Improving TLB Energy for Java Applications on JVM

Chinnakrishnan S. Ballapuram
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, Georgia 30332–0250
chinnak@ece.gatech.edu

Hsien-Hsin S. Lee
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, Georgia 30332–0250
leehs@gatech.edu

Abstract—**Java platforms are widely deployed and used ranging from ultra-mobile embedded devices to servers for their portability and security. The TLB, a content addressable memory, can consume a significant power in these systems due to the nature of its associative search mechanism. In this paper, we propose and investigate three different optimizations for the TLB design, aiming to improve its power consumption for Java applications running on top of Java Virtual Machines. Our techniques exploit unique memory reference characteristics demonstrated by the JVM and its interaction with the Java applications running atop. Our first technique J-iTLB shows an average of 12.7% energy reduction in the iTLB with around 1% performance improvement for eliminating conflict misses between the JVM code and the Java application code. The second technique combines the J-iTLB with an object iTLB scheme and achieves an energy savings of 51% with a small 1% performance impact. Our third technique, a read-write partitioned J-dTLB, shows an average of 34% energy savings in the dTLB with 1% performance impact. Finally, when the J-iTLB with an object iTLB is combined with the J-dTLB, we obtained 42% overall TLB energy savings.**

I. Introduction

The concept of "write once, run anywhere" has made the Java platforms and their applications widely adopted from wireless mobile devices to high performance servers. The applications being incorporated into these platforms are also becoming more sophisticated and diversified, as a result, consuming more energy. To extend the battery lifetime and reduce the cost of cooling solutions, energy efficiency is no longer a desired feature but a design constraint. To address the overall energy consumption, it becomes inevitable that a designer needs to consider energy efficiency in all different design stacks including circuits, microarchitecture, compilers, and even programming languages.

The Java language combines different features from different programming paradigms. The features include portability, object-oriented design, multithreading, garbage collection, and exception handling. These rich features come at the expense of decreased performance caused by the additional hardware abstractions. The Java source code is first translated into the architecture-independent bytecodes. These bytecodes can then be executed on any platform that supports an implementation of the Java Virtual Machine (JVM). The JVM insulates Java application from any contact with the underlying hardware. The JVM is an abstract computer implemented on top of a real hardware and operating system to run compiled Java bytecodes.

A JVM can be implemented via the following mechanisms: an interpreter, a Just-In-Time (JIT) compiler, a dynamic compiler, and a direct hardware execution. There are pros and cons with respect to performance and power to each of these techniques. The interaction between the Java program and the JVM for the shared resources also influences the power and performance of the overall system. An interpreter emulates the virtual machine by continuously fetching, decoding, and executing the bytecode until the program completes. Although the size of the interpreter is typically small (within hundreds of kilobytes), the per-bytecode translation can result in low performance. A JIT or a dynamic compiler compiles the bytecodes into the corresponding machine-specific binaries to execute them natively on the machine. Further, the native code can be dynamically optimized using runtime feedback profile with a dynamic compiler, leading to better performance for Java applications. But the performance also depends on the effectiveness of the JIT compiler itself. Today, a sophisticated compiler needs more than hundreds of kilobytes and memory space to translate and optimize the code for higher application performance. The last option is to run the bytecodes directly on a Java processor. This eliminates the abstraction of the software translation and promise the best possible performance. In this paper, we concentrate on the JIT and dynamic compiler approaches for their prevalence in practice.

The Java programming language offers features such as strong type checking and dynamic garbage collection with the extra layer of the JVM implementation that impacts both power and performance. In the embedded domain where Java is widely deployed with virtual memory support, power efficiency becomes even more important. Prior studies have shown that the memory behavior of Java applications is very different from regular C/C++ programs [9], [15], [20], [22]. In particular, the characteristics become very interesting when the compiler, optimizer, and garbage collector of a JVM interacts with the Java bytecodes within the same address space.

The instruction TLB (iTLB) and the data TLB (dTLB) are accessed by the JVM code, its associated data, and also by the Java application code and data. Due to the interference of these assorted accesses, the TLB power and performance of Java applications are unexpectedly exacerbated. In this paper, we propose three TLB mechanisms to improve power and performance for Java applications. The contributions of this paper are:

- We study the memory reference characteristics of a Java application running on a JVM and its interactions between them.
- We propose an energy efficient iTLB and dTLB mechanisms for the Java applications running on the JVM without compromising performance.
- Our split iTLB technique achieves an energy savings of 12.7% with 1% performance improvement. Our second technique combines a small object iTLB at first level with the first technique and reduces energy by 51% with 1% performance impact. The third technique splits dTLB reduces energy by 34% with 1% performance impact for the SPECjvm98 benchmark suite. Finally, when the J-iTLB with object iTLB is combined with J-dTLB, we obtained 42% overall TLB energy savings with less than 1% performance impact.

II. Motivation

To gain better understandings regarding the interaction of memory accesses between the JVM and the Java applications, we briefly discuss the way JVM loads a Java program and executes it in the interpreter, JIT, and dynamic compilation mode.

A. Interpreter Mode

In the interpreter mode, the JVM first loads the Java class in the dynamic heap memory. The JVM will then fetch, decode, and execute the Java application bytecode that flows through the dTLB and data cache (dCache). So, the dTLB and dCache are accessed by both the JVM's private data and the Java application bytecodes.

B. Just-In-Time and Dynamic Compilation Mode

In the Just-In-Time mode, the JVM first loads the Java class into the heap memory. A JVM compiler first compiles the bytecode to the native machine binaries of the target processor. In the dynamic compilation mode, the compiler may initially generate an unoptimized code. At runtime, the dynamic optimizer will continuously optimize the binaries with dynamic information. In both modes, the compiled code is stored in the heap space, which will always access the dTLB and dCache. In addition, the JIT or dynamic compiler that optimizes the code during runtime will also go through the dTLB and dCache as part of its own private data accesses.

Once the optimized/unoptimized code is written to the heap through dTLB and dCache, the native code on the heap will be executed through the iTLB and the instruction cache (iCache), so does the code of the JVM itself. This behavior is quite different from the normal execution of a C/C++ program, where instructions are only accessed within the code space. In the JIT or dynamic compilation mode, the JVM code will access the heap memory to fetch the Java application's code. As such, the iTLB is accessed by both the JVM and the Java application. This cross-sharing creates interference in the iTLB, increasing the iTLB miss rate, as the heap accesses for Java applications are dynamic and active. The activity is even busier in the dynamic compiler mode since the dynamic JVM optimizer will attempt to optimize the code at runtime while these accesses will potentially conflict with its own instructions in the iTLB and iCache, penalizing performance of both the Java applications and the JVM itself.

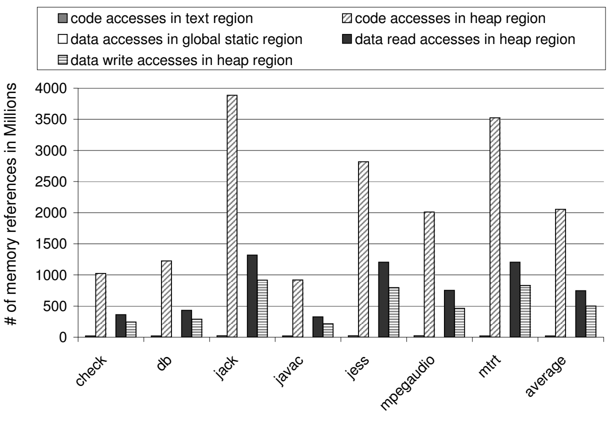

Fig. 1. Distribution of instruction and data memory references to different memory regions by the Java application running on a JVM

Fig. 1 shows the distribution of instruction and data accesses to different regions of the virtual address space for the seven SPECjvm98 [3] benchmark programs using input set s1. The distribution information was collected using the Dynamic SimpleScalar [1], [10] simulator simulating the Jikes RVM [2] running the Java applications. We modified the Dynamic SimpleScalar simulator to identify the regions of memory accesses and collect the distribution profile.

In this figure, the first two bars are plotted for the instruction side while the rest of the three are plotted for the data side. The leftmost bar shows the number of memory accesses to the code region followed by a bar showing the number of instruction accesses to the heap. The JIT compiler generates and writes the machine-dependent native instructions to the heap. These native instructions are read from the heap to execute the Java application. Notice that these heap accesses are dominant in the instruction side, accounting for almost all the instructions accessed.

The next three bars show the distribution of data memory accesses. The third bar shows the number of accesses to the static global region, which is apparently insignificant. The last two bars show the numbers of "data" reads from and writes to the heap. As mentioned earlier, part of these accesses are caused by the activities that the JIT compiler reads the Java bytecode and generates native code that is written onto the heap. We will exploit these memory access characteristics between the JVM and the Java applications for achieving an energy-efficient TLB design.

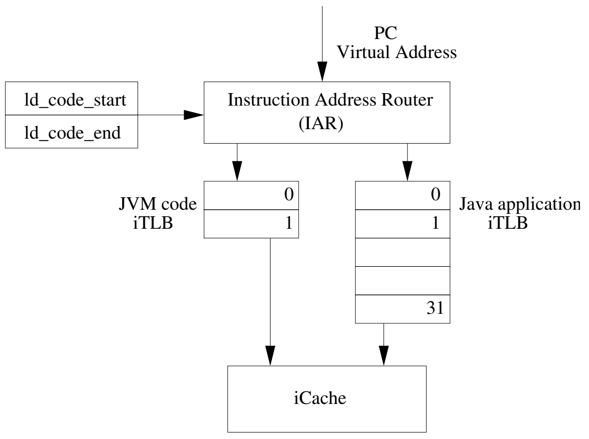

Fig. 2. JVM and Java iTLB (J-iTLB)

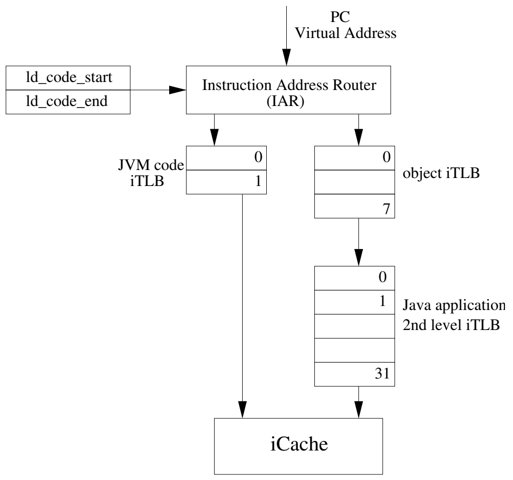

Fig. 3. JVM and Java iTLB (J-iTLB) with object iTLB

III. JVM and Java iTLB (J-iTLB)

As indicated in the second bar ofFig. 1, most of the instructions that go through the iTLB are in fact coming from the heap. These heap accesses are to fetch the native code and execute it. Unfortunately, these highly active accesses can adversely conflict with the normal JVM's code accesses, increasing the iTLB misses in the traditional iTLB structure. To address this interference issue, we propose a new structure that segregates the iTLB into two distinct TLB structures to improve both performance and power. Using two distinct iTLBs, the JVM code and the JIT-compiled application code will be stored separately to eliminate the iTLB interference completely. Also, as the JVM code itself is static, the requirement of its iTLB can be designed much smaller than one holding the JVM-compiled code. On the other hand, Java application codes are much more dynamic since the objects are created and freed at runtime in the Java programs. Worse yet, the unused or deallocated objects can be highly fragmented before they are recycled by the garbage collector. As a result, new objects will be less likely to be allocated within the same memory page or in consecutive pages, causing more TLB misses.

Fig. 2 illustrates our proposed iTLB organization in which the iTLB is horizontally split into two TLBs: one dedicated for the JVM code accesses, and the other for the Java application accesses to the heap. Not only does such segregation eliminate the conflict misses of address translations from two unrelated code space, it also improves power consumption when only one of the TLBs is looked up. During the program's start, the loader identifies the ld_code_start address (code start) and ld_code_end (code end) of the JVM. This information is readily available as part of the ELF or COFF executable format. The sizes of various sections such as .text, .init, .data, .rodata, and .bss sections are clearly defined in the executable file. These two addresses are then kept by a special hardware register pair stored inside the Instruction Address Router (IAR) as shown in the figure. The IAR will route each of the incoming PC virtual address based on this register pair. All incoming PC virtual addresses are checked against the register pair. If an address falls in the JVM's text range, it is routed to the JVM iTLB; otherwise, it is forwarded to the Java application iTLB. Also, the outcome of these comparisons by the IAR is used to clock-gate the unused iTLB for power savings.

A typical Java program creates many objects interacting with other objects using the methods. Once the object completes the work, its resources are released and re-allocated for other objects. These objects generally access both code and data memory regions, and are typically small with good locality and short life span [13]. In fact, these short-lived objects constitute a high percentage of the total memory references. Given the Java objects are small and dynamic, we introduce an object iTLB to exploit this behavior for reducing power.

Fig. 3 shows our proposed iTLB organization. In addition to the partitioned scheme shown in Fig. 2, the iTLB for the Java application code is further stratified into two levels. The first level, a smaller TLB, is inserted to support the dynamic object code accesses in the Java applications. It is then backed up by a second-level, larger TLB to improve the TLB miss rate. Again, the IAR routes the addresses based on the virtual address as explained earlier. We charge an additional extra cycle when it misses the object iTLB and looks up the second-level iTLB. We will analyze the power and performance of these two iTLB organizations in the experiments sections.

IV. JVM and Java dTLB (J-dTLB)

As shown in the last three bars of Fig. 1, the numbers of reads and writes to the heap account for the majority of data memory accesses in Java applications, dwarfing the number of reads in the global static data region. The JIT compiler translates the Java application bytecode into the native code and writes it in the heap space. These writes flow through the dTLB and the dCache. Finally, the underlying hardware reads the translated native code through the iTLB and the iCache.

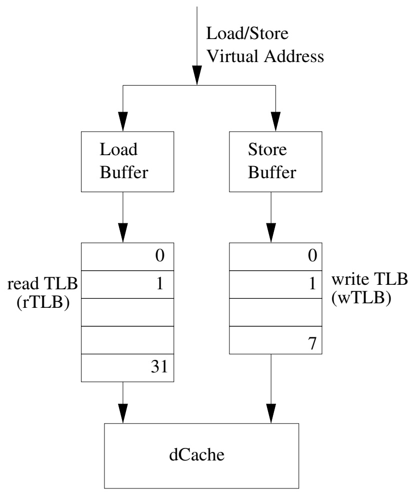

Fig. 4. JVM and Java dTLB (J-dTLB)

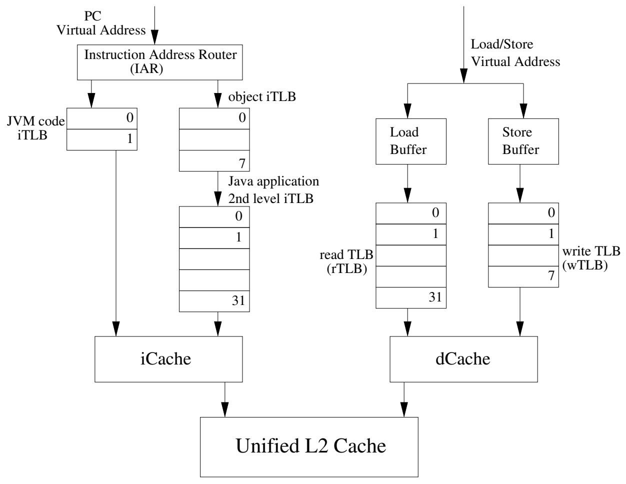

Fig. 5. Complete organization of iTLB and dTLB

To exploit the characteristics of data accesses by Java applications, we propose to modify the dTLB as shown in Fig. 4. In this new organization, the dTLB is split into two distinct TLBs, one for reads and one for writes. All load addresses are directed to the read TLB (rTLB) while all store addresses are routed to the write TLB (wTLB) to minimize the contention between them. Also, when the Java application is in execution, the JIT or dynamic compiler uses the wTLB for native code writes while the rTLB is used by the JVM and the Java application data accesses to read the data from the heap simultaneously without the need to multi-port a monolithic dTLB. Fig. 5 shows the complete schematic of our proposed energy efficient TLB structure for Java applications running on JVM.

V. Experimental results

Our power and performance evaluation infrastructure is based on Dynamic Simplescalar (DSS) [1] [10] simulator. The Dynamic Simplescalar simulator simulates Java programs running on a JVM that uses Just-In-Time or dynamic compilation. The Dynamic Simplescalar allows JIT compilers such as Jikes RVM to be simulated on top of it. The DSS also supports many other interesting features such as PowerPC ISA target, checkpointing, a vastly improved memory model, and a Wattch-based power model [6] [8] using 100nm technology. As the DSS simulates PowerPC ISA as the underlying target machine, we used GNU cross compiler tools and utilities to generate the Jikes RVM PowerPC binary, and the RVM code and data images on a Linux hosted Intel IA-32 machine. The Jikes RVM developed by IBM researchers includes an aggressive optimizing compiler and a flexible dynamic adaptive compilation infrastructure.

The SPECjvm98 benchmark suite [3] consists of eight programs and most of them are derived from real world applications. The SPECjvm98 allows users to evaluate the performance of both the hardware and the software aspects of a JVM platform. On the software side, it evaluates the performance of the JVM, JIT, and the operating system implementations. On the hardware side, it includes the CPU, the caches, and the memory subsystem. We simulated all the Java applications in the SPECjvm98 suite except *compress*, which had problems when running on Dynamic SimpleScalar. We ran all the programs in the SPECjvm98 from start to its completion without fast forwarding or skipping instructions. The simulated processor configuration is shown in TABLE I.

32-bit Processor Parameters	**Values**
Execution Engine	in-order
Number of baseline ITLB entries	32
Number of baseline DTLB entries	32
Number of JVM ITLB entries	2
Number of object ITLB entries	8
Number of 2nd level ITLB entries	32
Number of read DTLB entries	32
Number of write DTLB entries	8
Page size	4 KB
L1/L2 cache hit latency	1 / 6 cycle
Memory latency	150 cycles
TLB hit latency	1 cycle
L1I and L1D cache baseline	4-way associative 32KB, 32B line
L2 cache	4-way 512KB, 32B line
Number of TLB ports used	1
Each 20-bit comparator power	300μW

TABLE I
Processor model parameters

First, we evaluated the effectiveness of the new J-iTLB structure as proposed in Fig. 2. The power savings for this configuration is shown in the leftmost bar of Fig. 6. In this experiment, we focus on the iTLB and assume a 32-entry conventional dTLB for all the experiments. The extra energy consumed by the two comparators in the Instruction Address Router is taxed every cycle. Each comparator consumes 300μW using 100nm technology based on SPICE modeling. The energy savings is around 12.7% without any performance penalty. In fact, there is slight performance improvement around 1%. The reason is that the interference between the

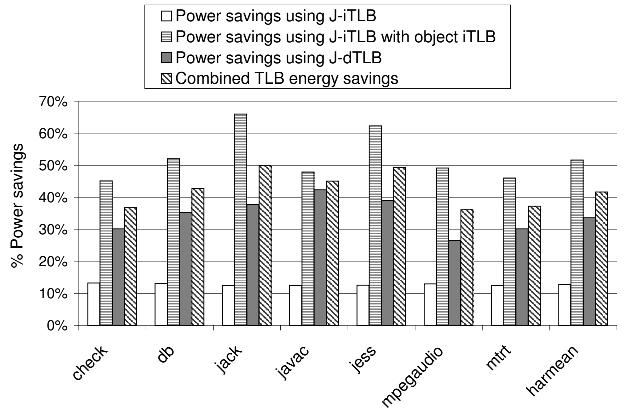

Fig. 6. Power savings using J-iTLB, J-iTLB with object iTLB, and J-dTLB

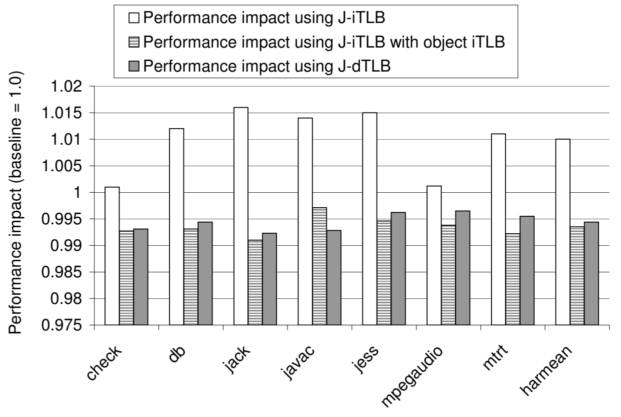

Fig. 7. Performance impact of J-iTLB, J-iTLB with object iTLB, and J-dTLB

JVM and the Java application accesses is eliminated in the new partitioned TLBs. The performance improvement is shown in the leftmost bar of Fig. 7.

The second experiment is based on the TLB configuration depicted in Fig. 3, where the J-iTLB is combined with a first level object iTLB. This configuration achieved higher energy savings of 51% as shown in the second bar of Fig. 6. We charge an extra cycle for the object iTLB misses that access the second level iTLB. The performance impact is around 1% as shown in the corresponding bar of Fig. 7.

The third experiment we conducted evaluates the power and performance impact of the segregated d-TLB configuration proposed in Fig. 4. The third bar in Fig. 6 shows the power savings achieved using the split dTLB for reads and writes. The power savings is around 34% on average. As the effective number of dTLBs are now divided, they cause around 1% performance impact. The performance impact is shown in the rightmost bar of Fig. 7.

We finally combined the J-iTLB with object iTLB configuration and the J-dTLB configuration to evaluate the overall TLB power savings in the processor. In this combined configuration, we obtained 42% overall TLB power savings as shown in the rightmost bar of Fig. 6 with less than 1% performance penalty.

VI. Related work

A number of TLB power reduction schemes have been proposed for the regular C/C++ applications. Kadayif *et al.* [12] added a register called Current Frame Register (CFR) to the instruction address translation. Instead of looking up the i-TLB, the processor fetches the translated address from the CFR unless there is a memory page change. Way-prediction [11] was proposed to reduce cache energy by speculating one predicted way instead of looking up all ways in a set-associative cache. Other approaches for TLB power reduction include selective filter-bank TLB [17], semantic-aware partitioned TLB [16], low-power TLB exploiting synonymous addresses [5], and entropy-based TLB [4].

Ramesh et al. [19] studied the characteristics of Java applications and SPECjvm98 suite and analyzed the memory, object size footprint, and instruction mix of the Java applications running on JVM. There interaction between the Java application and the JVM on which it is running was studied by Georges and Tia [9], [18]. It has been shown [22] that the dTLB miss rate of SPECjvm98 benchmark is around 2% that is much higher than 0.1% reported for SPEC95 benchmarks written in C/C++ workloads [15]. Similarly, the L1 data cache miss rates and more specifically the data write miss rates are also higher [9], [20].

Kim et al. [13] analyzed the memory reference patterns of applications in the SPECjvm98 suite and found that the size and the lifetime of the objects is small. Vijaykrishnan et al. [24] proposed an architectural support for object manipulation, stack processing, and two-level hybrid cache to improve the performance of Java applications. Shimizu and Kon [21] extended the basic idea in traja processor of java object lookaside buffer for Java applications running in interpreted mode or directly on Java processor. Java object lookaside buffer speeds up the resolution of constant pool references.

Vijaykrishnan et al. [23] presented a characterization of the energy consumption by the caches and the main memory when executing the SPECjvm98 benchmarks in the JIT and interpreter modes. The energy consumption is profiled for different hardware and software configurations. They made the observation that from the energy perspective JIT mode is better than the interpreter mode. The main memory energy consumption is more dominant than that of the caches, with the main contributor being data references. The energy consumed in the JIT mode is mainly due to code installation and the subsequent misses. Chen et al. [7] modified the cache organization to include a small cache for storing the dynamically generated native code to reduce the energy consumption in the caches for the Java applications running on JVM. Kim et al. [14] proposed an energy efficient Java execution using local memory and object co-location. In the object co-location, they study the static profile of Java programs running on JVM and use the results to intelligently place the objects in the heap space such that it reduces the energy consumption.

VII. Conclusion

In this paper, we investigate an energy-efficient Translation Lookaside Buffer (TLB) design for Java applications running on the JVM. Our analysis and insight led us to use a partitioned TLB structure that exploits memory reference characteristics and the interactions between the JVM and Java applications, which can not only reduce energy consumption but also incurs very little performance perturbation. The method horizontally partitions the traditional monolithic iTLB and dTLB into distinct, smaller TLBs for special purposes targeting for Java applications. Our J-iTLB scheme can reduce energy by 12.7% with a performance improvement of 1%. The performance improvement was obtained from the elimination of conflict misses between the JVM code and the Java application accesses in the iTLB. The energy saving is increased to 51% by combining the J-iTLB with a first-level small object iTLB at the expense of 1% performance impact. The J-dTLB organization, where the dTLB is split into two for reads and writes provides an energy saving of 34% with 1% performance impact. When the J-iTLB with an object iTLB is combined with the J-dTLB, we obtained 42% overall TLB energy savings with less than 1% performance impact.

VIII. Acknowledgment

This research was supported in part by NSF Grant CNS-0325536 and an NSF CAREER Award CNS-0644096.

References

[1] Dynamic SimpleScalar 1.0.1. http://www-ali.cs.umass.edu/DSS/index.html.

[2] JikesRVM Home Page. http://jikesrvm.sourceforge.net/.

[3] Spec JVM 98 Benchmarks. http://www.spec.org/osg/jvm98.

[4] Chinnakrishnan Ballapuram, Kiran Puttaswamy, Gabrieh H. Loh, and Hsien-Hsin S. Lee. Entropy-Based Low Power Data TLB Design. In *Proceedings of the ACM/IEEE International Conference on Compilers Architecture and Synthesis for Embedded Systems*, 2006.

[5] Chinnakrishnan S. Ballapuram, Hsien-Hsin S. Lee, and Milos Prvulovic. Synonymous Address Compaction for Energy Reduction in Data TLB. In *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*, 2005.

[6] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: a Framework for Architectural-level Power Analysis and Optimizations. In *Proceedings of the 27th International Symposium on Computer Architecture*, 2000.

[7] G. Chen, M. Kandemir, N. Vijaykrishnan, and M.J Irwin. Energy-aware Code Cache Management for Memory-constrained Java Devices. In *Proceeding of International Conference on Systems-On-Chip*, 2003.

[8] James Dinan and Eliot Moss. DSSWattch: Power Estimation in Dynamic SimpleScalar. In *Technical Report, UMass ALI Lab, Amherst, MA*.

[9] Andy Georges, Dries Buytaert, Lieven Eeckhout, and Koen De Bosschere. Method-level Phase Behavior in Java Workloads. In *Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented Programming, Systems, Languages, and Applications*, 2004.

[10] X. Huang, J. E. B.Moss, K. S. McKinley, S. Blackburn, and D. Burger. Dynamic simplescalar: Simulating java virtual machines. In *Technical Report TR-03-03, University of Texas at Austin, Department of Computer Sciences*, 2003.

[11] Koji Inoue, Tohru Ishihara, and Kazuaki Murakami. Way-predicting Set-associative Cache for High Performance and Low Energy Consumption. In *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*, 1999.

[12] Ismail Kadayif, Anand Sivasubramaniam, Mahmut Kandemir, Gokul Kandiraju, and Guangyu Chen. Generating physical addresses directly for saving instruction TLB energy. In *Proceedings of the 35th International Symposium on Microarchitecture*, 2002.

[13] Jin-Soo Kim and Yarsun Hsu. Memory system behavior of java programs: Methodology and analysis. In *In Proceedings of the SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 2000.

[14] S. Kim, S. Tomar, N. Vijaykrishnan, M. Kandemir, and M.J. Irwin. Energy-efficient java execution using local memory and object co-location. In *IEE Proceedings in Computer and Digital Techniques*, volume Vol.151 No.1, 2004.

[15] Dennis C. Lee, Patrick J. Crowley, Jean-Loup Baer, Thomas E. Anderson, and Brian N. Bershad. Execution characteristics of desktop applications on Windows NT. In *Proceedings of the 25th International Symposium on Computer Architecture*, 1998.

[16] Hsien-Hsin S. Lee and Chinnakrishnan S. Ballapuram. Energy Efficient D-TLB and Data Cache using Semantic-aware Multilateral Partitioning. In *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*, 2003.

[17] Jung-Hoon Lee, Gi-Ho Park, Sung-Bae Park, and Shin-Dug Kim. A Selective Filter-bank TLB System. In *ISLPED*, 2003.

[18] Tia Newhall. Performance Measurement of Interpreted, Just-in-Time Compiled, and Dynamically Compiled Executions. Master's thesis, University of Wisconsin, Madison, WI, 2003.

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

[20] A. S. Rajan, Shiwen Hu, and J. Rubio. Cache Performance in Java Virtual Machines: a Study of Constituent Phases. In *International Workshop on Workload Characterization*, 2002.

[21] Naohiko Shimizu and Chiaki Kon. Java object look aside buffer for embedded applications. In *Proceedings of the 2003 workshop on Memory performance*, 2003.

[22] Yefim Shuf, Mauricio J. Serrano, Manish Gupta, and Jaswinder Pal Singh. Characterizing the memory behavior of java workloads: a structured view and opportunities for optimizations. In *Proceedings of SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 194–205, 2001.

[23] N. Vijaykrishnan, M. Kandemir, S. Kim, S. Tomar, A. Sivasubramaniam, and M. J. Irwin. Energy behavior of java applications from the memory perspective. In *In USENIX Java Virtual Machine Research and Technology Symposium (JVM'01)*, 2001.

[24] Narayanan Vijaykrishnan, N. Ranganathan, and Ravi Gadekarla. Object-oriented architectural support for a java processor. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 330–354, 1998.