Power Consumption Reduction by Memory Compression in Java Embedded Systems

Chia-Tien Dan Lo and Mayumi Kato

Department of Computer Science
University of Texas at San Antonio
San Antonio, TX 78249, USA
Email: {danlo, mkato}@cs.utsa.edu

***Abstract*— Proposed in this paper is a new Java computation model that employs memory compression techniques to reduce memory footprint. Compressed data stored in RAM and ROM are decompressed on the fly whenever needed whereas data to be stored in RAM are compressed using a hardware de/compression module. Simulation results show that the proposed scheme reduces power consumption up to 78% compared to a non-compression counterpart. Moreover, performance evaluation of the proposed scheme is conducted and reported in terms of hardware and software implementations and real workload benchmarks.**

I. Introduction

Power Consumption is one of the major concerns in designing Java embedded systems such as personal digital assistants, cellular phones and pagers which typically are powered by re-chargeable batteries. Unfortunately, it may quickly run out of power during operations which involve simultaneous executions of many sophisticated applications such as 2D/3D graphics, graphical user interface (GUI), video, maps, mini-browsers, games, and interactive voice/image processing. For example, some battery lasts for 45 to 230 hours in standby mode but runs for 2 to 10 hours under normal operations for cellular phones/communicators. Every component in the system consumes energy. Especially, the memory system consumes a significant amount of power. One of the techniques for reducing power consumption is turn off unused memory banks if they do not contain live information. In this paper, a new architecture that employs memory compression techniques to reduce memory footprint will be studied. Compressed data stored in RAM and ROM are decompressed on the fly whenever needed whereas data to be stored in RAM are compressed using a hardware de/compression module.

De/compression algorithms can be mapped to hardware nicely. For example, X-Match de/compression algorithm has been implemented in Field-programmable Gate Arrays (FPGAs) and Application-Specific Integrated Circuit (ASIC) [19]. The X-Match is a dictionary-based partial match compression algorithm. A 4-byte pattern is considered as a match against a dictionary if at least 2 characters are the same as some dictionary entry. The algorithm returns a tuple <ML, MT, Literal>, following a signal that indicates hit/miss. The first term ML is match location, the second term MT is the match type, and the third term is Literal or empty. If an input byte does not match, the byte is transmitted literally. The input bytes are stored in the dictionary with a move-to-front (MTF) strategy: a new tuple is placed at the front of the dictionary while the rest move down one position. Hardware solutions for other de/compression algorithms such as X-Match with Run-Length, and Arithmetic Coding have been proposed [19], [10], [13], [14], [18], [11], [17].

Due to the high density of VLSI technology, mapping software functions to hardware becomes feasible and promising. In [4][20][21], a hardware dynamic memory management module is built to improve dynamic memory management performance which may not be achieved by the software counterpart. Moreover, a memory system can consume a large portition of the overall system energy [3]. It has been observed that Java memory is one of the major power consumption sources and consumes as high as 58.7% of the total energy [22] and in some ARM7 family embedded processor [2], up to 70% of the energy is consumed through memory accesses [15]. Therefore, by compressing data and code, the system power consumption can be reduced.

II. Related Work

Memory compression has been studied extensively over the past decade [16], [8], [1], [9], [12], [6]. The most related work is the memory architecture proposed by Chen *et al.* [5], [7], [6]. In their design, a decompressor is used for read-only memory and the leakage energy consumption is eliminated by reducing the number of active (powered on) memory blocks that holds read only data, KVM code and class libraries. The memory which contains read/write data is composed of memory banks, each of which can be independently turned on or off (a memory partition technique). The total energy consumed by the partitioned memory is the sum of the energy consumed by each active bank over the time. Figure 1 shows the high level view of this design.

When a data item belonging to a compressed memory block is requested, the whole block is decompressed and written into the Scratch Pad Memory (SPM). The advantage of this strategy is obvious that a lot of memory space is saved because data in read-only memory is in compressed form. This reduces the leakage energy consumption in read-only part of the main memory system. The amount of the saving is

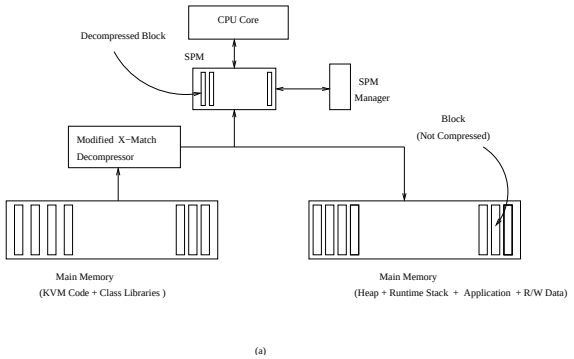

Fig. 1. High-level View of the SoC Memory Architecture with Decompressor

determined by the compression ratio and the algorithm used. It is concluded that compression is effective in reducing energy even when considering the runtime decompression overheads for most applications. In this approach, however, data stored in RAM are not compressed, and the results are mainly from simulations.

III. The Proposed Architecture

Our goal is to build a system with a de/compression module for data stored in RAM and ROM, which can compress/decompress data on the fly without losing overall system performance. One of the expected effects is the reduction of the leakage energy consumed by these memory banks which contain no live data. Meanwhile, the compressed data stored in memory can be retrieved as if there were no compression. The proposed architecture consists of hardware de/compressors, SPM, RAM and ROM, and a table (i.e., Address Lookaside Buffer (ALB)) to keep virtual page addresses, the size of compressed version and physical page address (the physical address points to the starting address of that page in compressed RAM). Figure 2 shows the proposed architecture with a compressor and a decompressor. Note that the compressor and decompressor can be built in one module.

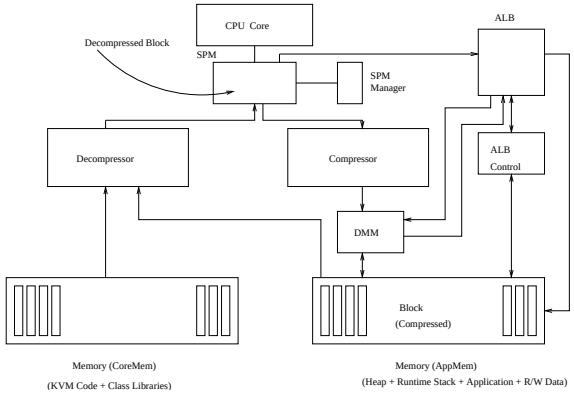

Fig. 2. Our SoC Memory Architecture with one Compressor and one Decompressor.

The SPM holds uncompressed data and acts as a traditional cache. Only if an SPM miss happens can the decompresion or compression occur. The first main memory module, called CoreMem, contains the ROM, KVM code, and Java class libraries. The second main memory module, called AppMem, is composed of the RAM which includes heap, run-time stack, r/w data and application class objects. AppMem is divided into blocks of equal size (e.g., 64 Bytes), and CoreMem is not. A bitmap is associated with the AppMem to indicate memory status: 0 indicates a free block; and 1 indicates an occupied block.

A compressed page is stored in the AppMem and may occupy several blocks. The size of a compressed page is measured in terms of number of blocks. The ALB is used to map a page's virtual address to the address where it is stored in the AppMem. When a compressed pages in the AppMem is retrieved, the starting address and size (in terms of blocks) are obtained from the ALB to index the corresponding compressed image. The data includes compressed version of run-time information necessary for a Java application such as heap, run-time stack, r/w data and application class objects. Therefore, there are both decompression and compression modules for accessing and storing data in AppMem. The data are read from AppMem and passed to the decompression module or from the SPM and passed to the compression module. Subject to compression algorithm, a compressed pages after execution may be shrinked or inflated. For the write operation, one of three types of operations is performed (B denotes block size): if the size of compressed version is lager than the old one following the relation that [new_size/B] - [old_size/B] $>$ 0. The compressed version of the page has to be relocated because the page grows after compression; if the size of compressed version is equal to the old one, follow the relation that [new_size/B] - [old_size/B] $= 0$. The compressed version of the page can be stored in the same address as the old one; if the size of compressed version is smaller than the old one, follow the relation that [new_size/B] - [old_size/B] < 0. The compressed version of the page can be stored in the same address as the old one but the bitmap has to be modified to reflect releasing free memory blocks. Moreoever, the ALB is updated accordingly.

The operations in the CoreMem, however, is slightly different from those in the AppMem for the reason that the CoreMem is read-only. The data are read from CoreMem and passed to the decompression module. The AppMem, on the other hand, stores data for read and write operations. Thus, block relocation is not necessary in the CoreMem. A directory is used to keep compressed page information and never being updated. Moreover, the content of CoreMem is obtained by off-line compressing KVM code and Java class libraries using a better compression algorithm because time is not an issue. In some applications, the memory reference patterns in ROM and RAM may be drastically different. For example, variable data, e.g., numbers, in the AppMem typically occupy lower bytes in a word and leave upper bytes null whereas Java byte code stored in the CoreMem is not. Thus, there is a need to used two different de/compression modules or tuning parameters in a de/compression algorithm.

In some applications, the memory reference patterns in ROM and RAM may be drastically different. For example, variable data, e.g., numbers, in the AppMem typically occupy lower bytes in a word and leave upper bytes null whereas Java byte code stored in the CoreMem is not. Thus, there is a need to used two different de/compression modules or tuning parameters in a de/compression algorithm. To get a better overall performance, one algorithm may be used for RAM whereas another algorithm should be applied to ROM. Moreover, different compression environments are assumed in that data compression in ROM is off-line and that in RAM is on-line. The off-line compression may be done in highl optimization level which results in longer time. However, the on-line compression can not have too much optimization and must be fast. The decompressor may use different parameters or policies in the underlying compression algorithm. Table I summarizes these compression algorithms.

IV. Simulation Details and Results

A. Simulation Design and Benchmarks

The goal of the simulation is to study the cost (energy consumption) of the de/compression module and compare the results to other related research work. Simulation are conducted by running benchmark programs with a modified KVM. Also measured are compression ratios for Java class files (libraries), heap, stack, and global data area. The benchmark programs examined are described in Table II. As to energy consumption estimates, an energy consumption model devised in [6] is employed. Table II shows descriptions of the benchmarks.

B. Simulation Results

1) Class-Oriented Compression: Before a Java program is executed, its class files have to be loaded into memory, i.e, CoreMem in our model. By compressing class files, we want to study the compression performance when the classes are stored in the CoreMem. Simulation results show that the Xp-h algorithm achieves up to 56.5% of compression ratio as shown in Figure 3. $gzip$ is roughly better than others. However, implementation complexity at hardware level of these algorithms remains to be studied.

2) Memory-Oriented Compression: In this experiment, we want to study the compression ratio when the RAM is compressed. Figure 4 shows the watermark with and without compression for the application "ManyBalls" where the watermark is the highest memory location reached by the program. The X-Match algorithm could achieve compression ratio of 18.45% on average for RAM compression. Similar watermarks and compression ratios can be obtained for the remaining benchmarks, which are omitted because of page limitation. Note that as startup class loading is small, compression ratio becomes low.

Table III shows the Xp-h compression ratios versus block sizes which is the virtual page size. Simulation results show that a block size of 8 KB yields a better compression ratio than other settings. However, the energy consumption for larger block settings may be larger [6]. The results also show that the X-Match algorithm achieves compression ratio up to 32.59% for RAM compression.

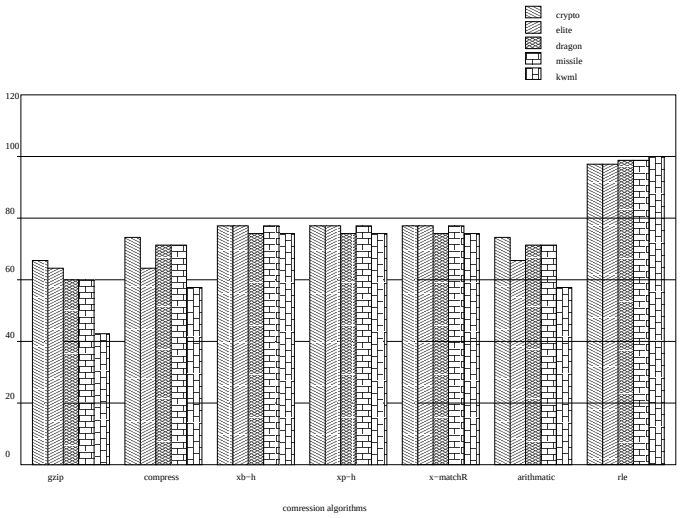

Fig. 3. Compression Ratio for Selected Applications

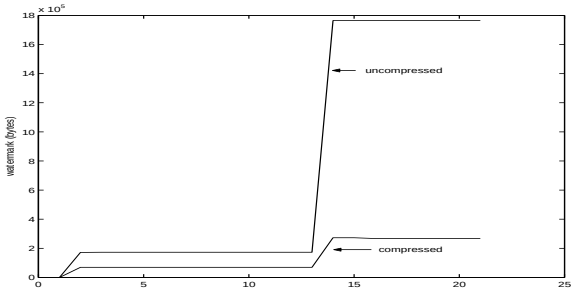

Fig. 4. Watermark for ManyBalls

3) Energy Estimate: Applying compression to memory reduces its demand for running applications. In other words, it reduces power consumption of CoreMem and AppMem by turning off unnecessary banks. This section provides depicts an energy model and reports energy saved with compression techniques. The energy model is based on the energy consumption and execution cycles assumed in [6]. We evaluate the X-Match runtime de/compressor with the assumption that a de/compression rate of 4 bytes/cycles and the energy consumed in each de/compression of a word (4 bytes) is equal to three SPM accesses. The overall energy consumption is equal to the sum of dynamic energy in the SPM, leakage energy in SPM, dynamic energy in main memory, leakage energy in main memory, de/compression module and the processor. The parameters for the energy model are summarized in Table IV.

The energy consumed by the de/compression module is

TABLE III
Compression Ratio vs. Block Size

Block Size	0.5KB	1.0KB	2.0KB	4.0KB	8.0KB
Compression Ratio	41.88%	37.43%	35.25%	34.08%	32.59%

TABLE I

DE/COMPRESSION ALGORITHMS

Algorithms	Description
gzip	$LZ77$-based algorithm followed by a variable-length coder which builds an adaptive dictionary of substrings of the message.
compress	a UNIX utility; one of the earlier applications of LZW.
Run-Length Encoding (RLE)	Consecutive identical characters are replaced by the length of a run
Arithmetic coding	A method of assigning code to symbols that have a known probability distribution.
X-Match	A dictionary-based compression technique that uses phased binary coding for match locations
Xp-h	A variant of X-Match in which a phase binary coding is used for match locations and a static variable-length Huffman codes for match types
JAR	A Java archive with gzip compression based on Huffman coding

TABLE II

BENCHMARKS

Programs	Brief Description	Sources
Crypto	A light-weight Java implementation of cryptographic algorithms	www.bouncycastle.org
Dragon	A game program	comes with Sun's KVM
Elite	A fast 3D rendering engine for small devices running Java.	home.rochester.rr.com/ohommes/Elite
KWML	KWML is a KVM WML (WAP) browser running on Palm.	www.jshape.com
ManyBalls	A game program	comes with Sun's KVM
Missiles	A game program	comes with Sun's KVM
WebViewer	A fast HTML Web browser for Java-enabled (J2ME) mobile devices.	www.reqwireless.com
HTTP Demo	Technical demonstration program of http for MIDP.	Comes with Sun KVM
Ticket Auction	Look up the ticket auction for a band, set an alert, or making a bid in this mockup of a ticket auction service.	comes with Sun KVM
Stock	Get stock quotes from a publicly available website, and set alerts.	comes with Sun KVM
Audiodemo	Listen to sounds using Mobile Media API's audio building block.	comes with Sun KVM

TABLE IV

PARAMETERS AND VALUES FOR BASE CONFIGURATION

parameter	value
SPM block size	1 KB for read-only data 512 bytes for writable data
Main memory capacity	512 KB
SPM access time	1 cycle
Main memory access time	3 cycles
SPM dynamic energy/read	0.5216 nJ
SPM dynamic energy/write	0.6259 nJ
Main memory dynamic energy/read	1.334 nJ
Main memory dynamic energy/write	1.601 nJ
Main memory leakage energy/byte/cycle	2.54×10^{-6} nJ
SPM leakage energy/byte/cycle	2.54×10^{-6} nJ

close related to the SPM miss that leads to accesses to the compressed memory. The de/compression occurs when data can not be found in SPM, i.e., the data stored in compressed CoreMem or AppMem must be retrieved or write back data to AppMem if the SPM is full. However, the optimal size of the block is determined by the SPM miss rate and miss penalty in combination with the spatial and temporal locality. Table V shows energy consumption and execution cycles for the ManyBalls benchmark under the base configuration, i.e., the system proposed in [6] which does not have RAM compression.

By introducing RAM compression, we have to evaluate energy saved in AppMem is larger than that consumed in the de/compression module itself. Let x be the number of times a block is compressed, and y be the number of times a block is decompressed. The total energy consumed by de/compression

TABLE VII

ENERGY COMPRESSION REDUCTION WITH X-MATCH ALGORITHM

Application	E_{RAM}	$E_{de/compression}$	Total
ManyBalls (w/ compression)	5.8 mJ	1.1 mJ	6.9 mJ
ManyBalls (w/o compression)	31.45 mJ	-	31.45 mJ

is $E = (x+y) \times B(bytes)/4\ (bytes) \times 0.6259\ nJ \times 3$ which equals to $1.1\ mJ$. We safely estimate by choosing the larger energy consumption of a SPM wirte. On the other hand, energy saved by RAM compression is estimated to $5.8\ mJ$. Thus, our system architecture can reduce 78% of energy consumption compared to the architecture without RAM compression. Table VI and Table VII list detailed numbers.

4) Software De/Compression Versus Garbage Collection: In some systems, hardware compression may not be affordable. Therefore, the performance of a software compression implemention is worth studying. A typical garbage collection cycle is invoked when free heap memory reaches a threshold. Fortunately, in a compressed memory configuration, number of garbage collection cycles should be reduced and so is the garbage collection time. If the time used for de/compression is less than that of garbage collection, the software version of the proposed model outperforms its original one.

The number of de/compression invocatoins is related to the number of the cache miss because data are brought from/to the compressed heap. The compression and decompression time is classified into two cases.

- Case 1: when a page is referenced, and the SPM is not

TABLE V

ENERGY CONSUMPTION FOR J2ME APPLICATION MANYBALLS

Application	Memory Energy Dynamic	Memory Energy Leakage	SPM Energy Dynamic	SPM Energy Leakage	Read-Only Contribution	Number of SPM Misses	Number of Cycles
ManyBalls	5.90 (5.05%)	73.08 (62.59%)	29.22 (25.03%)	8.56 (7.33%)	59.98%	11455	54.87

TABLE VI

ENERGY CONSUMPTION

Size(uncompress)	base config.	512KB
$E_{leakage/byte/cycle}$	base config.	$2.54 \times 10^{-6} nJ$
# of cycles (total)	base data	54.87 (million cycles)
r (compression ratio)	measured data	18.45%
# of cycles for ROM access	# of cycle $\times$ read-only contribution	6871
# of cycles for RAM access (x+y)	# of cycle $\times$ (1 - read-only contribution)	4586
E_{main} (uncompressed)	$E_{dynamic}$ (uncompress) + $E_{leakage}$(uncompress)	79.58 mJ
E_{RAM}(uncompressed)	E_{main} $\times$ (1 - read-only contribution)	31.45 mJ
$E_{RAM_{de/compression\_module}}$	(x + y) $\times$ Block size (bytes)/4(bytes) $\times$ 3 $\times$ $E_{SPMaccess}$	1.1 mJ
E_{RAM}(compressed)	E_{RAM}(uncompress) $\times$ (1 - read-only contribution)	5.8 mJ

filled, decompression is invoked.

- Case 2: when a page is referenced, and the SPM is filled, both compression and decompression are required: the former is for the page in the cache, and the latter is for the page in the memory.

The WK 4×4 compression algorithm [23], a variant of Ziv-Lempel algorithm, is used in this measurement. Other compression algorithms would also obtain similar results. The de/compression time is accumulated during program execution. Table VIII shows de/compression time and garbage collection time. The time of one GC invocation varies from 9,215,726 nsec (Audiodemo) to 21,459,819 nsec (WebViewer). The time of one compression (4KB data) ranges from 301,082 nsec (Audiodemo) to 345,353 nsec (WebViewer), and the time of one decompression is 192, 419 nsec (HTTP Demo) to 211,804 nsec (Audiodemo) on average.

Intuitively, the proposed architecture will outperform the original KVM if the sum of the compression and decompression times is less than or equal to the GC time. The compression time is 70 times shorter than the GC time, and the decompression time is 100 times shorter than the GC time. This indicates one GC invocation roughtly corresponds to 50 sets of de/compression invocations. By indroducing memory compression, the garbage colletion is not invoked in running all the benchmark programs. For most of the benchmark programs, the total execution is increased by less than 5.48%. The de/compression time is proposonal to the number of miss in the SPM. The SPM size in this experiment is set to 4 pages, i.e., 16 KBytes. The total de/compression time could be further reduced by increasing buffer size and applying better page replacement policies.

The high buffer miss rate (3,651) in the WebViewer benchmark results in 91% time penalty. The reason is because the smaller buffer can not hold each web pages accessed during web surfing. Although this problem is hard to solved in the software domain, it can be nicely overcome in the hardware domain in which de/compression time is reduced greatly.

5) Memory Footprint: The average compression ratio of WK 4×4 is 48.54%. The heap memory can be saved 51.46%. The proposed architecture can reduce up to 53% of memory demand for wireless Java applications. The performance of the proposed architecture is dependent on several factors: page replacement policy, data characteristics (operation types of benchmark applications, e.g., server-client applications, stand-alone applications, web interactive applications, cryptography, etc.), and the length of executions (long or short run, etc).

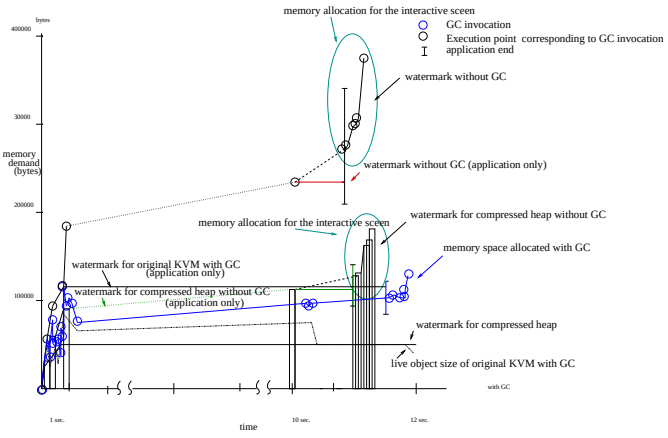

Fig. 5. Dynamic Watermark Aanlysis

TABLE IX

MEMORY FOOTPRINT

benchmarks	Web-Viewer	HTTP	Stack	Audio-demo	Many-balls
Watermarks orig KVM with GC,	125,084	91,420	121,035	125,824	135,808
Watermarks mod KVM w/ comp	101, 457	50, 395	72, 993	59,083	88,475
% reduced	18.89	44.88	39.69	53.04	34.85

Table IX shows the memory foorprint of the benchmark programs with/without compressed memory. The memory footprint is reduced from 19% to 53%. To analyze watermark change along the time, Figure 5 plots the watermark for the WebViewer in a run. It shows watermark without GC, watermark with compressed heap, and watermark with compressed heap and GC. At the end of the run, the watermak is incrased drastically due to the object creation for the preparation of the interactive screen.

TABLE VIII
TIME AND SPACE EFFICIENCY

Application	WebViewer	HTTP demo	Stock	Audiodemo	manyballs
# of GC	34	7	15	6	5
GC time (ave. nsec)	21,459,819.00	4,711,766.86	3,775,544.27	9,215,725.80	5,093,971.20
total (nsec)	729,633,846.00	32,982,368.02	56,633,164.05	55,294,359.80	25,469,855.00
# of miss (case1)	4	4	4	4	4
# of miss (case2)	3651	475	2020	1294	176
decomp. time (nsec)	204,191	192,419	175,644	211,804	190,149
comp. time (nsec)	345,353	337,019	306,075	301,082	386,815
total de/comp. time (nsec)	2,007,201,908	252,252,726	973,774,956	664,521,700	102,306,260
total exec. time (nsec)	1,401,416,332	6,804,729,505	16,742,939,643	13,857,409,290	4,959,255,549
% increased	91.16	3.22	5.48	4.4	1.55

V. CONCLUSIONS

When Java handheld devices become ubiquitous, how to prolong the operation time for the battery-powered devices is becoming an issue. In the proposed scheme, power consumption is reduced by memory compression. Both application memory (AppMem) and core memory (CoreMem) are compressed. Simulation results show that class compression ratio is up to 56.6% using the Xp-h algorithm, memory compression ratio is up to 18.45%, and the power consumption can be futhre reduced by 78%.

By and large, a software compression implementation yields less than 5.48% time overhead which can be further improved using a larger buffer and a better buffer replacement policy. Through memory compression, garbage collection is invoked less frequently and even is not necessary. The memory footprint is reduced up to 53% in real workload benchmarks. With the advance in VLSI technology, the proposed architecture can be easily implemented in hardware and it provides a transparent interface between CPU and memory as if there were no compression. Moreover, in addition to power saving, the compressed memory also results in better system performance such as page faults, cache miss, etc. Consequently, memory compression is promising in the Java embedded computing environment.

REFERENCES

[1] B. Abali, H. Franke, X. Shen, D. E. Poff, and T. B. Smith, "Performance of hardware compressed main memory," in *Proc. HPCA*, Nuevo Leone, Mexico, Jan. 2001, pp. 73–81.

[2] (2004) The ARM website. [Online]. Available: http://www.arm.com/products/CPUs/ARM7TDMI.html

[3] F. Catthoor, F. Franssen, S. Wuytack, L. Nachtergaele, and H. DeMan, "Global communication and memory optimizing transformations for low power signal processing systems," in *Proc. the IEEE Workshop on VLSI Signal Processing*, 1994, pp. 178–187.

[4] J. M. Chang, W. Srisa-an, C. D. Lo, and E. F. Gehringer, "Dmmx: Dynamic memory management extensions," *The Journal of Systems and Software, Elsevier Science*, vol. 63, no. 3, Sep. 2002.

[5] G. Chen, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and M. Wolczko, "Adaptive garbage collection for battery-operated environments," *The 2nd USENIX Java Virtual Machine Research and Technology Symposium (JVM'02)*, 2002.

[6] G. Chen, M. Kandemir, N. Vijaykrishnan, and W. Wolf, "Energy savings through compression in embedded java," *Tenth International Symposium on Hardware/Software Codesign (CODES'02)*, 2002.

[7] G. Chen, R. Sherry, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and M. Wolczko, "Tuning garbage collection in an embedded java environment," *The 8th International Symposium on High-Performance Computer Architecture (HPCA'02)*, 2002.

[8] J. M. Cheng and L. M. Duyanovich, "Fast and highly reliable ibmlz1 compression chip and algorithm for storage," in *Proc. Hot Chips VII*, Stanford, California, USA, Aug. 1995, pp. 155–165.

[9] F. Douglis, "The compression cache: Using on-line compression to extend physical memory," in *Proc. UNSNIX Conference*, San Diego, California, USA, Jan. 1993, pp. 519–529.

[10] S. Jones, "Partial-matching lossless data compression hardware," *IEE Proc-Comput. Digit. Tech*, vol. 147, no. 5, 2000.

[11] ——, "High-performance phased binary coding," *IEE proceedings*, 2001.

[12] M. Kjelso, M. Gooch, and S. Jones, "Design and performance of a main memory hardware data compressor," in *Proc. EUROMICRO-22*, Prague, Czech Republic, Sept. 1996, pp. 423–430.

[13] M. Kjelso, M. Gooch, and S.Jones, "Design and performance of a main memory hardware data compressor," *Proceedings of EUROMICRO-22*, 1999.

[14] ——, "Performance evaluation of computer architectures with main memory data compression," *Journal of Systems Architecture*, vol. 45, no. 571-590, 1999.

[15] S. Lafond and J. Lilius, "An energy consumption model for java virtual machine," Abo Akademi University, Department of Computer Science, Lemminkaisenkatu 14A, FIN-20520 Turku, Finland, TUCS Tech. Rep. 597, 2004.

[16] C.-Y. Lee and R.-Y. Yang, "High throughput data compressor designs using content addressable memory," *IEE Proceedings - Circuits, Devices and Systems*, vol. 142, no. 1, pp. 69–73, 1995.

[17] J. Nunex, C. Feregrino, S. Bateman, S. Jones, and S. Bateman, "The x-matchpro: A proasic-based 200 mbytes/s full duplex lossless data compressor," *IEE proceedings*, 2000.

[18] J. Nunex, C. Feregrino, S. Bateman, and S.Jones, "The x-matchlite fpga-based data compressor," *IEEE proceedings*, 1999.

[19] J. Nunex and S. Jones, "The x-matchpro 100 mbytes/second fpga-based lossless data compressor," *IEE proceedings*, 2000.

[20] W. Srisa-an, C. D. Lo, and J. M. Chang, "A hardware implementation of realloc function," *Integration, the VLSI Journal, Elsevier Science*, vol. 28, 1999.

[21] ——, "Scalable hardware-algorithms for object resizing and reclamation," *International Journal of Microprocessors and Microsystems, Elsevier Science*, vol. 25, 2002.

[22] N. Vijaykrishnan, M. Kandemir, S. Tomar, S. Kim, A. Sivasubramaniam, and M. J. Irwin, "Energy characterization of java applications from a memory perspective," *The USENIX Java Virtual Machine Research and Technology Symposium*, April 2001.

[23] P. R. Wilson, S. F. Kaplan, and Y. Smaragdakis, "The case for compressed caching in virtual memory systems," *USENIX Technical Conference '99*, 1999.