Using Complete System Simulation to Characterize SPECjvm98 Benchmarks

Tao Li[†], Lizy Kurian John[†], Vijaykrishnan Narayanan[*], Anand Sivasubramaniam[*], Jyotsna Sabarinathan[†], and Anupama Murthy[*]

[†] Laboratory for Computer Architecture
Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, TX 78712
{tli3,ljohn,sabarina}@ece.utexas.edu

[*] 220 Pond Lab
Department of Computer Science and Engineering
The Pennsylvania State University
University Park, PA 16802
{vijay,anand}@cse.psu.edu

ABSTRACT

Complete system simulation to understand the influence of architecture and operating systems on application execution has been identified to be crucial for systems design. While there have been previous attempts at understanding the architectural impact of Java programs, there has been no prior work investigating the operating system (kernel) activity during their executions. This problem is particularly interesting in the context of Java since it is not only the application that can invoke kernel services, but so does the underlying Java Virtual Machine (JVM) implementation which runs these programs. Further, the JVM style (JIT compiler or interpreter) and the manner in which the different JVM components (such as the garbage collector and class loader) are exercised, can have a significant impact on the kernel activities.

To investigate these issues, this research uses complete system simulation of the SPECjvm98 benchmarks on the SimOS simulation platform. The execution of these benchmarks on both JIT compilers and interpreters is profiled in detail, to identify and quantify where time is spent in each component. The kernel activity of SPECjvm98 applications constitutes up to 17% of the execution time in the large dataset and up to 31% in the small dataset. The average kernel activity in the large dataset is approximately 10%, in comparison to around 2% in four SPECInt benchmarks studied. Of the kernel services, TLB miss handling is the most dominant in all applications. The TLB miss rates in the JIT compiler, dynamic class loader and garbage collector portions of the JVM are individually analyzed. In addition to such execution profiles, the ILP in the user and kernel mode are also quantified. The Java code is seen to limit exploitable parallelism and aggressive instruction issue is seen to be less efficient for SPECjvm98 benchmarks in comparison to SPEC95 programs. Also, the kernel mode of execution does not exhibit as much ILP as the user mode.

This research is supported in part by the National Science Foundation under Grants EIA-9807112, CCR-9796098, CCR-9900701, Career Award MIP-9701475, State of Texas ATP Grant #403, and by Sun Microsystems®, Dell®, Intel®, Microsoft® and IBM®.

1. INTRODUCTION

Java offers the "write-once run-anywhere" promise that helps to develop portable software and standardized interfaces spanning a spectrum of hardware platforms. The Java Virtual Machine (JVM) is the cornerstone of this technology, and its efficiency in executing portable Java bytecodes is crucial for the success and widespread adoption of Java. A first step towards building an efficient JVM is to understand its interaction with the underlying system (both hardware and operating system), and to identify the bottlenecks. Such a study can provide information to optimize both systems software and architectural support for enhancing the performance of a JVM. In addition, a closer look at the execution profile of the JVM can also give revealing insights that can help to restructure its implementation. To our knowledge, existing studies [1, 2, 3, 4, 5, 6, 7, 8] have been confined to examining JVM profiles from the architectural perspective, and there has been no attempt at understanding the influence of the operating system activities. It is becoming increasingly clear [9, 10, 11, 12] that accurate performance analysis requires an examination of complete system - architecture and operating system - behavior.

Adhering to this philosophy, this paper presents results from an in-depth look at complete system profiling of the SPECjvm98 benchmarks, focusing on the operating system activity. Of the different JVM implementation styles [13, 14, 15, 16, 17, 18], this paper focuses on two popular techniques - interpretation and Just-In-Time (JIT) compilation. Interpretation [13] of the portable Java byte codes was the first approach that was used, and is, perhaps, the easiest to implement. In contrast, JIT compilers [14, 15, 16], which represent the state-of-the-art, translate the byte-codes to machine native code at runtime (using sophisticated techniques) for direct execution. While JIT compilers are known to outperform interpreters, it is still important to understand the performance of the interpretation process since it is a popular paradigm of Java execution and since it is an integral part of sophisticated JIT compilers [19]. Further, interpreters need a lower amount of memory than their JIT compiler counterparts, which can become important in resource-constrained environments, such as hand-held devices.

While complete system simulation has been used to study several workloads [9, 10, 11], it has not been used in the context of Java programs or JVM implementations. A JVM environment can be significantly different from that required to support traditional C or FORTRAN based code. The major differences are due to: 1) object-oriented execution with frequent use of virtual method calls (dynamic binding), dynamic object allocation and garbage collection; 2) dynamic linking and loading of classes; 3) program-level multithreading and consequent synchronization overheads; and 4) software interpretation or dynamic compilation of byte-codes. These differences can affect the behavior of the operating system kernel in a different manner than conventional applications. For instance, dynamic linking and loading of classes can result in higher file and I/O activities, while dynamic object allocation and garbage collection would require more memory management operations. Similarly, multithreading can influence the synchronization behavior in the kernel. A detailed profile of the interaction with the hardware and operating system can help us understand such intricacies so that the JVM can be restructured for better performance. Further, such profiles are also useful from the architectural and operating systems perspective to identify enhancements for boosting Java performance. As an example, one could opt to provide a multi-ported cache if it is found that memory bandwidth related stall cycles are a major impediment to increasing instruction-level parallelism.

Towards the goal of studying and profiling the complete system (architecture and operating system) behavior when executing Java programs, this paper specifically attempts to answer the following questions:

• How much time is spent in user and kernel modes? What kernel activities are exercised during the execution, and how much overhead is incurred in each of these activities? Are Java studies without OS activity representative of the aggregate Java execution behavior? How much of the time is spent in actually executing instructions (useful work), as opposed to being spent in stalls, synchronization and idling?

• Are these profiles different for the JIT compilation and interpreter modes? Can we attribute the kernel activities to specific phases (class loading, garbage collection, etc.) of the JVM execution?

• How are the kernel and user parts of the JVM execution suited to the underlying parallelism in modern microprocessors? What are the characteristics of these portions influencing the instruction level parallelism (ILP) that they can exploit? What is the ideal speedup that one could ever get for these portions?

We set out to answer these questions using the SPECjvm98 [20] benchmarks and the SimOS [10] complete system simulator. We find that the kernel activities are not as insignificant as in four studied SPEC95 [21] benchmarks. On the average, the SPECjvm98 workloads spend 10% of their execution time in kernel, contrasting to the less than 2% of kernel time found in studied SPEC95 benchmarks. Most of the kernel time is spent in TLB miss handling, with a significant fraction due to Java specific features for the JIT compiler mode. Among the architectural improvements, the most notable gain (20%-32%) is achieved by a 2-ported cache. We also find that dynamic scheduling, wide issue and retirement are not very effective for SPECjvm98 codes due to the inherent ILP limitations of Java code.

The rest of this paper is organized as follows. The next section gives an overview of the experimental platform and workloads. Section 3 presents the execution time and detailed statistics for the user and kernel activities in these workloads. Section 4 explores the ILP issues. Finally, Section 5 summarizes the contributions and implications of this work and identifies directions for future research.

2. EXPERIMENTAL METHODOLOGY

The experimental platform used to perform this study is SimOS [11, 22, 10], a complete simulation environment that models hardware components with enough detail to boot and run a full-blown commercial operating system. In this study, the SimOS version that runs the Silicon Graphics IRIX5.3 operating system was used. The interpreter and JIT compiler versions of the Java Development Kit 1.1.2 [23] from Sun Microsystems are simulated on top of the IRIX 5.3 operating system. Our studies are based on programs from the SPECjvm98 suite (see Table 1), a set of programs intended to evaluate performance for the combined hardware (CPU, cache, memory, and other platform-specific performance) and software aspects (efficiency of JVM, the JIT compiler, and OS implementations) of the JVM client platform [20]. The SPECjvm98 uses common computing features such as integer and floating point operations, library calls and I/O, but does not include AWT (window), networking, and graphics. Each benchmark can be run with three different input sizes referred to as s1, s10 and s100; however, it was observed that these data sizes do not scale linearly, as the naming suggests.

Table 1. SPECjvm98 Workloads

Benchmarks	Description
`compress`	Modified Lempel-Ziv method (LZW) to compress and decompress large file
`jess`	Java expert shell system based on NASA's CLIPS expert system
`db`	Performs multiple database functions on a memory resident database
`javac`	The JDK 1.0.2 Java compiler compiling 225,000 lines of code
`mpegaudio`	Decoder to decompress MPEG-3 audio file
`mtrt`	Dual-threaded raytracer
`jack`	Parser generator with lexical analysis, early version of what is now JavaCC

SimOS includes multiple processor simulators (Embra, Mipsy, and MXS) that model the CPU at different levels of detail [10]. We use the fastest CPU simulator, Embra [24] to boot the OS and perform initialization, and then use Mipsy and MXS, the detailed CPU models of SimOS to conduct performance measurements. For the large and complex workloads, the booting and initialization phase may cause the execution of several tens of billions of instructions [12]. SimOS has a checkpointing ability which allows the hardware execution status (e.g. contents of register file, main memory and I/O devices) to be saved as a set of files (dubbed as a checkpoint), and simulation may resume from the checkpoint. This feature allows us to conduct multiple runs from identical initial status. To ensure that SimOS accurately simulates a complete execution of each workload, we write annotations that allow SimOS to automatically invoke a studied workload after a checkpoint is restored and to exit simulation as soon as the execution completes and OS prompt is returned. Our techniques, which avoid the need of interactive input to control the simulation after it begins and before it completes, make each run complete, accurate, and comparable.

The performance results presented in this study are generated by Mipsy and MXS, the detailed CPU models of SimOS. Mipsy models a simple, single-issue pipelined processor with one-cycle result latency and one-cycle repeat rate [10]. Although Mipsy is not an effective model from the perspective of detailed processor performance investigations, it does provide valuable information such as TLB activities, instruction counts, and detailed memory system behavior. We use Mipsy to generate the basic characterization knowledge and memory system behavior of studied workloads.

The performance evaluation of microarchitecture level optimizations are done with MXS [25], which models a superscalar microprocessor with multiple instruction issue, register renaming, dynamic scheduling, and speculative execution with precise exceptions. Our baseline architectural model is a four issue superscalar processor with MIPS R10000 [26, 27] instruction latencies. Unlike the MIPS R10000, our processor model has a 64-entry instruction window, a 64-entry reorder buffer and a 32-entry load/store buffer. Additionally, all functional units can handle any type of instructions. Branch prediction is implemented as a 1024-entry table with 3-bit saturating counter predictors. By default, the branch prediction algorithm allows fetch unit to fetch through up to 4 unresolved branches.

The memory subsystem consists of a split L1 instruction and data cache, a unified L2 cache, and main memory. The processor has a MIPS R4000 TLB with a base page size of 4KB. The L1 instruction cache is 32KB, and has a cache line size of 64-bytes. The L1 data cache is 32KB, and has 32-byte lines. The L2 cache is 1MB with 128-byte lines. A hit in the L1 cache can be serviced in one cycle, while a hit in the L2 cache is serviced in 10 cycles. All caches are 2-way associative, with LRU replacement and write back write miss allocation policies and have four miss status handling registers (MSHR) [28]. Main memory consists of 256 MB DRAM with a 60-cycle access time. Our simulated machine also includes a validated HP97560 disk model and a single console device. The described architecture is simulated cycle by cycle. The instruction and data accesses of both applications and operating system are modeled.

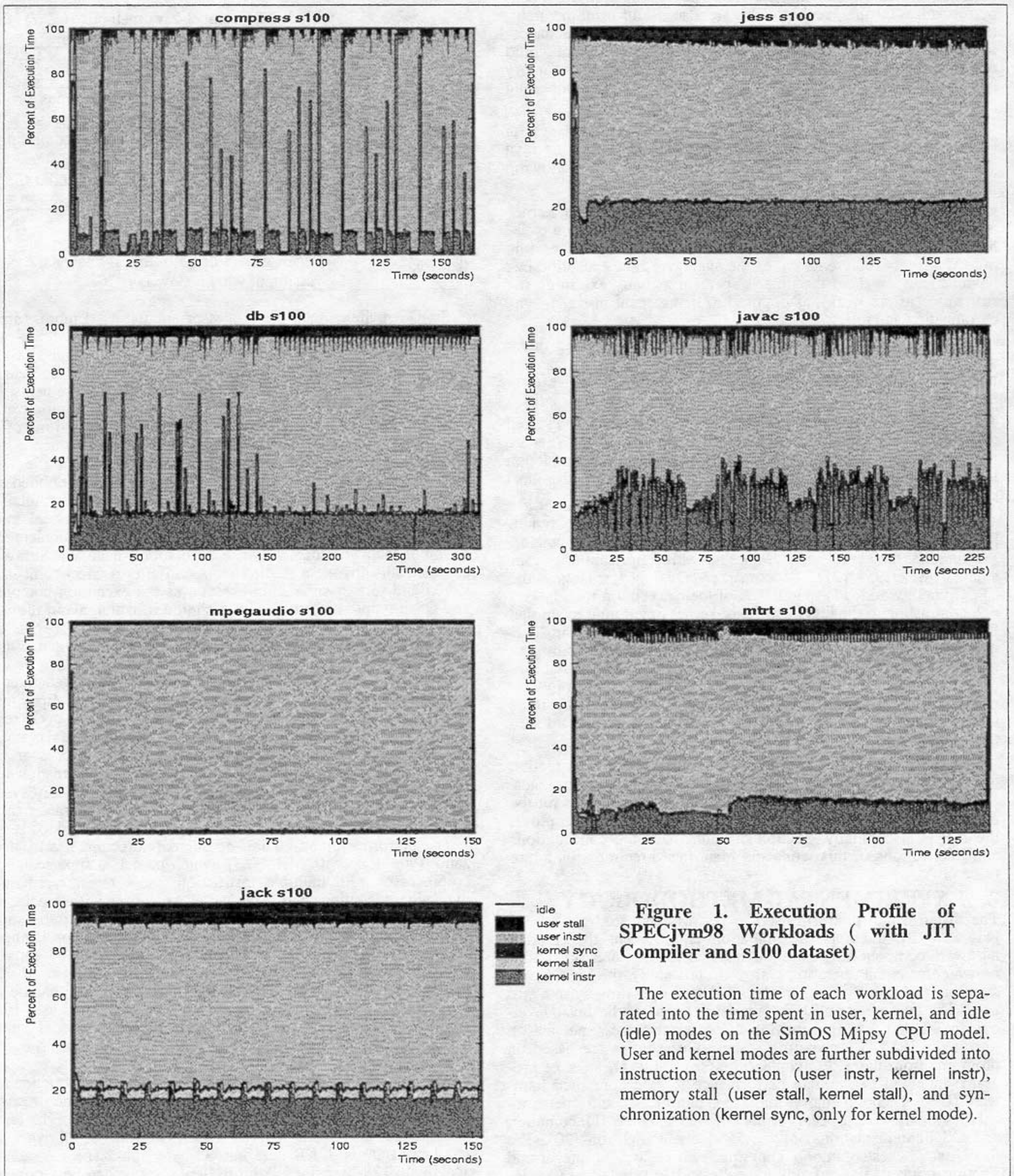


Figure 1. Execution Profile of SPECjvm98 Workloads (with JIT Compiler and s100 dataset)

The execution time of each workload is separated into the time spent in user, kernel, and idle (idle) modes on the SimOS Mipsy CPU model. User and kernel modes are further subdivided into instruction execution (user instr, kernel instr), memory stall (user stall, kernel stall), and synchronization (kernel sync, only for kernel mode).

The JVM and SPECjvm98 could be ported to SimOS in a straightforward manner because the SimOS environment allows any binary that is compatible to the IRIX 5.3 operating system. We are able to investigate complete executions of real world workloads due to this reason. In order to validate our experimental environment, during the early stages of this study, we worked with `pmake`, SPLASH-2 and some SPEC95 benchmarks and compared our experimental results with those published in [12, 25, 29, 30]. After the results were validated, we switched to the SPECjvm98 suite.

3. KERNEL ACTIVITY OF SPECjvm98

3.1 Execution Profile Using JIT Compiler and Interpreter

Figure 1 shows the execution time profile of the SPECjvm98 benchmarks for JIT compiler mode of execution. The measured period includes time for loading the program, verifying the class files, compiling on the fly by JIT compiler and executing native instruction stream on simulated hardware. The profile is presented in terms of the time spent in executing user instructions, stalls incurred during the execution of these instructions (due to memory and pipeline stalls), the time spent in kernel instructions, the stalls due to these kernel instructions, synchronization operations within the kernel and any remaining idle times. Some of the applications (`compress`, `jack`, and `javac`) iterate multiple times over the same input [35], which contributes the repetitive shape of some graphs in Figure 1.

Table 2 summarizes the breakdown of execution time spent in kernel, user, and idle for each SPECjvm98 benchmark. The kernel activity is seen to constitute 0.2% (`mpegaudio` with `intr`) to 17% (`jack` with `jit`) of the overall execution time. On the average, the SPECjvm98 programs spend 10.4% (with `jit`) and 7.7% (with `intr`) of their execution time in kernel, contrasting to the less than 2% of that found in SPECInt95 benchmarks. This fact implies that ignoring kernel instructions in SPECjvm98 workloads study may not represent complete and accurate execution behavior.

Table 2. Execution Time Percentages (s100 Dataset, J: with JIT compiler, I: with Interpreter)

Benchmarks		User	User Inst.	User Stall	Kernel	Kernel Inst.	Kernel Stall	Kernel Sync.	Idle
comp	J	92.81	87.19	5.62	4.30	3.78	0.49	0.03	2.89
	I	98.88	98.23	0.65	0.82	0.69	0.12	0.01	0.30
jess	J	84.95	73.63	11.32	14.90	14.19	0.66	0.05	0.15
	I	92.26	85.82	6.44	7.67	7.32	0.33	0.02	0.07
db	J	87.10	77.50	9.60	12.64	11.91	0.69	0.04	0.26
	I	83.28	76.90	6.38	16.57	16.05	0.50	0.02	0.15
javac	J	84.31	70.92	13.39	14.92	13.85	1.03	0.04	0.77
	I	88.57	79.77	8.80	10.92	10.15	0.75	0.02	0.51
mpeg	J	99.08	96.94	2.14	0.73	0.44	0.26	0.03	0.19
	I	99.78	98.95	0.83	0.20	0.12	0.07	0.01	0.02
mtrt	J	91.22	80.34	10.88	8.60	7.86	0.71	0.03	0.18
	I	98.26	94.33	3.93	1.70	1.22	0.47	0.01	0.04
jack	J	82.94	72.51	10.43	16.90	13.51	2.96	0.43	0.16
	I	83.78	74.08	9.70	16.11	13.25	2.52	0.34	0.11

Tables 3 and 4 further break down the kernel activities into specific services. These tables give the number of invocation of these services, the number of cycles spent in executing each routine on the average, a break down of these cycles between actual instruction execution, stalls and synchronization. The memory cycles per instruction (MCPI) while executing each of these services is also given together with its breakdown into instruction and data portions. The read or write kernel may involve disk accesses and subsequent copying of data between file caches and user data structures. It should be noted that the time spent in disk accesses is not accounted for within the read or write kernel calls, but will figure as idle times in the execution profile (because the process is blocked on I/O activity). So the read and write overheads are mainly due to memory copy operations.

In the execution profile graphs, we see that the bulk of the time is spent in executing user instructions. This is particularly true for `mpeg` that spends 99% of its execution in user instructions in both interpreter (Profile graphs for interpreter can be found in [31] and are omitted for lack of space) and JIT compiler modes. The `mpeg` benchmark decompresses audio files based on the MPEG layer3 audio specification. While I/O (read) is needed to load the audio files, subsequent executions are entirely user operations. These operations are mainly compute intensive with substantial spatial and temporal locality (as can be seen in the significantly lower user stalls compared to other applications in Table 2). This locality also results in high TLB hit rates making the TLB handler (*utlb*) invocation infrequent. This is also the reason why the percentage of kernel time spent in *utlb* is much lower (less than 35% in both execution modes) as compared to the other applications (See Tables 3 and 4). As a result, other service routines such as the *clock*, *read* and *runqproc* constitute a reasonable fraction of the kernel execution time.

While user execution constitutes over 90% of the time in `compress` as well, one can observe spikes in the kernel activity in the execution. This benchmark reads in five tar files and compresses them. These operations are followed by the decompression of the compressed files. The spikes are introduced by the file activities that can be attributed to both the application (loading of the five tar files) as well as the JVM characteristics. Most of the time spent in these spikes (*read*) is in memory stalls, particularly when reading data from file buffers. This is reflected in the higher d-MCPI compared to i-MCPI for the *read* routine in Tables 3 and 4. Other kernel routines such as *demand_zero* that is used to initialize new pages before allocation, and the process clock interrupt (*clock*) routines also contribute to the stalls. Despite these spikes, I/O activity still constitutes less than 10% of the kernel execution time. In addition to the spikes, we also see a relatively uniform presence of kernel instructions during the course of execution. As evident from Tables 3 and 4, this is due to the handling of TLB misses. In the user mode and with the JIT compiler, we find around 5% of the time being spent in stalls. The JIT compiler generates and installs code dynamically during execution resulting in bursty writes to the memory, leading to increased memory stalls in the JIT mode.

In general, the relatively flat kernel activity in the lower portion of Figure 1 is mainly due to TLB miss handling while spikes can be attributed to other services (*read* in particular). The kernel activity in `mtrt` and `jess` are dominated by the TLB miss handling with a small part spent in I/O (*read*) during initial class loading. On the other hand, the I/O (*read*) component of `jack` is 20-25% of the kernel execution time, with the stalls from this component showing up in the execution profile graphs. The TLB miss handling still constitutes the major portion. Benchmark `jack` performs 16 iterations of building up a live heap structure and collapsing it again while repeatedly generating a parser from the same input [35]. This behavior explains the repeated pattern observed for the kernel activity. The TLB miss handling overhead of `javac` is not as uniform as in the other applications. This application is a Java compiler that compiles the code for the `jess` benchmark repeatedly for four times. We observe this non-uniformity in the user stalls (top portion of the profile). This can be attributed to the code installation spikes during code generation by the compiler application. This is similar to the reason for the differences in stall behaviors between the JIT compiler and interpreter mode for compress. Code installation also worsens the locality of the application [2] resulting in higher TLB misses during those phases.

Figure 2 shows the memory stall time expressed as memory stall time per instruction (MCPI). The stall time is shown separately for the both the kernel (-K) and user (-U) modes and is also decomposed into instruction (-I) and data (-D) stalls. Further, the stalls are shown as that occurring due to L1 or L2 caches. For both the JIT compiler and interpreter modes of execution, it is observed that the kernel routines can experience much higher MCPI than user code, indicating the worse memory system behavior of the kernel. For example, mpeg has a MCPI of 0.55 within the kernel compared to the negligible MCPI for the user mode, since it touches a number of different kernel routines (See Tables 3 and 4) and data structures. Fortunately, the kernel portion forms a maximum of only 17% of the overall execution time among all the SPECjvm98 benchmarks and this mitigates the impact on overall MCPI. It can also be observed from Figure 2 that the MCPI in the user mode is less for the interpreter mode as compared to the JIT mode. The bursty writes during dynamic compilation and the additional non-memory instructions executed while interpreting the bytecodes result in this behavior. It is also observed that the stalls due to data references are more significant than that due to the instruction accesses. The MCPI due to L2 cache accesses is quite small for the compress and mpeg workloads that exhibit a significant data locality. The other SPECjvm98 benchmarks can, however, benefit from stall reduction techniques employed for the L2 cache.

Table 3. OS Kernel Characterization of SPECjvm98 Workloads (with JIT Compiler and s100 Dataset)
utlb fault reloads the TLB for user addresses. *demand_zero* is a block clear operation occurs when the operating system allocates a page for data. (The page has to be zeroed out before being used.) The *read* system calls is responsible for transferring data from kernel address space to application address space. *clock* and *vfault* are clock interrupt and page fault handler respectively.

Benchmarks	Service	%Kernel	Num.	Cycles	%Exec	%Stall	% Sync	MCPI	d-MCPI	i-MCPI
compress	utlb	80.85	8.64E+07	13	99	1	0	0.01	0.01	0
	read	9.51	6317	21140	39	58	3	1.42	1.32	0.1
	clock	3.41	16328	2934	37	60	3	1.56	1.07	0.49
	demand_zero	2.33	4807	6813	44	53	3	1.13	1	0.13
	other	3.90	--	--	--	--	--	--	--	--
jess	utlb	95.10	3.69E+08	13	98	2	0	0.02	0.02	0
	clock	1.48	17342	4396	26	72	2	2.77	1.44	1.33
	read	1.40	20889	3474	67	22	11	0.3	0.04	0.26
	other	2.02	--	--	--	--	--	--	--	--
db	utlb	94.17	5.60E+08	13	97	3	0	0.03	0.03	0
	clock	1.95	31439	4917	23	75	2	3.21	1.64	1.57
	read	1.44	30048	3804	61	29	10	0.41	0.1	0.31
	other	2.44	--	--	--	--	--	--	--	--
javac	utlb	91.39	4.71E+08	13	96	4	0	0.04	0.04	0
	DBL_FAULT	3.82	2812267	94	90	10	0	0.11	0.07	0.04
	clock	1.60	23302	4786	23	74	3	3.1	1.41	1.69
	read	1.0	10652	6386	48	46	6	0.89	0.41	0.48
	other	2.19	--	--	--	--	--	--	--	--
mpeg	utlb	34.22	5273604	14	90	10	0	0.11	0.11	0
	clock	23.34	15070	3426	32	65	3	1.96	0.93	1.03
	read	19.47	8013	5376	55	37	8	0.61	0.28	0.33
	runqproc	4.09	1	9039903	48	49	3	0.99	0.39	0.6
	demand_zero	3.09	997	6865	43	54	3	1.18	1	0.18
	BSD	2.07	4359	1052	52	45	3	0.84	0.33	0.51
	timein	1.66	1008	3633	44	48	8	0.99	0.34	0.65
	cacheflush	1.64	1821	1995	51	45	4	0.85	0.38	0.47
	open	1.49	228	14470	56	30	14	0.44	0.16	0.28
	fork	1.16	26	99071	39	51	10	1.1	0.57	0.53
	other	7.77	--	--	--	--	--	--	--	--
mtrt	utlb	93.41	1.61E+08	13	95	5	0	0.05	0.05	0
	clock	2.45	13745	4222	26	71	3	2.64	1.26	1.38
	read	1.19	7403	3804	64	26	10	0.36	0.11	0.25
	other	2.95	--	--	--	--	--	--	--	--
jack	utlb	63.13	2.38E+08	13	95	5	0	0.05	0.05	0
	read	25.21	296866	4401	52	40	8	0.67	0.09	0.58
	BSD	9.32	585482	825	67	30	3	0.44	0.14	0.3
	clock	1.09	15332	3686	30	68	2	2.2	0.92	1.28
	other	1.25	--	--	--	--	--	--	--	--

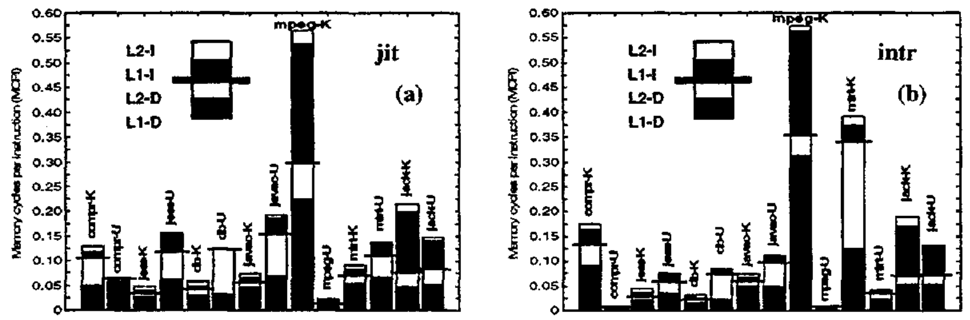


Figure 2. Memory Stall Time in Kernel (-K) and User (-U) Modes: (a) jit, (b) intr, both with s100 Dataset

Table 4. OS Kernel Characterization of SPECjvm98 Workloads (with Interpreter and s100 Dataset)

Benchmarks	Service	% Kernel	Num.	Cycles	%Exec	%Stall	% Sync	MCPI	d-MCPI	i-MCPI
compress	utlb	73.46	1.39E+08	13	98	2	0	0.02	0.02	0
	clock	13.64	152657	2245	49	48	3	0.94	0.67	0.27
	read	5.32	6324	21119	39	58	3	1.42	1.32	0.10
	runqproc	3.20	1	8026993	54	43	3	0.76	0.35	0.41
	timein	1.15	9336	3107	54	36	10	0.60	0.30	0.30
	demand_zero	1.02	3767	6786	44	53	3	1.12	0.99	0.13
	other	2.21	--	--	--	--	--	--	--	--
jess	utlb	94.20	4.17E+08	13	99	1	0	0.01	0.01	0
	clock	2.38	38068	3656	31	67	2	2.14	1.13	1.01
	read	1.30	20896	3625	65	25	10	0.35	0.04	0.31
	other	2.12	--	--	--	--	--	--	--	--
db	utlb	96.64	1.38E+09	13	98	2	0	0.02	0.02	0
	clock	1.21	56665	4008	28	70	2	2.44	1.33	1.11
	other	2.15	--	--	--	--	--	--	--	--
javac	utlb	93.67	5.53E+08	14	96	4	0	0.04	0.04	0
	clock	1.82	36676	3972	28	70	2	2.40	1.21	1.19
	DBL_FAULT	1.76	1487739	95	91	9	0	0.10	0.07	0.03
	other	2.75	--	--	--	--	--	--	--	--
mpeg	clock	49.93	136610	1974	55	41	4	0.71	0.45	0.26
	utlb	16.31	5643427	16	83	17	0	0.20	0.20	0
	runqproc	12.06	1	6511053	58	38	4	0.61	0.26	0.35
	read	7.82	8020	5268	56	36	8	0.57	0.25	0.32
	timein	4.59	8327	2976	57	33	10	0.53	0.27	0.26
	fork	2.48	26	515111	42	47	11	0.97	0.57	0.40
	bdflush	1.30	1	7014395	37	61	2	1.56	1.35	0.21
	demand_zero	1.09	857	6856	43	54	3	1.18	1.0	0.18
	other	4.42	--	--	--	--	--	--	--	--
mtrt	utlb	83.04	7.95E+07	17	77	23	0	0.29	0.29	0
	clock	9.13	47562	3096	36	61	3	1.66	1.06	0.6
	read	1.77	7410	3848	63	27	10	0.38	0.11	0.27
	runqproc	1.75	1	2821687	47	50	3	0.99	0.41	0.58
	demand_zero	1.0	2173	7375	40	57	3	1.31	1.09	0.22
	other	3.31	--	--	--	--	--	--	--	--
jack	utlb	70.21	3.51E+08	14	95	5	0	0.05	0.05	0
	read	20.30	296873	4672	49	43	8	0.77	0.09	0.68
	BSD	7.48	585470	872	63	33	4	0.52	0.21	0.31
	clock	1.08	21211	3495	31	66	3	2.03	0.85	1.18
	other	0.93	--	--	--	--	--	--	--	--

3.2 Differences within JVM Execution Styles and Dataset Sizes

Until now we have made some general observations regardless of the JVM execution style. We identify below some differences in the behavior of interpreters and JIT compilers.

• In general, looking at Table 2, the ratio of the time spent in executing instructions to the time spent in stalls in the user mode is lower for the JIT compiler as compared to the interpreter mode. This behavior is not only due to the code installation in JIT compiler that increases stall times, but also due to the better data and instruction locality of the interpreter loop compared to the native code execution [2].

• An important difference between the two modes is shown in Figure 3. We observe in five of the applications, the TLB miss rate is much higher in the JIT compiler. As in the previous observation for Table 2, the better locality in the interpreter loop translates to a higher TLB hit rate compared to the JIT compiler. However, the TLB miss rates of db and jack are lower in the JIT compiler which appears somewhat counter-intuitive. A detailed investigation of the interpreter behavior for these codes leads us to the following reasoning. Benchmark db has a few methods (658 unique methods called 91753107 times) that execute repeatedly while accessing several data items at random to perform database transactions. In both modes, the locality for these data items is relatively poor. This poor data locality also interferes with the access locality for the bytecode stream in the interpreter mode, where the bytecodes are treated as data by the interpreter loop. Since method reuse for the bytecode stream is high for db, this effect is amplified resulting in a higher TLB miss rate in the interpreter mode. While executing jack, a significant amount of time is spent in user and kernel stalls when reading 17 files during execution. This overshadows any difference in TLB miss behavior between both style of execution. The differences in TLB miss rate automatically translate to the differences in the percentage of kernel execution times between the two modes as shown in Table 2, particularly for those applications where the TLB miss handling overshadows the other kernel services.

We have made all the above observations for the s100 data set of the SPECjvm98 suite. We have also conducted a similar study for the s1 and s10 data sets [31]. Table 5 shows the kernel activities for these applications with the s1 and s10 data set.

An interesting observation is the fact that idle times (due to file reads) can be seen with the smaller data sets. As mentioned earlier, idle times are due to disk activity when the operation misses in the file cache. With the larger data set (s100) that operates on the same set of files/blocks as a smaller data set, the longer execution time makes it difficult to give any prominence to the idle times in Figure

1. In most applications, the operation is invoked repeatedly to the same files/blocks leading to a higher hit percentage in the file cache while using the s10 and s100 data sets. As a result, we observed that the percentage of kernel time spent in the *read* call goes up for the smaller data sets.

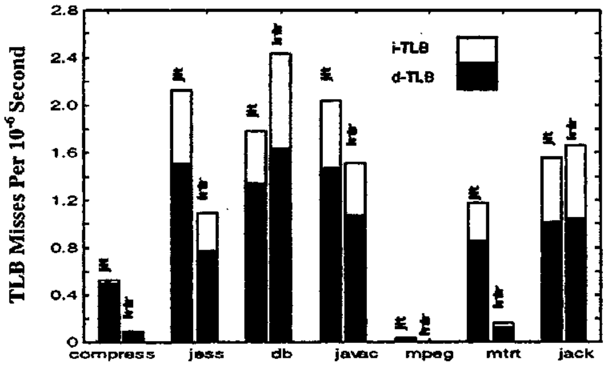


Figure 3. TLB Misses Behavior for SPECjvm98 Benchmarks

Table 5. Execution Time Breakdown for s1 and s10 (with JIT only)

Bench marks	Input	User	User Inst.	User Stall	Kernel	Kernel Inst.	Kernel Stall	Kernel Sync.	Idle
comp	S1	92.25	87.13	5.12	6.06	4.67	1.20	0.19	1.69
	S10	83.57	78.50	5.07	5.44	4.31	0.97	0.16	10.99
jess	S1	61.95	51.49	10.46	30.28	21.71	6.50	2.07	7.77
	S10	79.10	70.70	8.40	16.99	13.61	2.66	0.72	3.91
db	S1	52.07	44.19	7.88	30.91	20.12	8.23	2.56	17.02
	S10	79.08	70.45	8.63	15.89	12.69	2.45	0.75	5.03
javac	S1	71.18	62.08	9.10	18.56	12.17	5.13	1.26	10.26
	S10	73.06	62.50	10.56	11.99	9.89	1.82	0.28	14.95
mpeg	S1	81.04	76.12	4.92	10.92	6.79	3.27	0.86	8.04
	S10	96.21	93.88	2.33	2.25	1.41	0.70	0.14	1.54
mtrt	S1	89.99	81.23	8.76	7.27	5.08	1.87	0.32	2.74
	S10	91.98	82.50	9.48	6.71	5.37	1.18	0.16	1.31
jack	S1	80.53	70.34	10.19	17.36	13.31	3.46	0.59	2.11
	S10	81.47	71.34	10.13	17.27	13.46	3.30	0.51	1.26

3.2 TLB Miss Activity in Different Parts of the JVM

Since the TLB miss handler contributes to most of the kernel activity for both the JVM execution modes, we investigated the TLB characteristics in more detail to analyze the causes for the TLB misses. The TLB misses (See Table 6) that occurred during JVM execution while executing user code were split into those that occurred during class loading, garbage collection and dynamic compilation (for JIT mode only). Further, the misses in the unified TLB are split into those resulting from instruction or data accesses. The results for this section was performed with the s1 data set (due to the time-consuming nature of selective profiling). We believe that the TLB behavior in these parts of the JVM due to Java's unique characteristics still hold good for the s100 data set. However, the percentage contribution of TLB misses due to dynamic compilation and loading will decrease with increased method reuse in the s100 data set. The key observations from this study are summarized below:

- The TLB miss rate due to data accesses is higher than that due to instruction accesses in both the execution modes. This suggests that the instruction access locality is better than that of data accesses and is consistent with the cache locality behavior exhibited by SPECjvm98 benchmarks [2]. It is also observed that this behavior is true for the different parts of the JVM execution studied. In particular, this behavior is accentuated for the garbage collection phase, where the data miss rate is an order of magnitude or more than the instruction miss rate. This is due to the garbage collection of objects that are scattered in the heap space by a relatively small code sequence.
- The TLB locality is better for the interpreter mode as compared to the JIT mode for both instruction and data accesses. One of the reasons for such a behavior is the higher miss rate during dynamic compilation (due to both instruction and data accesses) as compared to that of the overall TLB miss rate in JIT compiler mode. For example, the miss rate due to instruction and data accesses during dynamic compilation of db is 0.24% and 0.67% as compared to the 0.17% overall TLB miss rate. It must be observed that the relative TLB miss rates of db in the interpreter and JIT compiler mode when using the s1 data set is different from that of the s100 data set, due to the reduced method reuse. The main reason for the poor TLB locality during dynamic compilation is due to the drastic change in working set of the JVM. The code and data structures associated with the dynamic compiler are quite different from that of the rest of the JVM execution. Further, dynamic compilation also incurs TLB misses when codes are installed in new locations for the first time. Another reason for the better performance of the interpreter is due to the better locality in the interpreter loop.
- Generally, the TLB miss rates for data accesses is the higher than the overall miss rates for each of the three instrumented parts of the JVM for both the modes. In particular, the TLB data miss rates are the highest during garbage collection. Thus, the frequent use of garbage collectors in memory-constrained systems will impact the overall JVM performance more significantly. In the JIT mode, the TLB instruction miss rate while dynamic loading is typically less than the overall instruction miss rate. But for the interpreter, the loading instruction miss rate is almost always higher than the general instruction miss rate. This behavior is more due to the increased instruction locality within the interpreter rather than a change in behavior of the loading routine itself.

3.3 Comparison with SPECInt Behavior

Next, we set out to study whether the kernel activity of the SPECjvm98 benchmarks is different from that of traditional applications. Figure 4 shows the most frequently used kernel routines in four benchmarks from the SPECInt95 suite.

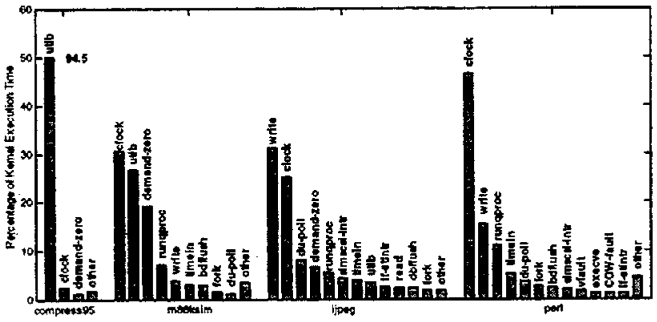


Figure 4. Kernel Routines Distribution for SPECInt95 Benchmarks

It is observed that the TLB miss handler that constituted a major part of the kernel activity in all SPECjvm98 benchmarks is quite insignificant for the `ijpeg` and `perl` benchmark. It was observed earlier that features of Java such as class loading, garbage collection, and dynamic loading affect the TLB even if the application exhibits good TLB locality. However, the application behavior plays an important role in the kernel activity as well. For example, *utlb* activity contributes to 94.5% of kernel activity when executing compress95 and 80% of kernel activity in JIT mode when executing compress (SPECjvm98 suite). Though, compress95 and compress implement essentially the same algorithm, there are differences in their data sets. These differences along with the dynamic class loading behavior of compress result in a higher percentage of kernel activity in the *read* routine. Further, we find that the SPECjvm98 benchmarks spend 10% of their execution time, on an average, in the kernel, in contrast to the four SPECint95 benchmarks that spend less than 2% of their execution in the kernel.

Table 6. Instruction and Data References and Misses within a Fully Associative, 64 Entry, Unified TLB with LRU Replacement Policy (with s1 dataset)

The TLB misses that occur during JVM execution using s1 dataset are split into those due to data accesses (Data) and instruction accesses (Inst). Further, we decompose the TLB misses based on the part of the JVM executed: class loading, garbage collection and dynamic compilation. The first row for each benchmark and execution mode gives the number of TLB misses in that part, and the second row gives the miss rate within that part of the JVM (Num.: Number of Misses, MR: Miss Ratio).

Benchmarks			Total		Loading		GC		Compilation		Other		Total
			Inst	Data	Inst	Data	Inst	Data	Inst	Data	Inst	data	Number
comp	jit	Num.	13006215	90164039	1924105	13224248	2892209	20555146	1820962	12184675	6368939	44199970	103170254
		MR	0.0135%	0.3526%	0.0128%	0.3309%	0.0109%	0.2911%	0.0167%	0.4215%	0.0145%	0.3804%	0.0846%
	intr	Num.	3621486	38088720	584432	5884113	1288338	13559689	---	---	1748716	18644918	41710206
		MR	0.0054%	0.2001%	0.0056%	0.1994%	0.0052%	0.1924%	---	---	0.0055%	0.2064%	0.0484%
jess	jit	Num.	503314	535910	15182	40626	512	12737	218118	163244	269502	319303	1039224
		MR	0.1290%	0.5501%	0.0491%	0.6150%	0.0096%	1.0429%	0.1771%	0.5239%	0.1168%	0.5464%	0.2131%
	intr	Num.	910696	2353513	15151	44031	520	12147	---	---	895025	2297335	3264209
		MR	0.0438%	0.4247%	0.0467%	0.6291%	0.0084%	0.7924%	---	---	0.0438%	0.4211%	0.1240%
db	jit	Num.	250358	236619	8782	23249	346	5759	129916	89773	111314	117838	486977
		MR	0.1086%	0.4054%	0.0510%	0.5587%	0.0119%	0.9199%	0.2419%	0.6706%	0.0711%	0.2932%	0.1686%
	intr	Num.	213813	421717	8725	23782	301	5224	---	---	204787	392711	635530
		MR	0.0496%	0.3658%	0.0508%	0.5726%	0.0097%	0.7258%	---	---	0.0499%	0.3557%	0.1164%
javac	jit	Num.	476746	514612	6206	46992	780	19541	180341	131670	279419	316409	991358
		MR	0.1196%	0.52817%	0.0431%	0.6121%	0.0077%	0.8295%	0.2490%	0.7250%	0.1004%	0.4570%	0.1999%
	intr	Num.	570349	1310814	16186	49790	108754	336362	---	---	445409	924662	1881163
		MR	0.0431%	0.3767%	0.0416%	0.6207%	0.0310%	0.3596%	---	---	0.0476%	0.3753%	0.1125%
mpeg	jit	Num.	310681	293071	12865	28360	374	6920	159838	111828	137604	145963	603752
		MR	0.0625%	0.2128%	0.0593%	0.5511%	0.0110%	0.9277%	0.1698%	0.4780%	0.0364%	0.1346%	0.0951%
	intr	Num.	226424	1312574	10676	27866	312	6219	---	---	215436	1278489	1538998
		MR	0.0041%	0.0832%	0.0526%	0.5840%	0.0086%	0.7131%	---	---	0.0039%	0.0813%	0.0219%
mtrt	jit	Num.	749105	1084365	9458	25102	920	36481	174117	122187	564610	900595	1833470
		MR	0.0224%	0.1258%	0.0498%	0.5590%	0.0016%	0.2550%	0.2486%	0.6847%	0.0177%	0.1092%	0.0437%
	intr	Num.	963364	4126122	9525	25958	901	35296	---	---	952938	4064868	5089486
		MR	0.0059%	0.0953%	0.0502%	0.5793%	0.0014%	0.2126%	---	---	0.0059%	0.0944%	0.0247%
jack	jit	Num.	1083999	1279611	14859	31418	1207	26056	194008	136337	873925	1085800	2363610
		MR	0.1466%	0.6897%	0.0654%	0.5830%	0.0054%	0.4709%	0.2382%	0.6612%	0.1426%	0.7052%	0.2556%
	intr	Num.	3191125	11725770	11415	29253	1183	23986	---	---	3178527	11672531	14916895
		MR	0.0267%	0.3618%	0.0543%	0.5864%	0.0046%	0.3358%	---	---	0.0267%	0.3615%	0.0982%

4. INSTRUCTION LEVEL PARALLELISM IN JAVA WORKLOADS

This section analyzes the impact of instruction level parallelism (ILP) techniques on SPECjvm98 suite by executing the complete workload on the detailed superscalar CPU simulator MXS. The effectiveness of microarchitectural features such as wide issuing and retirement and multi-ported cache are studied. Additionally, we investigate the bounds on available parallelism in Java workloads by studying the nature of dependencies between instructions and computing the program parallelism. Due to the large slowdown of MXS CPU simulator, we use the reduced data size s1 as the data input in this section. Just as before, we model instruction and data accesses in both application and OS.

4.1 Effectiveness of Aggressive ILP Architectures

Figure 5 illustrates the kernel, user, and aggregate execution speedup for a single pipelined (SP), a four-issue superscalar (SS) and an eight-issue superscalar microprocessor (normalized to the corresponding execution time on the SP system). Our four-issue SS simulates the machine model described in section 2. The eight-issue SS uses more aggressive hardware to exploit ILP. Its instruction window and reorder buffer can hold 128 instructions, the load/store queue can hold 64 instructions, and the branch prediction table has 2048 entries. Furthermore, its L1 caches support up to four cache accesses per cycle. To focus the study on the performance of the CPU, there are no other differences on the memory subsystem.

Figure 5 shows that microarchitectural techniques to exploit ILP reduce the execution time of all SPECjvm98 workloads on the four-issue SS. The total ILP speedup (in JIT mode), nevertheless, shows a wide variation (from 1.66x in jess to 2.05x in mtrt). The average ILP speedup for the original applications is 1.81x (for user and kernel integrated). We see that kernel speedup (average 1.44x) on an ILP processor is somewhat lower than that of the speedup for user code (average 2.14x). When the issue width is increased from four to eight, we observe a factor of less than 1.2x on performance improvement for all of SPECjvm98 applications. Compared with the 1.6x (in SPECInt95) and 2.4x (in SPECfp95) performance gains obtained from wider issuing and retirement [29], our results suggest that aggressive ILP techniques are less efficient for SPECjvm98 applications than for workloads such as SPEC95. Several features of SPECjvm98 workloads help explain this poor speedup: The stack based ISA results in tight dependencies between instructions. Also, the execution of SPEC Java workloads, which involve JIT compiler, runtime libraries and OS, tends to contain more indirect branches to runtime library routines and OS calls, and exceptions. The benchmark db has a significant idle component in the s1 data set, which causes the aggregate IPC to be low although both kernel and user code individually exploit reasonable ILP.

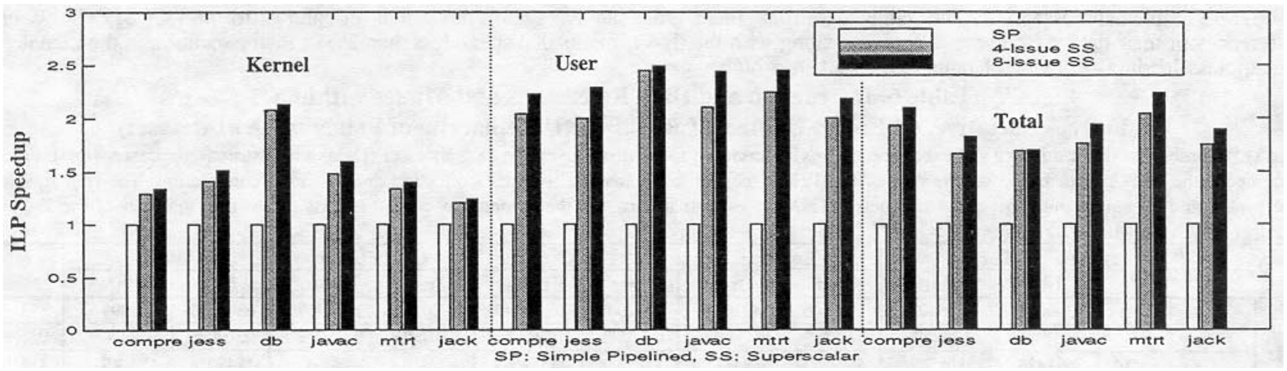
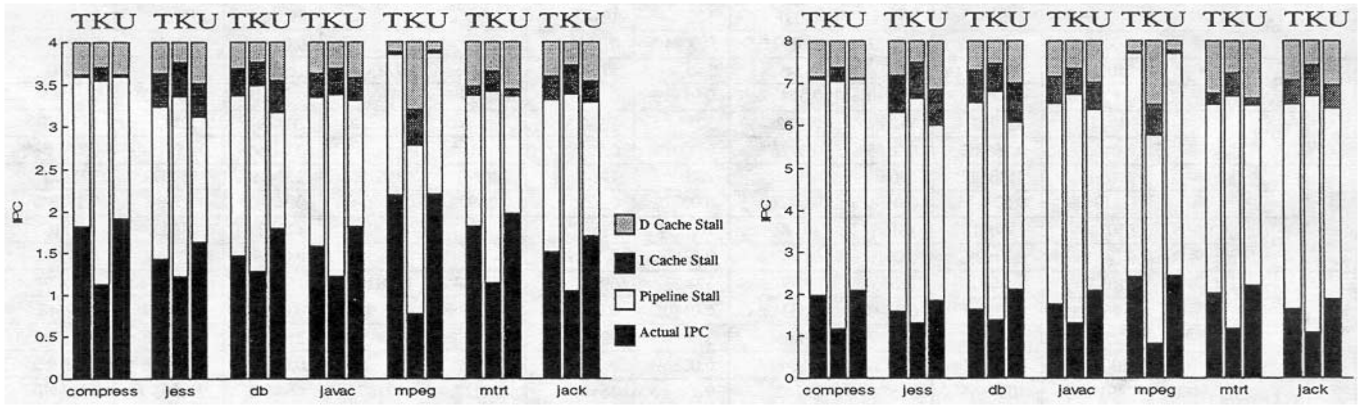


Figure 5. ILP Speedup (with JIT)



(a) 4-issue (b) 8-issue

Figure 6. IPC Breakdown for 4-issue and 8-issue Superscalar Processors (T: Total; K: Kernel; U: User, with s1 Dataset and JIT Compiler)

4.2 Breakdown of IPC - Where have the Cycles Gone?

To give a more detailed insight, we breakdown the ideal IPC into actual IPC achieved, IPC lost on instruction and data cache stall, and IPC lost on pipeline stall. We use the classification techniques described in [12, 29] to attribute graduation unit stall time to different categories: a data cache stall happens when the graduation unit is stalled by a load or store which has a outstanding miss in data cache. If the entire instruction window is empty and the fetch unit is stalled on an instruction cache miss, an instruction cache stall is recorded. Other stalls, which are normally caused by pipeline dependencies, are attributed to pipeline stall. Figure 6 shows the breakdown of IPCs on four-issue and eight-issue superscalar processors.

On four-issue superscalar microprocessor, we see `jess`, `db`, `javac` and `jack` lost more IPC on instruction cache stall. This is partially due to high indirect branch frequency which tends to interrupt control flow. These four programs contain 22% to 28% indirect branches. All studied applications show some IPC loss on data cache stall. The data cache stall time includes misses for bytecodes during compilation by the JIT compiler and those during the actual execution of compiled code on a given data set. Figure 6 shows that a significant amount of IPC is lost due to pipeline stalls and the IPC loss in pipeline stall on an eight-issue processor is more significant than that of four-issue processor. This fact implies that the more aggressive and complex ILP hardware may not achieve the desired performance gains on SPECjvm98 due to the inherent ILP limitation of these applications. All applications show limited increase in instruction cache IPC stall and data cache IPC stall on eight-issue processor.

4.3 Impact of Multi-Ported Cache

Speculative, dynamically scheduled microprocessors require high instruction and data memory bandwidth. A viable solution is to employ multiple cache ports that can be accessed simultaneously by the processor's load/store unit. This section characterizes the performance benefits of multi-ported cache on SPECjvm98 applications. We present the performance of a 32KB multi-ported cache for each benchmark and on both user and kernel mode (with `jit`), as shown in Figure 7. In user mode, increasing the number of cache ports from one to two shows 31% of improvement in performance for `compress` and `mpeg`, 22% of improvement for `javac` and `jack`, and 20% of improvement for `jess`, `db` and `mtrt`. The average performance improvement is 24%, which is approximately equivalent to the number for SPECInt95 reported in [36]. The average performance improvement for an increase of two to three and of three to four cache ports is 5% and 1% respectively.

In kernel mode, we did not observe notable IPC speedup by increasing cache ports from one to four. The average additional improvement in performance for 2, 3 and 4 cache ports is 8%, 3% and 1% respectively. Our results suggest that a dynamic superscalar processor with two-port cache is cost-effective on SPECjvm98 workloads. While the impact of multiple cache ports on Java user code is not different from that of SPECInt applications studied in [36], it is worth noting that the kernel code benefits less from 2 ports cache than user code.

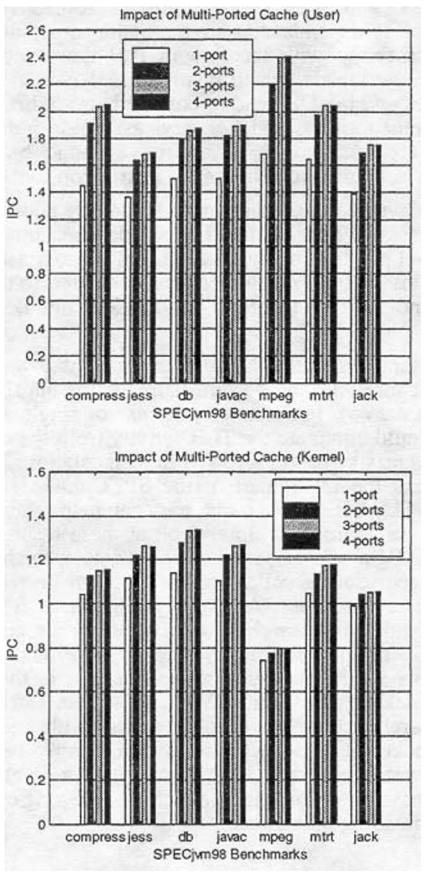


Figure 7. Impact of Multi-Ported Cache (with JIT Compiler and use s1 Dataset)

4.4 Limits of Available Parallelism

In order to understand the instruction level parallelism issues involving the stack-oriented Java code, we investigated the limits of available parallelism in Java workloads. We also compare the ILP of the Java benchmarks to SPECInt95 applications, and several C++ programs. This work is focused on logical limits rather than implementation issues and towards this end, we assume an ideal machine with infinite machine level parallelism (MLP). MLP is defined to be the maximum number of instructions that the machine is capable of scheduling in a single cycle.

We use dynamic dependence analysis in order to compute the limits of ILP as in previous parallelism investigations [32, 33]. First, we construct a Dynamic Dependency Graph (DDG) which is a partially ordered, directed, and acyclic graph, representing the execution of a program for a particular input. The executed operations comprise the nodes of the graph and the dependencies realized during the execution form the edges of the graph. The edges in the DDG force a specific order on the execution of dependent operations – forming the complete DDG into a weak ordering of the program's required operations. A DDG, which contains only data dependencies, and thus is not constrained by any resource or control limitations, is called a dynamic data flow graph. It lacks the total order of execution found in the serial stream; all that remains is the weakest partial order that will successfully perform the computations required by the algorithms used. If a machine were constructed to optimally execute the DDG, its performance would represent an upper bound on the performance attainable for the program. In our study, first, the critical path length defined as the height of the scheduled DDG (the absolute minimum number of steps required to evaluate the operations in the scheduled DDG) is determined. The available parallelism is computed by dividing the total number of nodes by the critical path length.

To give an upper bound on the available parallelism, an available Machine Level Parallelism (MLP) of infinity was considered but MLP of 8, 16, 32, 64 and 128 were also studied for comparative purposes (See Table 7). We consider only true dependencies (or RAW dependencies) while scheduling instructions, this is the absolute limit of parallelism that can potentially be exploited from that program, with the best of renaming, etc. The latency of all operations is set to be 1 cycle. Perfect memory disambiguation and perfect branch prediction are assumed. More details on these experiments can be found in [34]. In our study, the SPECjvm98 benchmarks are invoked with an interpreter.

Table 7 shows that Java code exhibits very low ILP in comparison to all other workloads analyzed. The average available parallelism (in terms of the harmonic mean of the observations) of the four different suites of programs for different window sizes is summarized in Figure 8.

Table 7. Available Instruction Level Parallelism of Different Benchmarks (SPECjvm98 Benchmarks are Invoked with an Interpreter)

Benchmarks		Available ILP					
		MLP= 8	MLP= 16	MLP= 32	MLP= 64	MLP= 128	MLP= Infinity
SPEC Int95	`compress`	6.94	13.68	21.9	37.65	65.89	265.01
	`gcc`	7.26	13.94	25.56	40.79	58.15	92.14
	`go`	7.15	14.11	27.26	48.79	70.23	83.81
	`li`	7.47	14.25	25.35	47.17	70.15	72.48
	`m88ksim`	6.45	11.09	21.50	45.59	104.43	133.36
	`ijpeg`	7.44	14.67	28.46	48.01	71.86	8465.51
C++	`deltablue`	7.28	15.15	30.79	60.48	111.28	270.97
	`eqn`	7.42	14.18	26.54	37.96	43.51	272.09
	`idl`	7.57	14.59	29.05	48.31	63.49	277.01
	`ixx`	7.2	13.83	26.92	51.32	76.05	111.71
	`richards`	7.56	13.78	24.83	35.84	55.99	125.57
SPEC Jvm98	`db`	6.75	11.59	19.54	28.55	33.7	41.06
	`javac`	6.66	10.67	15.55	19.82	21.89	25.13
	`jess`	6.69	11.02	16.17	19.54	20.78	22.83
	`mpeg`	7.27	10.11	12.86	14.63	15.19	15.39
	`mtrt`	6.76	10.26	12.92	14.06	14.38	15.82
	`compress`	6.98	9.66	11.46	11.68	11.88	11.95
DSP	`dot`	7.97	15.91	31.82	63.58	127.10	8239.82
	`g711`	7.57	15.39	31.31	62.53	125.86	11064.15
	`autocor`	6.66	15.94	31.86	62.96	125.62	12057.35
	`fir`	6.67	15.69	31.5	63.12	124.35	18964.54
	`audio`	6.79	15.67	31.52	60.35	117.41	21164.94
	`adpcm`	7.26	13:42	24.19	40.37	40.48	40.59

With infinite MLP, the mean ILP is 125 (for the SPECInt95 benchmarks), 175 (for the C++ programs), 20 (for the Java programs) and 240 (for the DSP benchmarks). The extremely low ILP of the Java programs, even with no other control or machine constraints, can be attributed to the stack-based implementation of the Java Virtual Machine (JVM). The stack nature of the JVM imposes a strict ordering on the execution of the bytecodes. This observation is further supported by the behavior of the `compress` benchmark, which is present in both the SPECInt95 and SPECJvm98 suites. Both are text compression programs and the Java version is a Java port of the integer benchmark from SPECCPU95. It can be seen that with an MLP of infinity, the CPU95 `compress` benchmark has the highest ILP among all the SPECInt95 benchmarks while the Java `compress` program has the least ILP among the SPECJvm98 benchmarks.

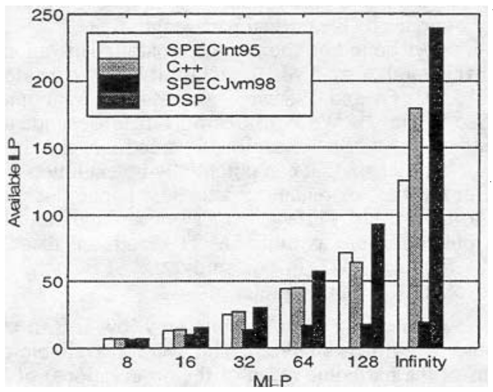


Figure 8. Average Available Parallelism of SPECInt95,C++, SPECjvm98 and DSP benchmarks for Different MLP

5. CONCLUSIONS

The popularity and wide-spread adoption of Java has necessitated the development of an efficient Java Virtual Machine. This study has provided insights into the interaction of the JVM implementations with the underlying system (both hardware and operating system). To our knowledge, this work is the first to characterize the kernel activity of the Java applications and analyze their influence on architectural enhancements. The major findings from this research are:

• The kernel activity of SPECjvm98 applications constitutes up to 17% of the execution time in the large (s100) data set and up to 31% in the small (s1) data set. The average kernel activity in s100 case is approximately 10%, in comparison to around 2% in four SPECInt benchmarks studied. Generally, the JIT compiler mode consumes a larger portion of kernel services during execution. The only exception we found was in db for s100 data set, which has high method reuse and interference of the bytecode and data locality in the interpreter mode.

• The SPECjvm98 benchmarks spend most of their time in executing instructions in the user mode and spend less than 10% of the time in stall cycles during the user execution. The kernel stall mode in all SPECjvm98 benchmarks, except jack that has a significantly higher file activity, is small. However, the MCPI of the kernel execution is found to be much higher than that of the user mode.

• The kernel activity in the SPECjvm98 benchmarks is mainly due to the invocation of the TLB miss handler routine, *utlb* and the *read* service routine. In particular, the *utlb* routine consumes more than 80% of the kernel activity for four out of seven SPECjvm98 benchmarks for the s100 dataset. Further, the dynamic compilation, garbage collection and class loading phases of the JVM execution utilize the *utlb* routine at a faster rate as they exhibit a higher TLB miss rate than the rest of the JVM. It is also observed that the dynamic class loading behavior influences the kernel activity more significantly for smaller datasets (s1 and s10) and increases the contribution of the read service routine.

• The average ILP speedup on a four-issue superscalar processor for the SPECjvm98 benchmarks executed in the JIT compiler mode was found to be 1.81 times. Further it is found that the speedup of the kernel routines (average 1.44 times) is lower than that of the speedup of the user code (average 2.14 times). This behavior can be attributed to the significant fraction of the kernel activity spent in the *utlb* routine that is a highly optimized sequence of interdependent instructions.

• Aggressive ILP techniques such as wider issue and retirement are less effective for SPECjvm98 benchmarks than for SPEC95. We observe that the performance improvement for SPECjvm98, when moving from 4 issue to 8 issue width is 1.2 times as compared to the 1.6 times and 2.4 times performance gains achieved by the SPECint95 and SPECfp95 benchmarks, respectively. The pipeline stalls due to dependencies are the major impediment to achieving higher speedup with increase in ILP issue width. Also, the SPECjvm98 workloads, which involve the dynamic compiler, runtime libraries and the OS, tend to contain more indirect branches to runtime library routines and OS services. These indirect branches that constitute up to 28% of all the dynamic branches in the SPECjvm98 suite tend to increase the instruction cache stall times.

• The SPECjvm98 benchmarks have inherently poor ILP compared to other classes of benchmarks. For an ideal machine with infinite machine level parallelism, the mean ILP is approximately 20 for the Java programs (in interpreted mode) as compared to the speedup of 125, 175 and 240 for the SPECInt95, C++ and DSP benchmark suites.

These results can help in providing insight towards designing systems software and architectural support for enhancing the performance of a JVM. For example, the use of single-address virtual space that could eliminate the TLB activity from the critical path of cache access may be useful for JVM implementations. Since the *utlb* handler forms the major part of the SPECjvm98 kernel activity, moving the TLB access to a cache miss can help reduce overall kernel activity. In addition, a closer look at these results can help restructure the JVM implementation. For example, the high TLB miss rates that occur during collecting garbage can be reduced by collecting objects page wise. A simple modification from depth first search to breadth first search of objects during the mark phase of a garbage collection algorithm may help improve this. The key to an efficient Java virtual machine implementation is the synergy between well-designed software (JVM and system software support), supportive architecture and efficient runtime libraries. While this paper has looked at kernel activities of SPECjvm98 benchmarks, we plan to characterize other Java applications as well, particularly those that have more garbage collection, network communication and multi-threading operations.

REFERENCES

[1] C.-H. A. Hsieh, M. T. Conte, T. L. Johnson, J. C. Gyllenhaal and W. W. Hwu, A Study of the Cache and Branch Performance Issues with Running Java on Current Hardware Platforms, In *Proceedings of COMPCON*, pages 211-216, 1997.

[2] R. Radhakrishnan, N. Vijaykrishnan, L. K. John and A. Sivasubramaniam, Architectural Issue in Java Runtime Systems, In *Proceedings of the 6th International Conference on High Performance Computer Architecture*, pages 387-398, 2000.

[3] N. Vijaykrishnan, N. Ranganathan and R. Gadekarla, Object-Oriented Architectural Support for a Java Processor, In *Proceedings the 12th European Conference on Object-Oriented Programming*, pages 430-455,1998.

[4] N. Vijaykrishnan and N. Ranganathan, Tuning Branch Predictors to Support Virtual Method Invocation in Java, In *Proceedings of the 5th USENIX Conference of Object-Oriented Technologies and Systems*, pages 217-228, 1999.

[5] A. Barisone, F. Bellotti, R. Berta and A. D. Gloria, Ultrasparc Instruction Level Characterization of Java Virtual Machine Workload, from the 2nd Annual Workshop on Workload Characterization, *Workload Characterization for Computer System Design*, Kluwer Academic Publishers, pages 1-24, 1999.

[6] J.-S. Kim and Y. Hsu, Analyzing Memory Reference Traces of Java Programs, from the 2nd Annual Workshop on Workload Characterization, *Workload Characterization for Computer System Design*, Kluwer Academic Publishers, pages 25-48, 1999.

[7] M. O'Connor and M. Tremblay, PicoJava-I: The Java Virtual Machine in Hardware, *IEEE Micro*, pages 45-53, Mar. 1997.

[8] R. Radhakrishnan, J. Rubio and L. John, Characterization of Java Applications at Bytecode and Ultra-SPARC Machine Code Levels, In *Proceedings of IEEE International Conference on Computer Design*, pages 281-284, 1999.

[9] L. A. Barroso, K. Gharachorloo, and E. Bugnion, Memory System Characterization of Commercial Workloads, In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 3-14, 1998.

[10] S. A. Herrod, Using Complete Machine Simulation to Understand Computer System Behavior, *Ph.D. Thesis*, Stanford University, Feb. 1998.

[11] M. Rosenblum, S. A. Herrod, E. Witchel, and A. Gupta, Complete Computer System Simulation: the SimOS Approach, *IEEE Parallel and Distributed Technology: Systems and Applications*, vol.3, no.4, pages 34-43, Winter 1995.

[12] M. Rosenblum, E. Bugnion, S. A.Herrod, E. Witchel, and A. Gupta, The Impact of Architectural Trends on Operating System Performance, In *Proceedings of the 15th ACM Symposium on Operating System Principles*, pages 285-298, 1995.

[13] 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*, pages 90-97, 1996.

[14] T. Cramer, R. Friedman, T. Miller, D. Seberger, R. Wilson and M. Wolczko, Compiling Java Just-In-Time, *IEEE Micro*, vol. 17, pages 36-43, May 1997.

[15] A. Krall, Efficient JavaVM Just-In-Time Compilation, In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 54-61, 1998.

[16] A. Adl-Tabatabai, M. Cierniak, G. Lueh, V. M. Parakh and J. M. Stichnoth, Fast Effective Code Generation in a Just-In-Time Java Compiler, In *Proceedings of Conference on Programming Language Design and Implementation*, pages 280-290, 1998.

[17] H. McGhan and M. O'Connor, PicoJava: A Direct Execution Engine for Java Bytecode , *IEEE Computer*, pages 22-30, Oct. 1998.

[18] N. Vijaykrishnan, Issues in the Design of a Java Processor Architecture. *PhD Thesis*, College of Engineering, University of South Florida, July 1998.

[19] M. C. Merten, A. R. Trick, C. N. George, J. Gyllenhaal, and W. W. Hwu, A Hardware Driven Profiling Scheme for Identifying Program Hot Spots to Support Runtime Optimization, In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 136-147, 1999.

[20] SPEC Jvm98 Benchmarks, *http://www.spec.org/osg/jvm98/*

[21] SPEC CPU95 Benchmarks, *http://www.spec.org/osg/cpu95/*

[22] S. A. Herrod, M. Rosenblum, E. Bugnion, S. Devine, R. Bosch, J. Chapin, K. Govil, D. Teodosiu, E. Witchel, and B. Verghese, The SimOS User Guide, *http://simos.stanford.edu/userguide/*

[23] Overview of Java Platform Product Family, *http://www.javasoft.com/products/OV_jdkProduct.html*

[24] E. Witchel and M. Rosenblum, Embra: Fast and Flexible Machine Simulation, In *Proceedings of ACM SIGMETRICS '96: Conference on Measurement and Modeling of Computer Systems*, 1996.

[25] J. Bennett and M. Flynn, Performance Factors for Superscalar Processors, *Technical Report CSL-TR-95-661*, Computer Systems Laboratory, Stanford University, Feb. 1995.

[26] MIPS Technologies, Incorporated, R10000 Microprocessor Product Overview, MIPS Open RISC Technology, Oct. 1994.

[27] K. C. Yeager, MIPS R10000, *IEEE Micro*, vol.16, no.1, pages 28-40, Apr. 1996.

[28] K. I. Farkas and N. P. Jouppi, Complexity/Performance Tradeoffs with Non-Blocking Loads, In *Proceedings of the 21th International Symposium on Computer Architecture*, pages 211-222, 1994.

[29] K. Olukotun, B.A. Nayfeh, L. Hammond, K. Wilson and K.-Y. Chang, The Case for a Single-Chip Multiprocessor, In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1-4, 1996.

[30] B. Nayfeh, L. Hammond and K. Olukotun, Evaluation of Design Alternatives for a Multiprocessor Microprocessor, In *Proceedings of the 23rd International Symposium on Computer Architecture*, pages 66-77, 1996.

[31] Tao Li, Using Complete System Simulation to Characterize the Execution Behaviors of SPECjvm98 Benchmarks, *http://www.ece.utexas.edu/~tli3/tao-jvm98.ps*

[32] T. M. Austin and G. S. Sohi, Dynamic Dependency Analysis of Ordinary Programs, In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 342-351, 1992.

[33] R. Sathe and M. Franklin, Available Parallelism with Data Value Prediction, In *Proceedings of International Conference on High Performance Computing*, pages 194-201, 1998.

[34] J. Sabarinathan, A Study of Instruction Level Parallelism in Contemporary Computer Applications, *Master Report*, University of Texas at Austin, Dec. 1999.

[35] S. Dieckmann and U. Hölzle, A Study of the Allocation Behavior of the SPECjvm98 Java Benchmarks, In *Proceedings of the 13th European Conference on Object-Oriented Programming*, 1999, Springer Verlag.

[36] K. M. Wilson, K. Olukotun, and M. Rosenblum, Increasing Cache Port Efficiency for Dynamic Superscalar Microprocessors, *In Proceedings of the 23rd International Symposium on Computer Architecture*, pages 147-157,1996.