Object Cache: An Energy Efficient Cache Architecture

S S Shekhar Y N Srikant

IISc-CSA-TR-2005-13
http://archive.csa.iisc.ernet.in/TR/2005/13/

Computer Science and Automation
Indian Institute of Science, India

October 2005

Object Cache: An Energy Efficient Cache Architecture

S S Shekhar Y N Srikant

Abstract

Object-oriented programming languages provide a rich set of features that provide significant software engineering benefits. The increased productivity provided by these features comes at a justifiable cost of complexity in the runtime environment. This complexity leads to reduced performance and increased energy consumption of the platform as well as the programs. To alleviate the problem of increased energy consumption in embedded system architectures that typically support runtime environments for object oriented programs, this paper proposes an energy-efficient cache architecture that can have a significant impact on the overall system energy consumption.

The proposed *Object-cache* architecture consists of a data cache (reduced in size) and an additional small cache structure that caches only the objects called the *Object-cache*. A high degree of temporal locality among a large number of short lived objects ensures good performance of such an architecture. At the same time, the reduced active cache size, at times accompanied by minor improvements in performance, leads to a significant improvement in the energy behavior of programs. Using applications from SPECjvm98 benchmark and a cycle accurate simulation, the *Object-cache* architecture is shown to reduce average data cache energy consumption by up to 35.95% and overall system energy consumption by up to 12.4%. Our study further shows that, using a configurable architecture for both data cache and object cache in embedded systems, we could fine tune the cache parameters for optimal energy-performance trade-offs.

1 Introduction

Embedded system developers have embraced Java because of the support it provides for dynamically downloading applications.[1] Java environament

[1]Although our work focuses on Java, it is equally applicable to most other object oriented programming languages with a similar behavior.

for embeddded systems typically come in two configurations. A lightweight configuration, where the Java VM is integrated as part of the software environment where a few modules may run native libraries and another that bases the entire programming and user environments on Java. The virtual machine is however, common to both these configurations. A typical Java execution environment is depicted in Fig. 1

Studies by Lafond [8] have shown some interesting results about energy consumption in typical Java environments. According to this study, Java Virtual Machine itself consumes about 60–65% of the total energy for running a program. Memory accesses contribute for upto 70% of this energy. The cache system being in the critical path of any memory reference contributes significantly to both performance and energy consumption of the memory hierarchy. In fact caches can consume upto 50% of a microprocessors energy, making it a ripe candidate for optimization. For energy efficient cache architecture design, we focus on optmizing the performance and energy consumption of the cache system.

Objects typically are small, short-lived and frequently accessed during their short life span. Thus object references exhibit certain characteristic properties that make the object fields referred by them suitable candidates to be cached separately in a cache structure known as an *Object-cache*. This paper describes such a novel *Object-cache* architecture and evaluates the effect of such a cache architecture on the energy consumption and performance behavior of typical Java programs such as the SPECjvm98 benchmarks.

The remainder of the paper is organized as follows. Section 2 discusses some related work. Certain properties of objects and object references which ensure good performance of the object cache architecture are discussed in Section 3. Section 4 motivates the necessity to cache objects in a separate *Object-cache*. In Section 5, architecture of the *Object-cache* is discussed in detail. The experimental details, simulation methodology and experimental results are described in Section 6. Section 7 describes the benefits of fine tuning the configuration of the object cache architecture. Some concluding remarks are provided in Section 8.

2 Related Work

An important trend in low-power hardware design is the partitioning of hardware components into smaller and less energy-consuming components ([3], [4]). The selective disabling of unused components is an effective mechanism for reducing energy consumption. Partitioning has been used in caches for both performance and energy considerations [12]. A large cache is broken

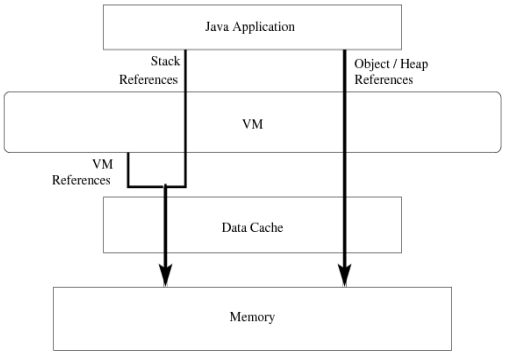

Figure 1: Typical Java Architecture.

down into smaller sub arrays to reduce the wiring and diffusion capacitances of bit lines as well as the wiring and gate capacitances of word lines used to activate the memory cells. The reduced capacitance helps lower the cache access time and dynamic energy consumption when accessing the caches. Flautner et.al [2] proposed the drowsy cache scheme which is one of the most poular schemes to reduce energy consumption of caches. In the drowsy cache scheme, the set of cache lines that are not active for a given period of time (cold cache lines) are put into a state preserving low power drowsy mode. Another recent study by Kim et.al [13] deals with partitioning of cache into sub-caches at the architecture level and selectively activating the sub-cache that contains the required data upon a memory reference as against previously proposed schemes of partitioning the cache at the circuit level and enabling and disabling them at the architectural level. It also evaluates such a partitioned sub-cache architecture with SPECjvm98 benchmarks and reports a significant reduction in the energy consumption of the cache.

A study of the memory system behavior of Java programs by analyzing memory reference traces of several SPECjvm98 benchmark applications is presented in [14] . This study supports our own observations that many objects are short lived. Dieckmann et.al [15] also conducted a similar study about the allocation behavior of the SPECjvm98 Java benchmarks. This study shows that typical Java benchmarks run with a fairly small heap size and the objects themselves are small and reiterated the result that many objects are short lived. Recently, a scheme to cache the objects separately to improve the functionality and efficiency (performance alone) of heap memory management where in the cache has a separate address space, being addressed with separate Object IDs has been proposed in [11]. Energy benfits of an anotation based allocation of objects in a local memory is studied in [7]. This technique requires major modifications in the memory management schemes

of the virtual machine. The anotation scheme itself requires tedious modifications to all Java classes involved. Our proposed *Object-cache* scheme which is a simpler and more amenable solution aims to capture similar application behavior without the tedium of annotating Java classes and modifying the memory management of the virtual machine.

3 Object Properties

Memory management in Java and its object model have several distinctive features. Extensive dynamic memory allocation of objects on the heap is one such feature. Java programs allocate a significant fraction of the memory (all objects) dynamically on the heap. In a virtual machine environment, we regard references to the objects of java application as object references. All other references including virtual machine references are treated as non-object references (See figure 4 and Section 5). Although languages like C also provide for dynamic memory allocation,the number of references to the dynamically allocated memory in C is very small and these do not exhibit behavior similar to that of object references. Certain interesting properties exhibited by Java objects and memory references to these objects make them suitable candidates to be cached in a separate *Object-cache*. Firstly, previous studies have shown that heap references constitute a considerable portion of the total memory references in Java programs. Kim and Hsu [7] report that 45–50% of all memory references, on the average, are heap references for the SPECJvm98 benchmarks suite. Secondly, there is a high degree of temporal locality observed among these object references themselves which inherently make them ripe candidates to be cached separately. Thirdly, most of the objects are short-lived, which means they would soon make way for other object accesses without causing any additional misses. A detailed description of the analysis is given in the following sub-sections.

3.1 Temporal Locality

Experiments were conducted to determine the temporal locality of the objects among themselves for various SPECjvm98 benchmarks. The interval in terms of the number of accesses between two consecutive accesses to the same object were measured. The results are plotted in Fig. 2. As seen in Fig. 2, a high degree of temporal locality is observed in object accesses. 40–80 % of the object accesses refer to the same object in the immediately following access. A whooping 90–95 % of the objects are re-accessed within 100 accesses of their initial access. It is important to note here that the

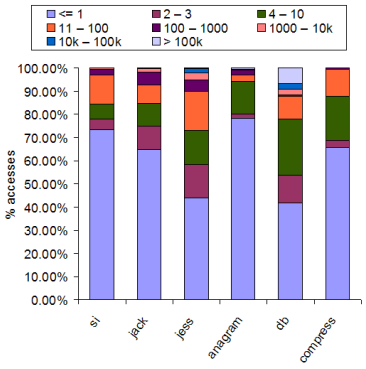

Figure 2: Temporal Locality of objects

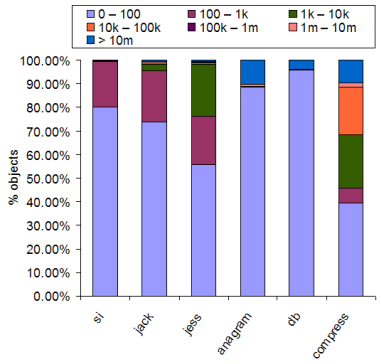

Figure 3: Life time objects across benchmarks

temporal locality observed is among the object accesses themselves. This high degree of temporal locality increases the hit rate of a separate *Object-cache*.

3.2 Life Time

The life time of an object for our analysis purpose is defined as the time duration in terms of total number of (global) memeory accesses between its first instantiation and its last access. Interestingly, it is observed that most of the objects have a very short life time. 40–90 % of the objects across different benchmarks have lifetime of less than 100 accesses. An average 80 % of the objects, have lifetime of less than 1000 accesses. The detailed break-up of life-times of objects in different benchmarks is given in Fig. 3. The definition for life time of an object restricts its scope only to the last use of the field instead of the time of garbage collection because we are interested only in the earliest time at which the object field may be released from the cache without causing any extra misses. Note that an object field may be replaced in the cache without it being garbage collected after its last access. The performance and energy benefits of the proposed *Object-cache* architecture does not depend on how long the object fields further stay in memory without being garbage collected as long as they are not accessed. However, the absolute life time of ojects are also shown to be small in **??**.

4 Motivation

Lafond ([8]) shows in his study that memory accesses contribute for upto 70% of this energy in typical Java programs. This motivates us to concentrate on optimizing the memory hierarchy energy requirements by reducing the number of memory accesses that go all the way upto the memory (cache miss). All the above discussed properties of objects and object references, emphasize the advantage of a separate *Object-cache* to cache only objects. The most significant energy benfit from using an object cache is that frequently used objects in the *Object-cache* are not easily replaced by other conflicting memory references. A high degree of temporal re-use of objects ensures a good performance of the perceived *Object-cache*. Moreover, Wright ([11]) shows that an *Object-cache* addressed by an object ID instead of the memory address, would also gain in performance. Short average life-time of objects implies that these short-lived objects need not have to reside in the *Object-cache* for a long period of time and hence make way for the other objects to be accessed without causing any additional misses. This leads us to think of having a smaller *Object-cache* which would handle larger amount of data (object fields) than it would have, if there were conflicts with regular data in the data cache, and yet deliver the same or better performance (in terms of hit-rate). A smaller cache means lesser energy consumption. Therefore, the energy reduction in the perceived *Object-cache* architecture is mainly due to the partitioned architecture and reduced cache size. It should however be noted that the observed properties of objects and object references ensure good performance of a partitioned architecture and reduced cache size. Apart from this major energy benefit of an object cache, embedded systems could make use of configurable caches to configure cache architectures with optimal parameters to save energy. We also study the benefits of using such a configurable cache architecture to both object and data caches. The potential benefits obtained by configuring the two cache structures separately for optimal energy performance trade-offs are higher than those acheivable by configuring a single large data cache.

5 *Object-cache* Architecture Design

As explained in the previous sections, an *Object-cache* is a cache structure, separately designed for caching only objects. It creates a parallel data access path for objects. The typical memory hierarchy in the presence of an *Object-cache* is shown in Fig. 4. The object references are directed through the *Object-cache* while the other references are directed through a regular

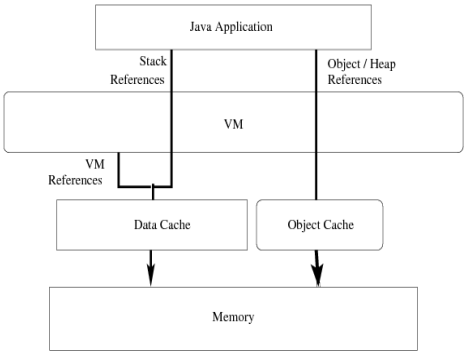

Figure 4: Memory hierarchy with Object Cache

data cache. Apart from introducing an additional *Object-cache* module, a system supporting the *Object-cache* hierarchy must also implement demultiplexing of the references to their respective caches. This demultiplexing could potentially be carried out by the hardware or by the virtual machine.

To implement separating the object references in the hardware, two additional registers, *object-heap-start* and *object-heap-end* are required. These registers bind the *object area* on the heap where objects are allocated. Any reference that is addressed to this *object area* bound by the *object-heap-start* and *object-heap-end* is cached in the *Object-cache* by the hardware while the remaining references are directed through the regular data cache. The increase in logic is only that of a test if the address requested by virtual machine lies in this range or not. The *object-heap-start* and *object-heap-end* registers could be set by the JVM during its initial boot and subsequently maintained to bind the region in which objects are allocated.

The other alternative is to modify the Instruction set architecture (ISA) of the target architecture to have separate instructions for loading and storing objects called *load-obj* and *store-obj* apart from the regular load and store instructions. On issue of a *load-obj* instruction (object reference), the processor accesses the *Object-cache* and loads it from the *Object-cache* if it is already available (*Object-cache* hit) and on an *Object-cache* miss, it accesses the memory (and also caches it on to the *Object-cache* if such a cache policy is used). Other data references (non-object references) are loaded through the regular data access path of a data cache using the regular *load* instruction. The Java virtual machine could now make use of the *load-obj* and *store-obj* instructions to load (store) from memory.

Hardware implementation of demultiplexing the references is best suited for legacy codes where no modification is required in the legacy bytecodes

that are being used. The only change required would be in the Java virtual machine which needs to set the *object-heap-start* and *object-heap-end* registers according to its allocation policy. This mechanism may however incur a small performance degradation because of the logic to check for object area on every memory access. The modified ISA mechanism saves this overhead but all applications have to be re-compiled to adopt the new instruction set. All our experiments use the former method to simulate the dimultiplexing of references.

6 Experiments

6.1 Experimental Setup

To determine the benefits of our *Object-cache* architecture with respect to reducing energy consumption, we simulated the architecture for a variety of cache configurations using Dynamic Simplescalar cycle level simulator ([10], [9]) running Jikes RVM Java interpreter. Dynamic SimpleScalar (DSS) is a simulator suite for the PPC instruction set. It has Wattch [5] integrated with it for energy measurements. A cycle level system simulator was used to obtain cycle accurate simulation results instead of using the easier, more commonly adopted trace generation and cache simulation based approach which has been used in many previous works ([11], [14]). Cycle level simulation has several advantages over the trace based method for our studies. Firstly, it captures the timing details of memory accesses hence accounting for certain latency hiding optimizations. Secondly, it captures intricate interactions among other units easily. For instance the intricate interactions between the energy and performance of load-store queue unit with cache performance are inherently captured in a system simulation model. Thirdly, system performance results are easily obtained while in the cache simulation based model only the cache miss rate is available which does not necessarily model the performance behavior of the entire program. The only disadvantage of cycle level simulation is its extremely slow speed.

The experimental setup for the simulation experiments is shown in Table 1. The configuration of data cache shown in Table 1 is the default data cache configuration which is used in all our further experiments. We experimented with a variety of common configurations of data cache (no *Object-cache*) and decided on the default configuration by selecting the configuration that gave us the lowest average energy-delay product ([16]) for the SPECjvm98 benchmarks. By this we are making sure that we are not comparing our results with sub-optimal data cache configurations. SPECjvm98 benchmarks

Table 1: Experimental Setup

Cache and Memory Hierarchy	
Simulation Parameter	**Parameter Value**
L1 Data Cache	16 KB, 32 byte blocks, 2-way associative
L1 Instruction Cache	16 KB, 32 byte blocks, 1-way associative
L2 Cache (unified)	1 MB, 32 byte blocks, 4-way associative
Data TLB	32 entries, 30 cycle miss latency
Instruction TLB	16 entries, 30 cycle miss latency
Memory	100 cycle latency

have been used for simulation experiments. S100 full length reference inputs have been used as input. To collect results from a representative part of the program, the first one billion instructions which correspond mostly to the virtual machine loading phase, have been forwarded. All benchmarks have been simulated further for 4 billion instructions.

6.2 Evaluation Metrics

As the proposed *Object-cache* architecture influences cache behavior, performance and energy behavior of programs, the following metrics can be used for a comprehensive evaluation

- **Cache Miss Rate**: Miss rates of object references and non-object references with/without separate caches is a metric that we like to measure because we intend to reduce energy consumption by reducing the miss rates of both the data cache and the object cache relative to datat cache alone.
- **Energy Savings**: Data cache energy savings and total processor energy savings realtive to the default configuration.
- **Performance (IPC - Instructions Per Cycle)**: Cache miss rate is a good indicator of the performance of the cache architecture in isolation. However, execution time is what is accepted as the metric for measuring performance of programs. Instruction count of the simulated benchmarks and the processor clock frequency being constant across the experiments, the execution time of programs is directly proportional

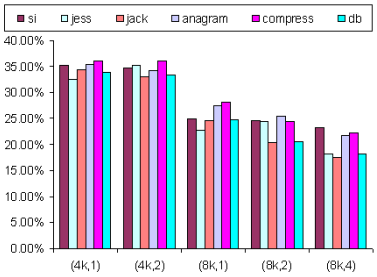

Figure 5: Data cache energy savings across various configurations of *Object-cache*

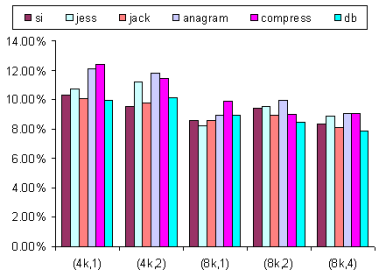

Figure 6: Total processor energy savings across various configurations of *Object-cache*

to the Instructions Per Cycle (IPC) metric and is therefore used as a metric to measure the performance of the benchmark programs.

- **Energy-Delay Product**: In evaluating new architectural features that influence both energy consumption and performance, energy performance trade offs have to be measured. It has been demonstrated ([16]) that the energy-delay product metric causes the architectural improvements that contribute the most to both performance and energy efficiency to stand out. A smaller energy delay product indicates a better configuration. This metric is used to chose statically among the configuration alternatives available.

6.3 Experimental Results

All comparisons were made between the default data cache only configuration (16K, 2-way associative) against a combination of an 8K data cache and a given size of object cache. The default data cache size with the object cache was chosen to be 8K becuase 50% of the references are non object references which need to pass through this data cache. For example, whenever we say we use a (4K,1) object cache, we mean we use a combination of 4KB, 1way associative object cache and an 8KB, 2 way associative data cache (default data cache size in the presence of an object cache).

6.3.1 Energy Results:

Figure 5 depicts the energy savings in data cache relative to our default data cache only configuration for various cache configurations. It is evident that

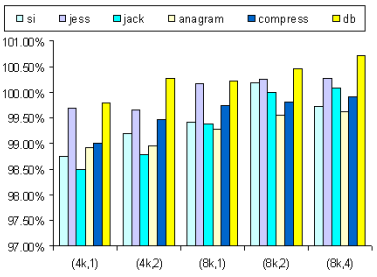

Figure 7: Performance of benchmarks across various configurations of *Object-cache*

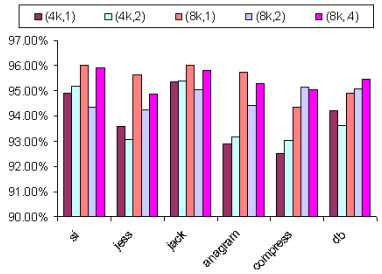

Figure 8: energy-delay products for various configurations of *Object-cache*

the data cache energy savings for any 4 KB *Object-cache* configuration is greater than that of any 8KB configuration because static energy consumption is proportional to the size of data cache. But the relationship is not linear due to the variations in the performance of the benchmark programs with varying cache parameters (size and associativity) which leads to different *execution times* which again is proportional to the energy consumption. Also in most cases, it can be observed that the energy consumption increases with increasing associativity. This is due to the energy required for additional circuitry for a higher degree of associative search. Again variations in performance with associativity plays a role in the total energy consumption.

Our final goal is to reduce the total processor energy consumption. Also, the total processor energy consumption is more sensitive to performance of the cache than the energy consumption of the data cache (or *Object-cache*) itself. The difference in data cache (or *Object-cache*) energy consumption between a miss and a data hit is the dynamic energy associated with loading the new value by replacing the old value while the difference in total processor energy consumption is that of a memory access which is a lot more expensive in terms of energy. Figure 6 depicts the total processor energy savings relative to total processor energy with our default data cache only configuration for various *Object-cache* configurations. The energy savings are found to be in the range of 7.9% to 12.4%. The total processor energy savings also gives us a realistic picture of the savings, because a 10% savings in total processor energy means that the same battery which powered the processor earlier running for ten hours would now last eleven hours.

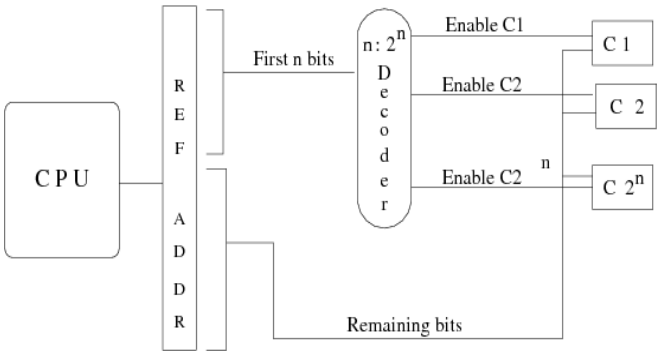

Figure 9: Address range based splitting of memory references

6.3.2 Performance Results:

Figure 7 shows the impact of *Object-cache* architecture on the performance of the benchmarks. Relative performance of the benchmarks with respect to their performance on the default configuration has been reported. One can see from Figure 7 that in many cases, the performance improves from the base case while for others it loses on performance by a small margin (less than 1.5%). As predicted earlier, the performance of a 4KB *Object-cache* (and an 8K data cache makes it a total of 12K cache) is not much different from a 16 KB data cache only configuration or a 8KB *Object-cache* and 8KB Data cache combination. In fact, for benchmarks like *db* the performance even improves with the 4KB 2-way associative *Object-cache* architecture.

To ensure that such behavior was an intrinsic property of object references, and to confirm that the results were not co-incidental with merely splitting the cache into two parts, we conducted various experiments of splitting the memory references into different modules of cache, but in all cases we lost heavily on performance (to the tune of 12 to 20 %). We implemented a mechanism of splitting memory references based on the reference address range into different caches. The mechanism is depicted in Fig. 9. The first n bits of the reference address were used to split the references into 2^n caches so that the spatial locality among references was maintained. Such an architecture led to a performance loss of 14–20% depending on the value of n. This emphasized the significance of temporal locality among object references for the good performance of *Object-cache*.

6.3.3 Miss Rate Analysis

Performance of the cache architecture varies directly with cache miss rates. Having split the cache into two modules, we analyzed both the components

Table 2: Cache Miss rates

	si		*jess*		*jack*		*anagram*		*compress*		*db*	
	data	**obj**	**data**	**obj**	**data**	**obj**	**data**	**obj**	**data**	**obj**	**data**	**obj**
(4k,1)	-1.33%	0.00%	0.13%	-5.26%	0.88%	-2.94%	0.00%	3.70%	0.45%	2.17%	-0.87%	-4.00%
(4k,2)	-1.33%	-8.33%	0.13%	-2.63%	0.88%	-2.94%	0.00%	-3.70%	0.45%	0.00%	-0.87%	-8.00%
(8k,1)	-1.33%	0.00%	0.13%	-7.89%	0.88%	-8.82%	0.00%	-3.70%	0.45%	-2.17%	-0.87%	-12.00%
(8k,2)	-1.33%	-12.50%	0.13%	-13.16%	0.88%	-8.82%	0.00%	-11.11%	0.45%	-4.35%	-0.87%	-16.00%
(8k,4)	-1.33%	-4.17%	0.13%	-10.53%	0.88%	-11.76%	0.00%	-7.41%	0.45%	-4.35%	-0.87%	-20.00%

for their miss rates and compared it against the miss rate for the data-cache-only architecture. Table 2 shows the percentage change in miss rates as compared to the miss rate in our default data cache only configuration(16K). A positive value for the given configuration indicates an increase in miss rate, while a negative value indicates a decrease in miss rate compared to the default configuration. The use of *Object-cache* resulted in an improvement in cache miss rates as expected. This is so mainly due to the properties of object references and the observed fact that object references constitute almost 50% of the total memory references. Also, it is important to see that an *Object-cache* which is of size 4K also reduces miss-rates in many cases. This leads to the observed increase in performance in certain cases. The small increases in miss rates in data cache in certain cases could only be explained as incidental loss of locality.In most other cases, we observe that the miss rates in the data cache also improves (marginally). This is so because, it is least likely that there would be either temporal or spatial locality between object references and non object references but the presence of object references in the same cache could have caused certain conflict misses which are now avoided. The miss rate of data cache does not vary with Object-cache configuration for a given benchmark because the configuration of data cache is fixed before hand. We could have experimented with varying data cache configurations, but chose to avoid the huge increase in number of configurations to be evaluated as we had chosen the data cache architecture based on minimum average energy-delay product.

7 Optimal *Object-cache* Configuration

The results of total processor energy, data cache energy and performance of various benchmarks for a range of *Object-cache* configurations and a huge degree of variations in each, present to us, a confusing picture about the optimal cache configuration that needs to be selected. In order to select such an optimal cache configurations a metric such as energy delay product of the configuration for different benchmarks is of great help. Figure 8 plots the energy delay products of various benchmarks for all the evaluated configurations. Considering the average energy delay product as the metric for choosing the optimal cache configuration, an *Object-cache* configuration of 1-way associative, 4K *Object-cache* with the default 8K, 2 way associative data cache could be selected for the given set of benchmarks.

Although there is a significant amount of energy savings with the configuration having minimum average energy-delay product across benchmark programs, in case of individual benchmarks alternative configurations might just work better. Zhang [6] proposes a highly configurable cache architecture for embedded systems, which enables us to tune cache associativity and cache size (without additional expense of static energy) based on techniques called *way-concatenation* and *way-shutdown*. Fine tuning the configuration of *Object-cache* is possible by using this architecture. To see the benefits of fine tuning the optimal cache configuration, Table 3 plots the range of difference in energy consumption and range of difference in performance between the configurations evaluated. We would like to mention here that these benefits are only in addition to those mentioned in the previous result sections. We observe that energy consumption varies over a much wider range than performance in general. For example, for *jess, anagram* and *compress*, energy consumption varies upto 3.27%, 3.43% and 3.65% respectively while their performance only differs by 0.61%, 0.70% and 0.90%. In embedded systems, where the general execution charecteristics of programs can be determinde by profiling, this suggests that we could go in for an aggressive strategy in chosing the optimal configuration by chosing the lowest energy value. We might need to be conservative only in case the execution charecteristics of the program is similar to that of *si* or *jack* where in the energy benefits range over 2.07% and 2.06% respectively while the performance also varies in the range of 1.46% and 1.61%. One can also see from plot of energy-delay products in Fig. 8 that in case of *si* and *jack* an alternative configuration such as (8k,2) is most beneficial. In essence, we could use the configurable architecture described in [6] to fine tune the configuration of the *Object-cache* aggressively and reap additional energy benefits (of the order of 3% or more) at the cost of less than 0.9% performance.

Table 3: Benefits of Fine Tuning Optimal *Object-cache* Configuration

Benchmark	**Energy Diff**	**Performance Diff**
si	2.07%	1.46%
jess	3.27%	0.61%
jack	2.06%	1.61%
anagram	3.43%	0.70%
compress	3.65%	0.90%
db	2.40%	0.91%

8 Conclusion and Future Work

The paper presents a novel cache architecture for storing objects known as the object- cache. The *Object-cache* exploits certain special properties exhibited by the objects in order to improve energy consumption of the data cache. Impressive results were obtained where in an average 9.7% of total processor energy savings, 35.9% of data cache energy savings were reported at the cost 1% performance gain to 1.5% performance loss for different cache configurations. Energy delay product metric was used to select an optimal cache configuration.

The most important pending issue in the study of *Object-cache* is the determination of the size and associativity of the *Object-cache*. Better than deciding the Object cache parameters statically would be to have a dynamically reconfigurable architecture to adapt the cache size and associativity according to program behavior. A new algorithm for reconfigurability may be needed which would be another interesting topic.

References

[1] Trevor Mudge: Power: A first class design constraint. Computer, 34(4):52-57, April 2001

[2] K. Flautner, N. Kim, S. Martin, D. Blaauw, and T. Mudge: Drowsy caches: Simple techniques for reducing leakage power. In 29th Annual International Symposium on Computer Architecture, pages 148 – 157, May 2002.

[3] V. Zyuban and P. Kogge: Split register file architectures for inherently low power microprocessors. Power Driven Microarchitecture Workshop at ISCA98, June 1998.

[4] J. L. Cruz, A. Gonzalez, M. Valero, and N. P. Topham: Multiple-banked Register File Architectures. In 27th Annual International Symposium on Computer Architecture, 2000, p. 316-325.

[5] D. Brooks, V. Tiwari, and M. Martonosi: Wattch: A framework for architectural-level power analysis and optimizations. In Proceedings of the 27th Annual International Symposium on Computer Architecture, pages 8394, June 2000.

[6] Chuanjun Zhang, Frank Vahid and Walid Najjar: A Highly Configurable Cache Architecture for Low Energy Embedded Systems. ACM Transactions on Embedded Computing Systems (TECS), Volume 4 , Issue 2, May 2005.

[7] S. Kim, S. Tomar, N. Vijaykrishnan, M. Kandemir and M.J. Irwin Use of Local Memory for Efficient Java Execution p. 0468, IEEE International Conference on Computer Design (ICCD'01), 2001.

[8] Bastien Lafond and Johan Lilius An Opcode Level Energy Consumption Model for a Java Virtual Machine. Proceedings of the 3rd Virtual Machine Research and Technology Symposium, May 2004.

[9] D. Burger and T. M. Austin: The SimpleScalar tool set, version 2.0. Technical Report 1342, Computer Sciences Department,University of WisconsinMadison, June 1997.

[10] X. Huang, J. E. B. Moss, K. S. Mckinley, S. Blackburn, and D. Burger: Dynamic SimpleScalar: Simulating Java Virtual Machines. Technical Report TR-03-03, University of Texas at Austin, Feb. 2003.

[11] Greg Wright, Matthew L. Seidl and Mario Wolczko: An object aware memory architecture. SMLI Technical report, February 2005.

[12] S. Wilton and N. P. Jouppi: CACTI: An Enhanced Cache Access and Cycle Time Model. IEEE Journal of Solid-State Circuits, pages 677-687, 1996.

[13] Soontae Kim, N. Vijaykrishnan, Mahmut Kandemir, Anand Sivasubramaniam, Mary Jane Irwin: Partitioned instruction cache architecture for energy efficiency. ACM Transactions on Embedded Computing Systems (TECS). Volume 2,Pages: 163 - 185, May 2003.

[14] Jin-Soo Kim, Yarsun Hsu: Memory System Behavior of Java Programs: Methodology and Analysis. In Proceedings of the 2000 International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2000), June 2000.

[15] Sylvia Dieckmann, Urs Holzle: A Study of the allocation behavior of the SpecJVM98 Java benchmarks. In Proceedings of European Conference on Object Oriented Programming, June 1999.

[16] Gonzalez R, Horowitz M: Energy dissipation in general purpose microprocessors. IEEE Journal of Solid-State Circuits 31, 9 (September 1996), 1277-1284.