A Study of the Cache and Branch Performance Issues with Running Java on Current Hardware Platforms

Cheng-Hsueh A. Hsieh Marie T. Conte Teresa L. Johnson
John C. Gyllenhaal Wen-mei W. Hwu
Center for Reliable and High-Performance Computing
University of Illinois
Urbana-Champaign, IL 61801
ada,mconte,tjohnson,gyllen,hwu@crhc.uiuc.edu

Abstract

The Java bytecode language is emerging as a software distribution standard. With major vendors committed to porting the Java run-time environment to their platforms, Java bytecode programs are expected to run without modification on multiple platforms. These first generation run-time environments rely on an interpreter to bridge the gap between the bytecode instructions and the native hardware. However, Java interpreters cause performance problems with microarchitectural features such as the caches and the Branch Target Buffer. Some of these problems can be solved by translating Java bytecode to native code. In this paper we compare the performance of code run through the SUN Java interpreter to code compiled through Caffeine*, a bytecode to native code translator, as well as to compiled C/C++ versions of the code, using large applications and common benchmarks. We discuss the reasons for several performance problems incurred by both approaches to running Java code, and examine possible solutions.*

1. Introduction

The speed of executing programs in a modern computer system is not determined solely by the number of instructions executed. A significant amount of the execution time can be attributed to inefficient use of the microarchitecture mechanisms such as the cache and branch prediction hardware.

Currently there are four approaches to running Java bytecode: an interpreter, a Just-In-Time (JIT) compiler, a Java bytecode to Native Executable Translation (NET) compiler, and a hardware implementation. Software interpreters have poor performance, although advances in processor speed have partially compensated for this disadvantage, and for small or rarely executed programs most users accept the added delay. Consumers also appear willing to accept the delay when using specialized applications such as Internet browsers, where performance is often limited by network delays rather than processor speed. However, this consumer acceptance quickly vanishes when running general applications that execute frequently.

We have studied the performance of a Java interpreter and have determined that a significant performance penalty is incurred due to the behavior of the underlying microarchitecture mechanisms. In understanding what features within the interpreter cause the performance penalties we hope to find ways to overcome them. One way to avoid the performance penalty experienced when executing Java is to eliminate the interpreter by translating the Java bytecode to efficient native code. We have developed an initial prototype of a NET compiler called Caffeine [6]. One of the objectives of our work is to run our translated code at nearly the full performance of native code directly generated from a source representation such as the C/C++ programming language. Our translated code still implements the Java language features and semantics [4] [1]. We use our translated code to isolate the interpreter behavior from the bytecode being executed and

use this, as well as C/C++ versions of the programs, to study the performance hits experienced in both approaches.

2. Experimental Model

We used several benchmarks to examine the performance of some of the approaches to running Java bytecode. Some of the benchmarks did not have Java versions available. These benchmarks were hand translated into Java code. They include smaller benchmarks such as *wc*, up to larger programs from the Spec '95 benchmark suite, such as *ijpeg* and *go*. This range of benchmarks was choosen to allow us to examine behavior across a range of application sizes. Other C benchmarks include *cmp*, *grep* and *des* (data encription standard), as well as *compress* from Spec '92. Additionally, we examined several applications developed in Java source language, *javac* and *cup*. For these benchmarks only translator and interpreter results are available, no native C code results can be shown.

The Java interpreter used is Sun JDK Version 1.0.2 running under Linux Version 2.1.13 on an x86 machine. The *IMPACT* NET Compiler, Caffeine Version 1.0, is used to generate x86 code running under Linux. The C code was optimized and scheduled for an Pentium Pro style implementation of the x86 instruction set using the *IMPACT* compiler [3]. We used the *IMPACT* simulation environment, which performs a detailed cycle-by-cycle execution-driven simulation, to generate results. Our simulation model is also based on a Pentium Pro style microarchitecture. It consists of a 32K first-level Icache with 64 byte blocks and 2-way associativity. We also include a 256-entry 2-level (GAp) BTB with 14 bits of history and 4-way associativity [7][5]. The data cache is a 2- level Dcache with a 16K first-level non-blocking Dcache containing 64 byte blocks, 2-way associativity with a maximum of 8 outstanding misses. The second-level Dcache is a 256K non-blocking Dcache with 256 byte blocks and 4-way associativity, allowing up to 50 outstanding misses.

3. Instruction Overhead

The first column of Tables 1 and 2 shows the total number of dynamic instructions executed by the processor for each benchmark under each execution model. As shown by the fourth row for each benchmark in Table 1, the interpreter approach results in many times more dynamic instructions than native C code. This is largely because interpreters must emulate all aspects of the Java Virtual Machine for each Java bytecode instruction in the program, resulting in many native instructions executed per bytecode instruction. The instruction overhead is particularly high for *cmp* due to the two calls to the *getc* I/O routine in its inner loop. Java adds an extra layer to the I/O calls in order to

System	Total Instr.	Total Pred.	Total Mispr.	Mispr. Dir.	Mispr. Target
WC					
C Code (Raw #)	5.70M	1.35M	117248	117246	2
Caffeine w/AIBC (Relative to C)	1.61	1.63	0.63	0.62	495.00
Caffeine w/o AIBC (Relative to C)	1.37	1.28	0.63	0.63	498.50
Sun Interpreter (Relative to C)	76.86	56.77	127.69	14.65	6.63M
GREP					
C Code (Raw #)	6.18M	1.91M	21881	21881	0
Caffeine w/AIBC (Relative to C)	2.52	2.19	1.46	1.38	1805/0
Caffeine w/o AIBC (Relative to C)	1.32	1.13	1.51	1.42	2023/0
Sun Interpreter (Relative to C)	43.79	24.61	452.62	10.98	9.66M/0
COMPRESS					
C Code (Raw #)	13.62M	2.41M	268398	268396	2
Caffeine w/AIBC (Relative to C)	2.18	2.46	1.02	1.02	832.00
Caffeine w/o AIBC (Relative to C)	1.43	1.14	1.01	1.01	947.00
Sun Interpreter (Relative to C)	67.75	68.02	123.64	7.70	15.56M
GO					
C Code (Raw #)	65.60M	12.12M	4.18M	3.78M	399592
Caffeine w/AIBC (Relative to C)	3.45	4.03	1.50	1.58	0.76
Caffeine w/o AIBC (Relative to C)	1.58	1.31	1.16	1.15	1.21
Sun Interpreter (Relative to C)	49.86	48.15	27.81	1.31	278.77
CMP					
C Code (Raw #)	6.32M	1.17M	8133	8133	0
Caffeine w/AIBC (Relative to C)	2.31	2.61	58.55	1.10	0.47M/0
Caffeine w/o AIBC (Relative to C)	1.87	1.81	58.55	1.10	0.47M/0
Sun Interpreter (Relative to C)	119.10	111.49	2825.17	149.92	21.8M/0
DES					
C Code (Raw #)	30.61M	444548	902	901	1
Caffeine w/AIBC (Relative to C)	2.94	29.03	101.61	100.62	996.00
Caffeine w/o AIBC (Relative to C)	1.38	2.48	3.64	2.50	1029.00
Sun Interpreter (Relative to C)	53.39	642.56	71598.0	185.12	644.11M
IJPEG					
C Code (Raw #)	34.57M	2.80M	245147	175138	70009
Caffeine w/AIBC (Relative to C)	3.79	8.49	10.32	2.05	31.01
Caffeine w/o AIBC (Relative to C)	2.06	3.20	9.77	1.26	31.08
Sun Interpreter (Relative to C)	66.26	135.02	292.99	4.31	1015.18

Table 1. BTB Performance.

System	Total Instr.	Total Pred.	Total Mispr.	Mispr. Dir.	Mispr. Target
CUP					
Caffeine w/AIBC (Raw #)	20.94M	3.86M	512568	246004	266564
Caffeine w/o AIBC (Raw #)	19.48M	3.42M	493476	232914	260562
Sun Interpreter (Raw #)	349.35M	58.41M	10.90M	1.47M	9.43M
JAVAC					
Caffeine w/AIBC (Raw #)	7.27M	1.63M	230465	153081	77384
Caffeine w/o AIBC (Raw #)	6.24M	1.30M	217631	139413	78218
Sun Interpreter (Raw #)	146.20M	24.77M	4.62M	690240	3.93M

Table 2. BTB Performance for Java Programs.

make the bytecode system-independent and add exception handling, which results in many extra instructions per I/O call.

Translating the Java bytecode programs to native code via Caffeine removes the software machine emulation overhead, and results in a large reduction in the total instruction count. However, the second row for each benchmark in Table 1 shows that there is generally still two to three times more instructions executed after translation, as compared to native C code. This is a result of the Java language specifications on exception handling as well as the Array Index Bounds Checking (AIBC) required by the specification. Note from the third row for each benchmark in Table 1 that removing AIBC support results in a total instruction count much closer to that of native C code. Both *cmp* and *ijpeg* still have roughly twice the instructions of native C code. This is in part due to I/O calls, which add extra bytecode instructions as explained earlier. Additionally, *ijpeg* has many pointer accesses. Because Java has no notion of pointers, our hand-translated Java bytecode implements these as array accesses, which often requires some extra code to setup the array base and index. When Caffeine translates the bytecode to native code these extra instructions are also translated and executed in the new native executable.

4. Branch Target Buffer Performance

The next area of investigation is branch target buffer (BTB) performance. In the case of interpreters, the branch instructions seen by the hardware are those of the interpreter, not the Java bytecode program. Thus, the BTB performance reflects the behavior of the interpreter. Tables 1 and 2 show that the largest number of mispredicted branches occur for the Sun interpreter. In Sun's current implementation, a bytecode instruction is decoded using a switch statement with over 200 cases. The target of the switch cannot be predicted from past history since each execution of the switch utilizes the bytecode instruction opcode it is executing to determine the correct target. Because this switch statement is actually translated into a series of jump statements, the majority of the mispredictions involve an incorrect target, not direction. This is reflected in the large number of target mispredictions for the interpreter shown in Tables 1 and 2, and the large number of jump mispredictions shown in Tables 3 and 4. The direction of the branches in the interpreter is actually more predictable than in the original native C code, as shown in Table 1, because direction mispredictions increase less than total predictions. However, also note from Table 1 that total mispredictions always increase much more than total instructions for the interpreter over native C code, except for in *go*. The behavior of *go* will be explained in greater detail in Section 5.

Many of the BTB performance problems can be eliminated by translating the bytecode to native code

System	Conditional		Jump Predictions	
	Total	Incorrect	Total	Incorrect
WC				
C Code (Raw #)	1.26M	117231	89169	8
Caffeine w/AIBC (Relative to C)	1.30	0.61	0.93	5.38
Caffeine w/o AIBC (Relative to C)	0.93	0.61	0.93	5.88
Sun Interpreter (Relative to C)	34.28	3.80	194.74	1.17M
GREP				
C Code (Raw #)	1.65M	21865	231061	4
Caffeine w/AIBC (Relative to C)	2.29	1.18	1.16	370.25
Caffeine w/o AIBC (Relative to C)	1.07	1.22	1.16	310.75
Sun Interpreter (Relative to C)	16.97	6.03	77.09	2.34M
COMPRESS				
C Code (Raw #)	1.76M	268373	551369	12
Caffeine w/AIBC (Relative to C)	2.80	1.01	0.43	40.42
Caffeine w/o AIBC (Relative to C)	1.01	1.01	0.43	17.00
Sun Interpreter (Relative to C)	52.25	3.00	90.23	2.13M
GO				
C Code (Raw #)	9.81M	3.11M	460492	111347
Caffeine w/AIBC (Relative to C)	4.51	1.04	4.66	5.73
Caffeine w/o AIBC (Relative to C)	1.15	1.04	4.66	3.69
Sun Interpreter (Relative to C)	36.04	1.48	486.38	991.84
CMP				
C Code (Raw #)	1.17M	8192	7618	3
Caffeine w/AIBC (Relative to C)	1.81	1.03	0.29	10.33
Caffeine w/o AIBC (Relative to C)	1.01	1.03	0.29	11.67
Sun Interpreter (Relative to C)	62.78	3.97	3381.73	4.37M
DES				
C Code (Raw #)	268580	880	1029	5
Caffeine w/AIBC (Relative to C)	45.73	2.10	259.85	42.80
Caffeine w/o AIBC (Relative to C)	1.79	1.78	259.86	13.60
Sun Interpreter (Relative to C)	576.14	27.71	126303.5	12.86M
IJPEG				
C Code (Raw #)	2.39M	170254	224590	594
Caffeine w/AIBC (Relative to C)	7.34	1.28	7.64	42.77
Caffeine w/o AIBC (Relative to C)	1.19	1.10	7.64	10.90
Sun Interpreter (Relative to C)	89.11	2.90	698.52	117850.31

Table 3. Conditional Branches vs. Jumps.

System	Conditional		Jump Predictions	
	Total	Incorrect	Total	Incorrect
CUP				
Caffeine w/AIBC (Raw #)	2105125	87654	156052	9547
Caffeine w/o AIBC (Raw #)	1673021	90955	156054	8244
Sun Interpreter (Raw #)	34108055	556347	13599630	6779781
JAVAC				
Caffeine w/AIBC (Raw #)	1133534	60958	87160	5598
Caffeine w/o AIBC (Raw #)	793632	57485	87406	4574
Sun Interpreter (Raw #)	14923330	331942	6413238	3160900

Table 4. Conditional Branches vs. Jumps for Java Programs.

using Caffeine. When the bytecode is translated to native code, the need for a software machine to execute the program is eliminated. Therefore, the branch instructions seen by the hardware correspond directly to the bytecode instructions. The additional conditional branches for Caffeine as compared to native C code, shown in Table 3, are due to the Java language specifications on exception handling and AIBC. This effect is particularly noticeable for *des* and *ijpeg*, where the inner loops have large numbers of array accesses, requiring high amounts of AIBC. These additional branches also exist in the interpreter approach, but are overwhelmed by the more dominating increase in jumps due to the software decoder explained earlier. Note from Table 3 that removing AIBC support reduces the number of conditional branches, almost to the same number as native C code. Since most of the conditional branches fall through (execution continues at the next sequential line of code), even with AIBC, the branch prediction rate is not severely impacted. However, since we are translating to static native code, some of these additional branch instructions could be eliminated by utilizing analysis tools [2].

Table 3 shows that there are large numbers of additional mispredictions for *cmp*, even after translation. These are a result of the two calls to the *getc* routine in the inner loop of *cmp*. Because *getc* is implemented in Java by calling an I/O buffer read function, and because that function returns to an alternating target address every time it is called, the return target is mispredicted every time. In C, *getc* is a macro which expands into a conditional statement calling the I/O buffer routine infrequently.

5. Instruction Cache Performance

Since the software decoder uses the switch statement discussed in Section 4, and the branch address out of this switch could target anywhere in the code address space, the target code is usually not in the Icache. This accounts for the large number of Icache misses incurred by the interpreter in comparison to native C code, as shown in Table 5. This problem is also reduced significantly by translation to native code, as reflected in Tables 5 and 6. The virtual machine's poor performance here is again due to the software decoding of instructions. By eliminating the software emulation layer, we eliminate the software decoder and with it the added Icache performance loss.

One exception is in the case of *go*, for which the Icache performance improves for the interpreter over native C code. The reason is that in *go* only a few bytecode types are executed most of the time, resulting in repeated execution of a few cases of the switch statement, and therefore more Icache reuse. Profiling shows that in *go* only 5 bytecode types make up 59% of the dynamic bytecode instructions. This fact also accounts for the smaller increase in total mispredictions

System	Icache Requests	Icache Misses	Blocks Purged	L1-L2 Bus Read Request Cycles
WC				
C Code (Raw #)	3.32M	13	0	13
Caffeine w/AIBC (Relative to C)	1.52	36.69	84/0	36.69
Caffeine w/o AIBC (Relative to C)	1.31	33.38	65/0	33.38
Sun Interpreter (Relative to C)	80.64	471.46	5628/0	471.46
GREP				
C Code (Raw #)	3.30M	36	0	36
Caffeine w/AIBC (Relative to C)	2.48	13.44	89/0	13.44
Caffeine w/o AIBC (Relative to C)	1.42	15.47	211/0	15.47
Sun Interpreter (Relative to C)	50.54	173.64	5750/0	173.64
COMPRESS				
C Code (Raw #)	8.17M	44	0	44
Caffeine w/AIBC (Relative to C)	1.98	12.39	158/0	12.39
Caffeine w/o AIBC (Relative to C)	1.36	24.57	737/0	24.57
Sun Interpreter (Relative to C)	69.53	152.86	6225/0	152.86
GO				
C Code (Raw #)	46.62M	242114	241602	242114
Caffeine w/AIBC (Relative to C)	2.79	11.87	11.89	11.87
Caffeine w/o AIBC (Relative to C)	1.46	2.41	2.41	2.41
Sun Interpreter (Relative to C)	43.02	0.08	0.07	0.08
CMP				
C Code (Raw #)	3.30M	11	0	11
Caffeine w/AIBC (Relative to C)	2.65	40.91	68/0	40.91
Caffeine w/o AIBC (Relative to C)	2.22	63.91	342/0	63.91
Sun Interpreter (Relative to C)	137.20	556.73	5623/0	556.73
DES				
C Code (Raw #)	15.50M	42	0	42
Caffeine w/AIBC (Relative to C)	2.94	25.21	556/0	25.21
Caffeine w/o AIBC (Relative to C)	1.38	12.26	150/0	12.26
Sun Interpreter (Relative to C)	65.85	159.05	6170/0	159.05
IJPEG				
C Code (Raw #)	18.77M	6402	5893	6402
Caffeine w/AIBC (Relative to C)	3.98	3.35	3.40	3.35
Caffeine w/o AIBC (Relative to C)	2.38	2.98	3.02	2.98
Sun Interpreter (Relative to C)	73.39	7.95	8.55	7.95

Table 5. Instruction Cache Performance.

System	Icache Requests	Icache Misses	Blocks Purged	L1-L2 Bus Read Request Cycles
CUP				
Caffeine w/AIBC (Raw #)	12.36M	17436	16927	17436
Caffeine w/o AIBC (Raw #)	11.61M	13695	13183	13695
Sun Interpreter (Raw #)	209.96M	334576	329460	33457
JAVAC				
Caffeine w/AIBC (Raw #)	4.40M	20372	19860	20372
Caffeine w/o AIBC (Raw #)	3.85M	29167	28655	29167
Sun Interpreter (Raw #)	88.23M	39679	39174	39679

Table 6. Instruction Cache Performance for Java Programs.

for *go* noted in Section 4.

Note that the total number of Icache requests shown in Tables 5 and 6 is smaller than the total number of instructions executed, shown in Tables 1 and 2. This is due to the fact that in a multiple issue machine, such as the Pentium Pro, multiple instructions are fetched from the Icache in one request.

6. Data Cache Performance

As mentioned earlier, in the interpreter approach the Java bytecode effectively becomes data. Therefore, both the benchmark data and the benchmark bytecode will be allocated in the Dcache, which will result in many more Dcache compulsory and conflict misses for the interpreter. This effect can be seen by examining Table 7, which shows the misses generated by the L1 Dcache, and verifies that the Sun interpreter incurs many more misses than native C code. Table 9 shows a corresponding increase in the bus cycles resulting from data traffic. Also, because the interpreter must read in the class files and set up internal data structures, a large number of read and write compulsory misses are incurred. This effect is most noticeable for *wc* and *grep*, which had particularly low numbers of misses in native C code. On the other hand, the increases are smallest for *compress*, *go* and *ijpeg*, because they already have high numbers of misses in native C code.

Translating the Java bytecode to native code will separate the benchmark data and code, as in native C code. Both Caffeine models have large improvements in Dcache performance over the Sun interpreter, as seen in Tables 7 - 10.

However, many of the translated benchmarks still have worse Dcache performance than native C code. One reason is that Java has no notion of array constants. Therefore, in Java each individual array element must explicitely be assigned a value at runtime (possibly with AIBC), whereas in native C code the array will simply be allocated in the data segment. Our translator currently cannot translate these explicit initializations into static variable declarations. Also, the Java bytecode contains some initializations of Java class library data. If these initializations cannot be fully resolved at translation time they must be explicitely performed at run time in the translated executable. Additionally, in benchmarks such as *wc* there are many calls to *getc*. The C preprocessor will normally expand this macro into a complex assignment statement involving a few data accesses. However, in Java *getc* is implemented as a routine, where several of the variables are accessed via object pointers, which can result in many more data accesses than an access with a direct data address. These extra accesses will be carried over to the translated code, and often result in extra misses.

Notice, however, that the translated native code for *cmp* contains less than half the Dcache misses of its native C code. Initial investigations suggest that an input buffer is conflicting with another data structure in the C code, resulting in many conflict misses. In the translated code this buffer is allocated at a different address that does not appear to cause conflicts.

System	Dcache Requests	Read Misses	Write Misses	Blocks Purged
WC				
C Code (Raw #)	1.49M	75	8	0
Caffeine w/AIBC (Relative to C)	2.29	4.11	211.50	84/0
Caffeine w/o AIBC (Relative to C)	2.13	4.12	168.38	85/0
Sun Interpreter (Relative to C)	138.45	9572.56	31435.5	480069/0
GREP				
C Code (Raw #)	1.37M	143	50	0
Caffeine w/AIBC (Relative to C)	3.05	10.32	253.02	1128/0
Caffeine w/o AIBC (Relative to C)	2.33	9.58	258.10	1045/0
Sun Interpreter (Relative to C)	100.09	652.43	1307.36	78184/0
COMPRESS				
C Code (Raw #)	3.20M	472230	198963	461098
Caffeine w/AIBC (Relative to C)	3.77	0.98	1.03	0.99
Caffeine w/o AIBC (Relative to C)	3.19	0.98	0.98	0.99
Sun Interpreter (Relative to C)	142.36	3.50	2.57	2.86
GO				
C Code (Raw #)	33.59M	771252	352744	700886
Caffeine w/AIBC (Relative to C)	2.93	1.53	0.93	1.48
Caffeine w/o AIBC (Relative to C)	1.81	1.16	1.18	1.12
Sun Interpreter (Relative to C)	50.32	11.05	4.01	10.00
CMP				
C Code (Raw #)	2.81M	4271	3559	3883
Caffeine w/AIBC (Relative to C)	2.26	0.13	0.43	0.08
Caffeine w/o AIBC (Relative to C)	2.10	0.13	0.44	0.08
Sun Interpreter (Relative to C)	124.62	512.99	136.25	380.98
DES				
C Code (Raw #)	6.91M	2414	3738	987
Caffeine w/AIBC (Relative to C)	4.80	8.44	11.81	16.42
Caffeine w/o AIBC (Relative to C)	2.64	9.99	5.96	16.43
Sun Interpreter (Relative to C)	122.46	172.51	20.70	391.65
IJPEG				
C Code (Raw #)	15.84M	46568	716353	35943
Caffeine w/AIBC (Relative to C)	3.81	2.46	1.16	2.69
Caffeine w/o AIBC (Relative to C)	2.65	2.16	1.36	2.12
Sun Interpreter (Relative to C)	77.22	26.12	1.98	26.14

Table 7. L1 Data Cache Performance.

7. Conclusions and Future Work

If a universal distribution language such as Java is to take a firm hold in the software market, it must handle large scale commonly used applications without incurring a noticeable delay. To better understand the tradeoffs and issues involved with the various approaches to running Java bytecode, we have studied the

performance of two options, an interpreter and a bytecode to native code translator. In this paper we compared the performance of code run through the SUN Java interpreter to code compiled through *Caffeine*, as well as to compiled C/C++ versions of the code, using large applications and common benchmarks. In particular, we focused on the performance of underlying microarchitectural features such as the branch target buffers, the instruction cache and the data cache, and we gave some insights into the performance problems incurred by both approaches to running Java code.

As future work we are examining ways to improve the performance of both the translator and the interpreter approaches. For the interpreter we will examine alternate BTB designs that look at the data fields of the instruction, rather than just the instruction alone, which will help improve the branch prediction performance of the highly input-dependent switch statement. For the translator, we will continue to explore the performance of our current implementation of Caffeine, and examine ways to improve the performance of the resulting native code. We are currently incorporating several optimization techniques from our compiler research into the translator.

System	Dcache Requests	Read Misses	Write Misses	Blocks Purged
CUP				
Caffeine w/AIBC (Raw #)	9.19M	60153	98249	50922
Caffeine w/o AIBC (Raw #)	8.94M	56657	94263	47995
Sun Interpreter (Raw #)	173.79M	1.08M	201883	922085
JAVAC				
Caffeine w/AIBC (Raw #)	2.67M	27418	38192	22964
Caffeine w/o AIBC (Raw #)	2.49M	28164	45756	22261
Sun Interpreter (Raw #)	73.25M	793954	245052	621032

Table 8. L1 Data Cache Performance for Java Programs.

System	L1-L2 Bus Cycles		System Bus Cycles	
	Read Request	Write Request	Read Request	Write Request
WC				
C Code (Raw #)	71	7	23	7
Caffeine w/AIBC (Relative to C)	4.07	313.29	4.43	134.29
Caffeine w/o AIBC (Relative to C)	4.08	264.86	4.43	83.86
Sun Interpreter (Relative to C)	6765.15	273477.14	33.35	832.00
GREP				
C Code (Raw #)	139	44	39	11
Caffeine w/AIBC (Relative to C)	9.96	386.86	6.90	333.45
Caffeine w/o AIBC (Relative to C)	9.36	385.82	6.92	354.00
Sun Interpreter (Relative to C)	564.32	3135.43	24.38	1020.27
COMPRESS				
C Code (Raw #)	461355	663837	76615	1.17M
Caffeine w/AIBC (Relative to C)	0.99	0.96	1.04	0.98
Caffeine w/o AIBC (Relative to C)	0.99	0.92	1.03	0.98
Sun Interpreter (Relative to C)	2.86	4.51	1.18	1.07
GO				
C Code (Raw #)	701143	2.23M	824	67226
Caffeine w/AIBC (Relative to C)	1.48	1.52	64.48	12.78
Caffeine w/o AIBC (Relative to C)	1.12	1.26	64.21	12.78
Sun Interpreter (Relative to C)	10.00	4.64	180.19	20.69
CMP				
C Code (Raw #)	4018	17348	40	6
Caffeine w/AIBC (Relative to C)	0.14	0.19	2.70	100.83
Caffeine w/o AIBC (Relative to C)	0.14	0.12	2.78	102.00
Sun Interpreter (Relative to C)	368.24	217.99	19.12	989.50
DES				
C Code (Raw #)	1156	4246	48	92
Caffeine w/AIBC (Relative to C)	14.24	39.20	62.00	785.18
Caffeine w/o AIBC (Relative to C)	14.25	34.02	62.04	727.66
Sun Interpreter (Relative to C)	334.62	90.53	191.25	2056.28
IJPEG				
C Code (Raw #)	36199	768879	511	288387
Caffeine w/AIBC (Relative to C)	2.68	1.41	18.16	1.69
Caffeine w/o AIBC (Relative to C)	2.11	1.51	16.56	2.21
Sun Interpreter (Relative to C)	25.97	3.09	68.73	2.91

Table 9. Bus Traffic Generated by Data Cache.

References

[1] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, Reading, MA, 1996.

[2] R. A. Bringmann. *Compiler-Controlled Speculation*. PhD thesis, Department of Computer Science, University of Illinois, Urbana, IL, 1995.

[3] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu. IMPACT: An architectural framework for multiple-instruction-issue processors. In *Proceedings of the 18th International Symposium on Computer Architecture*, pages 266–275, May 1991.

[4] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, Reading, MA, 1996.

[5] T. R. Halfhill. Intel's P6. *BYTE*, pages 42–58, April 1995.

[6] C.-H. A. Hsieh, J. C. Gyllenhaal, and W. W. Hwu. Java bytecode to native code translation: the caffeine prototype and preliminary results. In *Proceedings of the 29th International Symposium on Microarchitecture*, December 1996.

[7] T. Y. Yeh and Y. N. Patt. Two-level adaptive training branch prediction. In *Proceedings of the 24th Annual International Symposium on Microarchitecture*, pages 51–61, November 1991.

System	L1-L2 Bus Cycles		System Bus Cycles	
	Read Request	Write Request	Read Request	Write Request
CUP				
Caffeine w/AIBC (Raw #)	51178	351423	9136	266488
Caffeine w/o AIBC (Raw #)	48251	332528	9148	267126
Sun Interpreter (Raw #)	922342	1.38M	25731	475690
JAVAC				
Caffeine w/AIBC (Raw #)	23220	133658	3101	62954
Caffeine w/o AIBC (Raw #)	22517	141295	3205	66061
Sun Interpreter (Raw #)	621289	1.02M	15648	215400

Table 10. Bus Traffic Generated by Data Cache for Java Programs.