931	20	10384	10384	10384	0	1	1	1	0	34	353056	31%		
..rowout.1	920	925	6	10370	10370	10370	0	1729	1729	1729	0	34	352580		31%	
..getbit	867	895	29	15	14	25	4	1	1	1	0	14337	220438	20%		
														100%	80%	49%

Figure 4: Loop statistics for MIPS (*jpeg*, *summin*, *ucbqsort*, and *v42*).

Region	Start	End	Static Size	Dynamic Instrs per Iteration				Iter per Exec.				Total Execs	Total Dynamic Instrs	%		
				avg	min	max	stddev	avg	min	max	stddev					
jpeg																
.	2	1491	1490	4594721	4594721	4594721	0	1	1	1	0	1	4594721	100%		
..fast_idct_8	1115	1331	217	217	217	217	0	1	1	1	0	9600	2083200	45%		
..huff_ac_dec	963	1114	152	2337	1544	4658	725	1	1	1	0	600	1081601	24%		
..huff_ac_dec.1	977	1068	92	1435	642	3756	725	6	2	19	4	600	540401		12%	
..main	1379	1491	113	4594611	4594611	4594611	0	1	1	1	0	1	476963	10%		
..main.5	1445	1473	29	452922	452922	452922	0	21	21	21	0	1	452922		10%	
..main.5.1	1446	1468	23	22640	22640	22640	0	8	8	8	0	20	452800			10%
..main.5.1.1	1448	1464	17	2824	2824	2824	0	31	31	31	0	160	451840			
..dquantz_lum	1362	1378	17	710	710	710	0	1	1	1	0	600	426000	9%		
..dquantz_lum.1	1365	1375	11	704	704	704	0	64	64	64	0	600	422400		9%	
..main.5.1.1.1	1449	1459	11	88	88	88	0	8	8	8	0	4800	422400			
..getbit	867	900	34	20	19	31	2	1	1	1	0	19228	381749	8%		
..huff_ac_dec.3	1085	1100	16	498	498	498	0	32	32	32	0	600	298800		7%	
..huff_ac_dec.1.6	1058	1065	8	437	226	506	65	55	29	64	8	600	261960			6%
														97%	37%	16%
summin																
.	2	1035	1034	1909787	1909787	1909787	0	1	1	1	0	1	1909787	100%		
..summation	927	987	61	44118	44118	44118	0	1	1	1	0	24	813120	43%		
..argmin	905	926	22	79	79	79	0	1	1	1	0	10000	790000	41%		
..argmin.1	910	921	12	69	69	69	0	7	7	7	0	10000	690000		36%	
..summation.2	951	978	28	27850	27850	27850	0	50	50	50	0	24	668400		35%	
..summation.2.1	952	974	23	552	552	552	0	24	24	24	0	1200	662400			35%
..init_2d	883	904	22	9779	9779	9779	0	1	1	1	0	24	234696	12%		
..init_2d.1	888	900	13	9770	9770	9770	0	25	25	25	0	24	234480		12%	
..init_2d.1.1	890	897	8	402	402	402	0	51	51	51	0	576	231552			12%
..summation.1	945	949	5	6002	6002	6002	0	1201	1201	1201	0	24	144048		8%	
														96%	91%	47%
ucbqsort																
.	2	1211	1210	219978	219978	219978	0	1	1	1	0	1	219978	100%		
..qst	1034	1211	178	12	9	90	8	1	1	2	0	11097	134628	61%		
..qst.1	1051	1199	149	12	9	51	3	1	1	2	0	11353	130887		60%	
..qst.1.3	1119	1169	51	11	7	36	2	1	1	2	0	7037	76297			35%
..qst.1.3.1	1128	1148	21	11	2	11	1	2	1	2	0	7055	75028			
..compare	867	870	4	4	4	4	0	1	1	1	0	12098	48392	22%		
..qst.1.2	1117	1126	10	9	2	10	1	2	1	2	0	4058	38364			17%
..QSORT	907	1033	127	19	11	75	2	1	1	1	0	1004	19085	9%		
..QSORT.3	985	1023	39	19	12	19	0	2	1	2	0	1000	18987		9%	
														92%	68%	52%
v42																
.	2	1598	1597	2442551	2442551	2442551	0	1	1	1	0	1	2442551	100%		
..search_dict	1049	1079	31	62	10	503	72	1	1	1	0	11074	687013	28%		
..add_dict	1080	1203	124	97	75	349	31	1	1	1	0	6922	674028	28%		
..search_dict.1	1061	1075	15	50	1	492	72	5	1	39	6	11071	558280		23%	
..decode	1399	1554	156	1040386	1040386	1040386	0	1	1	1	0	1	294901	12%		
..decode.1	1409	1544	136	1040366	1040366	1040366	0	3526	3526	3526	0	1	294881		12%	
..encode	1223	1398	176	1348598	1348598	1348598	0	1	1	1	0	1	252529	10%		
..encode.1	1237	1388	152	1348574	1348574	1348574	0	7557	7557	7557	0	1	252505		10%	
														78%	45%	
													Average:	93%	66%	

Figure 5: Loop statistics for 8051 (*bcnt* and *binary*).

Region	Start	End	Static Size	Dynamic Instrs per Iteration				Iter per Exec.				Total Execs	Total Dynamic Instrs	%		
				avg	min	max	stddev	avg	min	max	stddev					
bcnt																
.	0	5466	5467	131146	131146	131146	0	1	1	1	0	1	131146	100%		
..?C_C51STARTUP	5150	5289	140	131145	131145	131145	0	1	1	1	0	1	131145	60%		
..?C_C51STARTUP.5	5224	5288	65	78394	78394	78394	0	3	3	3	0	1	78394		60%	
..?C_C51STARTUP.5.1	5263	5286	24	39176.5	4609	73744	34567.5	2176	256	4096	1920	2	78353			60%
..?C?LIB_CODE	5290	5574	285	137.35	16	339	124.13	1	1	1	0	337	46288	35%		
..?C?LIB_CODE.1	5293	5306	14	224	112	336	91.45	16	8	24	6.53	192	43008		33%	
														95%	93%	60%
binary																
.	0	400	401	1016	1016	1016	0	1	1	1	0	1	1016	100%		
..?C_C51STARTUP	262	401	140	1015	1015	1015	0	1	1	1	0	1	1015	76%		
..?C_C51STARTUP.3	277	363	87	504	504	504	0	2	2	2	0	1	504		50%	
..?PR?_BINARY_SEARCH?BINARY	53	197	145	238	238	238	0	1	1	1	0	1	238	23%		
..?PR?_BINARY_SEARCH?BINARY.1	71	191	121	225	225	225	0	5	5	5	0	1	225		22%	
														99%	97%	

Figure 6: Loop statistics for 8051 (*blit*, *brev*, *crc*, *g3fax*, *matmul*, *summin*, and *ucbqsort*).

Region	Start	End	Static Size	Dynamic Instrs per Iteration				Iter per Exec.				Total Execs	Total Dynamic Instrs	%		
				avg	min	max	stddev	avg	min	max	stddev					
blit																
.	0	5824	5825	1229154	1229154	1229154	0	1	1	1	0	1	1229154	100%		
..?C?LIB_CODE	5682	5968	287	99.8	13	619	109.91	1	1	1	0	10023	1000323	81%		
..?C?LIB_CODE.2	5721	5734	14	224.27	42	616	57.44	16.02	3	44	4.1	2005	449652		37%	
..?C?LIB_CODE.1	5702	5715	14	223.66	70	280	56.36	15.98	5	20	4.03	2005	448448		36%	
..?PR?_BLIT?BLIT	3	1338	1336	577538	575651	579425	1887	1	1	1	0	2	1155076	13%		
..?PR?_BLIT?BLIT.4	693	845	153	578025	578025	578025	0	1001	1001	1001	0	1	578025		6%	
..?PR?_BLIT?BLIT.2	330	471	142	574024	574024	574024	0	1001	1001	1001	0	1	574024		6%	
..?C_C51STARTUP	5542	5681	140	1229153	1229153	1229153	0	1	1	1	0	1	1229153	6%		
..?C_C51STARTUP.5	5616	5680	65	73757	73757	73757	0	2	2	2	0	1	73757		6%	
..?C_C51STARTUP.5.1	5655	5678	24	73744	73744	73744	0	4096	4096	4096	0	1	73744			6%
														100%	92%	6%
brev																
.	0	2405	2406	82516	82516	82516	0	1	1	1	0	1	82516	100%		
..?C?LIB_CODE	2229	2496	268	77.92	16	227	74.53	1	1	1	0	769	59920	73%		
..?C?LIB_CODE.1	2232	2245	14	86.8	14	224	76.37	6.2	1	16	5.46	320	27776		34%	
..?C?LIB_CODE.2	2251	2264	14	86.8	14	224	76.37	6.2	1	16	5.46	320	27776		34%	
..?PR?MAIN?BREV	3	1763	1761	76460	76460	76460	0	1	1	1	0	1	76460	20%		
..?PR?MAIN?BREV.1	8	1754	1747	76455	76455	76455	0	1	1	1	0	1	76455		20%	
..?PR?MAIN?BREV.1.1	26	1735	1710	76442	76442	76442	0	17	17	17	0	1	76442			20%
..?C_C51STARTUP	2089	2228	140	82515	82515	82515	0	1	1	1	0	1	82515	7%		
..?C_C51STARTUP.5	2163	2227	65	5775	5775	5775	0	2	2	2	0	1	5775		7%	
..?C_C51STARTUP.5.1	2202	2225	24	5762	5762	5762	0	320	320	320	0	1	5762			7%
														100%	94%	27%
crc																
.	0	809	810	72799	72799	72799	0	1	1	1	0	1	72799	100%		
..?PR?_ICRC1?CRC	3	80	78	189	125	253	22.63	1	1	1	0	256	48384	66%		
..?PR?_ICRC1?CRC.1	17	74	58	176	112	240	22.63	8	8	8	0	256	45056		62%	
..?PR?_ICRC?CRC	81	474	394	35877	2393	69361	33484	1	1	1	0	2	71754	31%		
..?PR?_ICRC?CRC.1	116	230	115	67073	67073	67073	0	256	256	256	0	1	67073		26%	
..?PR?_ICRC1?CRC.1.1	30	35	6	7	7	7	0	2	2	2	0	1024	7168			10%
..?PR?_ICRC?CRC.2	297	419	123	2302	2246	2358	56	42	41	43	1	2	4604		6%	
														98%	93%	10%
g3fax																
.	0	8269	8270	4918854	4918854	4918854	0	1	1	1	0	1	4918854	100%		
..?PR?_ROWOUT?G3FAX	144	255	112	79521	79521	79521	0	1	1	1	0	34	2703714	49%		
..?PR?_ROWOUT?G3FAX.1	171	241	71	79504	79504	79504	0	1729	1729	1729	0	34	2703136		49%	
..?PR?MAIN?G3FAX	256	866	611	585	3	116975	6189	1	1	1	0	8063	4716620	33%		
..?PR?MAIN?G3FAX.1	261	831	571	585	55	116975	6189	1	1	1	0	8062	4716604		33%	
..?PR?MAIN?G3FAX.1.1	303	807	505	585	55	116975	6189	1	1	2	0	8062	4715836			33%
..?PR?MAIN?G3FAX.1.1.1	368	777	410	248	22	37282	1417	2	1	2	0	8095	2007554			
..?PR?MAIN?G3FAX.1.1.1.2	615	656	42	469	31	36310	2393	23	2	1729	114	2622	1230659			
..?C?LIB_CODE	8221	8307	87	5	5	13	1	1	1	1	0	68606	375282	8%		
..?PR?GETBIT?G3FAX	3	100	98	26	23	50	9	1	1	1	0	14337	376365	7%		
														97%	82%	33%
matmul																
.	0	835	836	29855	29855	29855	0	1	1	1	0	1	29855	100%		
..?C?LIB_CODE	671	842	172	15	8	19	4	1	1	1	0	925	13500	45%		
..?PR?_MATMUL?MATMUL	3	315	313	26922	26922	26922	0	1	1	1	0	1	26922	45%		
..?PR?_MATMUL?MATMUL.2	101	312	212	25394	25394	25394	0	5	5	5	0	1	25394		42%	
..?PR?_MATMUL?MATMUL.2.1	107	293	187	5069	5069	5069	0	5	5	5	0	5	25345			42%
..?PR?_MATMUL?MATMUL.2.1.1	113	274	162	1004	1004	1004	0	5	5	5	0	25	25100			
..?C_C51STARTUP	531	670	140	29854	29854	29854	0	1	1	1	0	1	29854	10%		
..?C_C51STARTUP.5	605	669	65	2636	2636	2636	0	3	3	3	0	1	2636		9%	
..?C_C51STARTUP.5.1	644	667	24	1297	1297	1297	0	72	72	72	0	2	2594			9%
														100%	51%	50%
summin																
.	0	1592	1593	27455473	27455473	27455473	0	1	1	1	0	1	27455473	100%		
..?C?LIB_CODE	1160	1647	488	30	7	86	24	1	1	1	0	546400	16193200	59%		
..?PR?SUMMATION?SUMMIN	444	824	381	37	18	5919	78	1	1	1	0	144048	5367960	19%		
..?PR?SUMMATION?SUMMIN.2	572	815	244	39	14	62	17	1	1	2	0	115224	4533576		17%	
..?PR?SUMMATION?SUMMIN.2.1	577	796	220	39	7	62	17	1	1	2	0	116400	4520400			16%
..?PR?_ARGMIN?SUMMIN	274	443	170	11	4	30	9	1	1	1	0	350000	3880000	14%		
..?PR?_ARGMIN?SUMMIN.1	303	432	130	11	1	30	9	1	1	2	0	340000	3710000		14%	
..?PR?_INIT_2D?SUMMIN	124	273	150	30	2	38	8	1	1	1	0	58800	1777560	6%		
..?PR?_INIT_2D?SUMMIN.1	143	270	128	30	1	38	8	1	1	2	0	58800	1777296		6%	
..?PR?_INIT_2D?SUMMIN.1.1	172	260	89	30	1	38	8	1	1	2	1	58752	1764864			6%
														99%	36%	23%
ucbqsort																
.	0	3062	3063	13430476	13430476	13430476	0	1	1	1	0	1	13430476	100%		
..?PR?_QSORT?UCBQSORT	133	765	633	3321588	7	13286268	5753111	1	1	1	0	4	13286350	56%		
..?PR?_QSORT?UCBQSORT.5	643	763	121	5805	97	25519	5708	2	2	2	0	974	5654147		31%	
..?PR?_QSORT?UCBQSORT.5.1	695	750	56	5765	57	25479	5708	152	2	671	150	974	5615184			31%
..?PR?_QSORT?UCBQSORT.4	504	669	166	7827	27	34291	7659	1	1	9	0	975	7631790		25%	
..?PR?_QSORT?UCBQSORT.4.1	547	587	41	7570	51	34221	7659	148	1	671	150	999	7562280			24%
..?C?LIB_CODE	2864	3065	202	5	5	108	0	1	1	1	0	596886	2984751	22%		
..?PR?_COMPARE?UCBQSORT	3	35	33	29	29	29	0	1	1	1	0	149612	4338748	21%		
														99%	56%	55%
													Average:	99%	77%	