An Adaptive, Energy Efficient Object Cache Architecture

S S Shekhar Y N Srikant

IISc-CSA-TR-2006-1
http://archive.csa.iisc.ernet.in/TR/2006/1/

Computer Science and Automation
Indian Institute of Science, India

January 2006

An Adaptive, Energy Efficient Object Cache Architecture

S S Shekhar Y N Srikant

Abstract

An important trend in the design of embedded computing systems is the proliferation of energy conscious architectures. Another important trend is the emergence of Java as a popular platform of choice for implementing various applications that run on mobile and hand held devices. General purpose cache architectures that support Java in embedded systems do not respond well to special properties and behavior of object oriented programs and thus consume a significant fraction of the total processor energy. In this paper we present an adaptive, energy-efficient object cache architecture and a hardware implementable reconfiguration algorithm to alleviate the problem of increased energy consumption of cache architectures in embedded systems that typically support runtime environments for object oriented programs.

An *Object-cache* architecture consists of a data cache (reduced in size) and an additional small cache structure that caches only the objects. Certain special properties of object oriented programs such as a high degree of temporal locality among object references and a short life span of 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. However, both energy behavior and performance vary widely with the configuration of the *Object-cache*. Thus we propose an adaptive object cache architecture and a reconfiguration algorithm for dynamically reconfiguring the size and associativities of the *Object-cache* such that it obtains the best possible energy efficiency with the least impact on performance. Using applications from SPECjvm98 benchmarks and a cycle accurate simulation, the adaptive *Object-cache* architecture is shown to reduce average data cache energy consumption by up to 43% and also improve performance by about 1%.

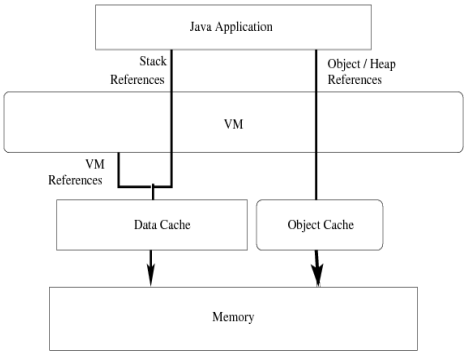

Figure 1: Memory hierarchy with Object Cache

1 Introduction

In our previous work [2], we discuss the impact of an Object-cache on the energy consumption of the memory hierarchy and the overall system energy consumption. As discussed in this work, 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*. 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. 1. The object references are directed through the *Object-cache* while the other references are directed through a regular data cache. Demultiplexing of the references to their respective caches could be carried out by the hardware. Two additional registers, *object-heap-start* and *object-heap-end* are used to bind the *object area* on the heap where objects are allocated. Any reference that is addressed to this *object area* is cached in the *Object-cache* by the hardware while the remaining references are directed through the regular data cache. 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 performance and energy behavior of the benchmark programs was found to be extremely sensitive to the configuration of the *Object-cache*, when the optimal configuration was determined on a per-application basis. The realtionship between the configuration and performance and energy charecteristics of a program (of the entire application) can be explained as follows. A direct mapped *Object-cache* is more energy efficient per access, consuming only about 30% of the energy of a similar sized four-way set associative cache

[9]. This reduction occurs because a direct mapped cache accesses only one tag and data array per access, while a four-way cache accesses four tag and data arrays per access. A direct mapped cache can also have a shorter access time in part because multiple data arrays need not be multiplexed. While a direct-mapped caches hit rate may be acceptable for many applications, for some applications a direct-mapped cache exhibits a very poor hit rate and hence suffers from poor performance and energy consumption. Adding set associativity increases the range of applications for which a cache has a good hit rate, but for many applications, the additional associativity is unnecessary and thus results in wasted energy and longer access time. Deciding on a Object caches total size involves a similar dilemma. A small Object cache is more energy efficient and has a good hit rate for a majority of applications, but a larger cache increases the range of applications displaying a good hit rate, at the expense of wasted energy for many applications.

Recent research on this area has focussed on configurable cache architecture designs. However, most of these works seem to ignore the issue of selecting the right configuration at the right time. Merely having a configurable cache is not sufficient until we have a mechanism to decide on the right configuration. The major problem involved in deciding the optimal configuration for the object cache is that the applications are dynamically downloaded and their behavior is not predictable. Also, each application has a dynamically varying cache configuration requirement over the period of execution. Setting a single configuration for the whole application execution, specially an application with unknown charecteristics (such as Java programs downloaded on mobiles or PDAs) does not result in either optimal energy or optimal performance results.

In this paper, we describe an adaptive *Object-cache* architecture that dynamically tunes itself to an optimal configuration. The granularity of tuning the configuration during program execution was given due importance and with an eye on minimizing the overhead involved in the tuning process. A direct relationship between program phase and cache configuration was observed. Whenever the program execution phase changed, it was necessary to change the configuration and it was also observed that there was no necessity to change the configuration when a program is executing in the same phase. We use a simple hardware based scheme for detecting phase changes and further study the effectiveness of such an architecture by evaluating it with an algorithm that choses the ideal configuration and propose a simple algorithm which approximates the ideal case to decide on an optimal configuration. Energy consumption and performance behavior of typical Java programs such as the SPECjvm98 benchmarks are evaluated on the architecture with the above mentioned algorithms.

The remainder of the paper is organized as follows. Section 2 discusses some related work. Section 3 motivates the need for adaptivity and a phase level granularity for configuring the *Object-cache*. In Section 4, architecture of the Adaptive *Object-cache* is discussed in detail. The experimental details, simulation methodology and experimental results are described in Section 5. An ideal reconfiguration algorithm is proposed in Section 6. Section 7 describes a simple algorithm for reconfiguring the object cache and approximating the ideal reconfiguration algorithm. Some concluding remarks are provided in Section 8.

2 Related Work

Energy benfits of an anotation based allocation of objects in a local memory is studied in [7]. They propose an object allocation strategy to reduce the energy consumption of Java applications. This object allocation strategy uses a part of the on-chip memory resources as a local memory to achieve better performance than a cache only architecture. A few chosen objects are permanently allocated by the virtual machine on the local memory. The object allocation strategy uses an annotation based approach where in Java classes are annotated based on profile data from the execution of programs.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.

A phase in a program is an interval of execution during which a measured program metric is relatively stable. Recently, it has been shown that many programs execute as a series of phases, where each phase may be very different from the others, while still having a fairly homogeneous behavior within a phase. Taking advantage of this time varying behavior can lead to, among other things, improved power management, cache control, and more efficient simulation. Sherwood [1] proposes an efficient run-time phase tracking architecture that is based on detecting changes in the proportions of the code being executed. Since this phase tracking implementation is based upon code execution frequencies, it is independent of any individual architecture metric. Independence from architecture metrics allows us to consistently track phase information as the programs behavior changes due to phase based optimizations.

A highly configurbale cache architecture was proposed by Zhang in [8].

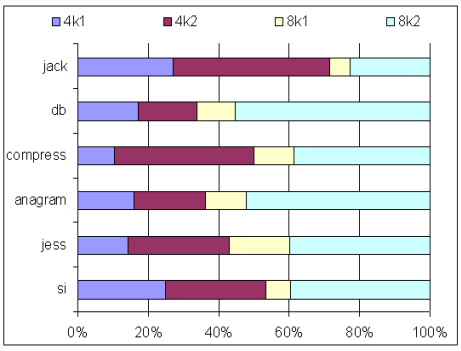

Figure 2: Algorithm for finding the ideal configuration

Many cache parameters like cache size, cache line size and associativity can be tuned dynamically in this architecture. The configuration of associativity is implemented by a technique we call way concatenation. The register value directly acts as a control to the circuitry involved in the cache. Tuning of cache size is also permitted through way shutdown. The shutdown logic uses sleep transistors to reduce static power dissipation. We use this architecture for the Object cache to reconfigure the cache along with the phase detection hardware to decide when to reconfigure and implement our hardware based reconfigurability algorithms in order to reduce energy consumption.

Pokam [3] proposes a compiler scheme for phase based cache resizing which allows the compiler to detect phase changes and add additional instructions to change the configuration. The compiler directed reconfiguration is not a possibility for an Object-cache, as we target to run the dynamically downlaoded applications whose bytecodes cannot be easily modified Thus, our work is different from this work in the sense that we propose a totally hardware based reconfiguration scheme including phase change detection and reconfiguration along with a simple algorithm to choose the optimal configuration.

3 Motivation

As we stated earlier, the energy consumption per access of the cache is a direct function of its configuration. It increases almost linearly with cache size and associativity. For example, A direct mapped *Object-cache* is more energy efficient per access, consuming only about 30% of the energy of a similar sized four-way set associative cache and a cache which is half the size as the other consumes about 50% of the energy as the later. At the

same time some applications (or parts of applications to be more precise) require higher end configurations for good performance. By having a fixed configuration through out the execution of an application we fail to capitalize on the varying requirements of parts of applications.

Pokam [3] suggests that we could use a phase as a unit of execution for reconfiguration. In order to verify this, we conducted experiments to find an optimal configuration (among 4 selected configurations) in terms of the energy consumption for every interval of 10 Million instructions (Any interval lesser than this for a reconfiguration might hurt the performance). Figure 2 plots the percentage of the total number of intervals that had a given configuration as the optimal energy configuration. About 40% of the intervals have optimal configuration as one of the 4K cache configurations while the rest are 8K cache configurations. It clearly indicates that there is no single optimal energy configuration across the intervals. Also it was observed that, in 73% of the intervals, the optimal energy (lowest) configuration was also the optimal performance (highest) yielding configuration for that particular interval. It is evident because the energy condumption of a module is itself dependant on the performance and if performance decreases energy consumption would increase. Further phase information collected also suggested that, the optimal energy configuration at the start of the phase was the optimal one in 87% (on an average across the benchmarks) of the intervals within that phase. Thus we need not reconfigure the Object cache within a single phase (other than in the beginning). Also, an optimal configuration change occured in 84%(on an average across the benchmarks) of the phase changes that occured suggesting that we need to reconfigure the Object cache on the onset of a new phase. Thus our reconfiguration algorithms need to focus on reconfiguring once for every phase change.

4 Adaptive Cache Architecture Design

Our hardware based reconfiguration setup builds upon the highly reconfigurable cache architecture described by Zhang in [8] and uses a simplified version of phase tracking and prediction hardware proposed by Sherwood [1] to detect phase changes. The structure of the phase tracking and prediction hardware proposed in [1] is shown in Figure 3. The key idea in this hardware is to capture basic block information during execution, while not relying on any compiler support. Larger basic blocks need to be weighed more heavily as they account for a more significant portion of the execution. To approximate gathering basic block information, branch PCs and the number of instructions executed between branches is captured. The input to the architecture

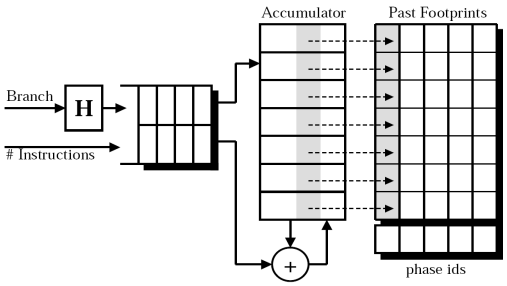

Figure 3: Phase Tracking and Prediction Hardware

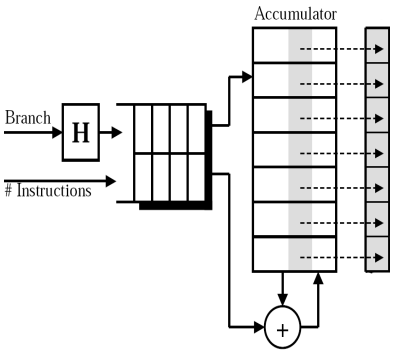

Figure 4: Phase Change Detection Hardware

is a tuple of information: a branch identifier (PC) and the number of instructions since the last branch PC was executed. This allows to roughly capture each basic block executed along with the weight of the basic block in terms of the number of instructions executed. Such Footprints of execution are calculated over an interval and whenever the Basic Block vector differs from those in the Past Foot Prints Table by a certain distance a new phase is said to have occured.

The phase tracking and prediction hardware needs to store footprints of many previous phases so that it could predict the phase that is likely to occur next assuming the same kind of phases occur repeatedly over the period of execution of the program. We eliminate the *Past Foot prints Table* and *Phase ID Table* and instead store the footprint information of the previous phase only as we only need to detect a phase change. The *Phase Change Detection Hardware* is shown in Figure 4. A threshold of 1 Million, manhattan distance between the current basic block vector and previous phase's footprint has been experimentally determined to be suitable for an interval of execution of 10 Million instructions as presented in [1]. We also have found it reasonable and used the same threshold and interval size in our experiments. Whenever the Phase Change Detection hardware detects a phase change, a reconfiguration algorithm needs to determine the optimal configuration of the Object cache for the new phase.

5 Experimental Setup

To determine the benefits of our adaptive *Object-cache* architecture with respect to reducing energy consumption, we simulated the architecture using Dynamic Simplescalar cycle level simulator ([6], [5]) running Jikes RVM Java

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)	512 KB, 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

interpreter. Dynamic SimpleScalar (DSS) is a simulator suite for the PPC instruction set. It has Wattch [4] integrated with it for energy measurements. Cycle level simulation has several advantages 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 only configuration in all our experiments. Whenever we refer to an Object cache configuration, the data cache size is reduced to 8K and the Object cache is added whose configuration would be mentioned. For better clarity about the results obtained, we compare our results against both data cache only configuration and static (non-adaptive) object cache configurations. SPECjvm98 benchmarks have been used for simulation experiments. S100 full length reference inputs have been used as input.

6 Ideal Reconfiguration Algorithm

We use four alternative configurations for the adaptive object cache (4K and 8K direct mapped configurations and 4K and 8K 2way associative configu-

```
Ideal_conf_select ( )
{
    Sort all configurations in ascending order of their energy efficiency for the last interval

    best_perf = best performance ( 1 / least cycles ) among various configurations
                for the last interval.

    ideal_conf = configuration with lowest energy.

    while ( Ideal Configuration not found ) {

        If ( (best_perf / Performance (ideal_conf) )< (1 + PERF_THRESHOLD) ) {

            Ideal Configuration Found

            break;
        }

        ideal_conf = next higher energy configuration
    }

}
```

Figure 5: Algorithm for finding the ideal configuration

rations). The job of the reconfiguration algorithm is to select one optimal configuration out of these so as to minimize energy consumption for the execution of the program (not on a per access basis) and maintain the performance without allowing for any significant degradation in the performance. By carefully selecting the optimal configuration which consumes least energy with a permissible degradation in performance, we can achieve optimal energy results.

In order to control the worst case degradation in performance, we designed an ideal algorithm for highest energy savings with *worst case performance degradation* as a parameter. The simple but idealistic algorithm is presented in Figure 5. The algorithm has *PERF_THRESHOLD* as a parameter that stands for the maximum permissible percentage performance degradation as compared to the best achievable performance with the given resources. The algorithm proceeds by finding the least energy consuming configuration, with performance above the permissible threshold, for each phase. If the lowest energy configuration leads to a performance degradation of more than the threshold, it then selects the next higher energy configuration until the performance degradation is less than the allowed limit. The worst case of the algorithm is when the highest energy configuration is the configuration that yields the best performance and all other configurations lead to performance degradations above the threshold value. Since we select the ideal configuration for each phase, we obtain optimal results for the entire program execution.

The data cache energy savings with Ideal reconfiguration algorithm as

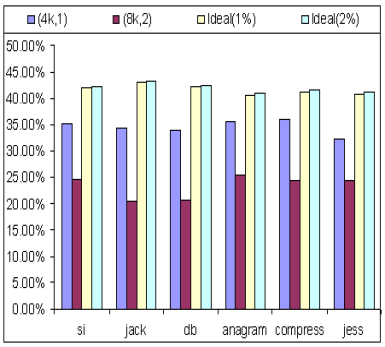

Figure 6: Data cache energy savings for Ideal reconfiguration algorithm

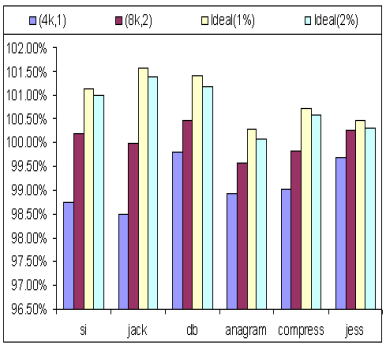

Figure 7: Performance behavior for Ideal reconfiguration algorithm

compared to static object cache configurations of 4K direct mapped and 8K 2 way associative caches is shown in Figure 6. All energy savings are with respect to the data cache only configuration. The energy savings with Ideal reconfiguration algorithm is as high as 43% while that of even a static 4K direct mapped configuration is about 35% and that of a static 8K 2 way associative configuration is about 25%. The additional energy savings as compared to even the 4K configuration, comes from the fact that the performance is not allowed to degrade beyond the given degradation limit in case of Ideal reconfiguration algorithm (actually in all cases it gains on performance as shown in Figure 7). Whenever performance is likely to be hurt by the 4K configuration, the Ideal reconfiguration algorithm shifts to a higher end configuration and maintins good performance, inturn leading to good overall energy behavior.

We set the worst case percentage performance degradation threshold to 1% and 2%, so that we could obtain performance results that are atleast as good as the optimal single configuration case [2]. But the results of our experiments indicated much better results. The performance behavior of various benchmark programs with Ideal reconfiguration algorithm compared against static object cache configurations of 4K direct mapped and 8K 2 way associative caches is shown in Figure 7. The performance of benchmarks on data cache only configuration is our reference (100%). In both cases (1% and 2% for *PERF_THRESHOLD*), we actually observed a performance gain of about 1% as compared to data cache only configuration and an improvement even over the 8K, 2way associative static configuration for the Object-cache. In every phase we choose the ideal configuration in terms of energy which has performance better than the *PERF_THRESHOLD* degradation limit.

Optimal energy configuration coincides with the optimal performance configuration for more than 73% of the intervals, this results in overestimating the performance threshold, without leading to any additional energy consumption at almost every phase. The combined effect of this is visible in the form of the observed incerase in performance.

In both energy and performance, we observe that the difference between Ideal reconfiguration algorithm with *PERF_THRESHOLD* as 1% and 2% is minimal. This can also be explained by the fact that, for more than 73% of the intervals, optimal energy and optimal performance configurations coincide. This configuration is going to have the best performance and energy. Therefore, the selected configuration is anyway going to remain the same in case of 1% and 2% performance degradation limits. The small differences that occur happen due to alternate configurations chosen among the other intervals (actually phases). This observation that optimal energy and performance configurations coincide in a majority of phases, leads us to think of approximating the ideal reconfiguration algorithm, in a realistic hardware implementable manner.

7 Approximation Using Energy Delay Product

The Ideal reconfiguration algorithm is hard to be implemented in hardware as it requires sorting of all the configurations in ascending order of their energy consumptions. Further, the complexity of the hardware for sorting values might not be scalable as it may turn out to be prohibitively high for a set of eight alternative configurations. To overcome this complexity in implementing the algorithm we need to device an approximation algorithm that closely tracks the reconfiguration mechanism of the Ideal reconfiguration algorithm and is scalable. As mentioned earlier, the observation that most of the optimal energy configurations coincide with the optimal performance configuration which inturn is the configuration that needs to be chosen, indicating that the energy-delay product metric is a good choice for approximating the Ideal reconfiguration algorithm as energy delay product would be lowest for such configurations.

In evaluating new architectural features that influence both energy consumption and performance, energy performance trade offs have to be measured. It has been demonstrated by Horowitz([10]) 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 en-

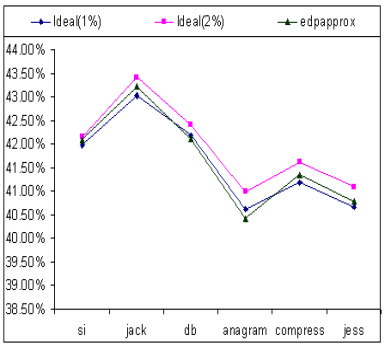

Figure 8: Data cache energy savings of our energy delay product based approximation algorithm relative to data cache only configuration

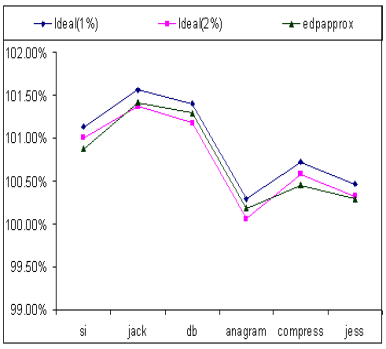

Figure 9: Performance of our energy delay product based approximation algorithm relative to data cache only configuration

ergy delay product indicates a better configuration. We could use an energy model similar to that used in [3] to keep track of the energy while merely counting the number of cycles taken to execute for completing an interval of 10 Million instructions is sufficient to keep track of the performance. Once the energy values and delays are evaluated for each configuration, in the beginning of a phase, they are simply multiplied and the configuration for which the product is lowest is selected for the remaining part of the phase.

As observed in our experiments, whenever an optimal energy configuration (low energy consumption) coincides with optimal performance configuration (low delay), energy delay product is at its lowest. Thus the selected configuration by the lowest energy delay product approximation is the same as Ideal reconfiguration algorithm in atleast 73% of the cases. In fact, the approximation turns out to be tracking the ideal reconfiguration algorithm to a good accuracy (to an accuracy of less than 1% difference) as depicted in Figure 8 and Figure 9 in terms of both energy and performance. In case of the energy delay product based approximation, we observe average energy consumption savings of 41.5% and an improvement of about 0.87% as compared to data cache only configuration.

8 Conclusion

The paper presents an adaptive cache architecture for the *Object-cache*. Potential energy benefit by reconfiguring the cache at the granularity of a phase

of execution is exploited by phase change detection hardware and a simple reconfiguration algorithm. The hardware based reconfiguration mechanism for energy efficiency ensures that the optimization is benefitial to the dynamically downloaded Java applications. It is also shown that our simple algorithm based on energy delay product approximates the ideal configuration selecting algorithm to a good accuracy in both energy savings and perofrmance. Impressive results were obtained where in 43% of data cache energy savings were reported along with a 1% performance gain.

References

[1] Timothy Sherwood, Suleyman Sair, Brad Calder. Phase Tracking and Prediction. In Proceedings of the 30th International Symposium on Computer Architecture (ISCA), June 2003.

[2] Shekhar S S, Y N Srikant Object Cache : An energy efficient cache architecture http://archive.csa.iisc.ernet.in/TR/2005/13 Technical Report, IISc, October, 2005.

[3] Gilles Pokam, Franois Bodin. Energy-efficiency potential of a phase-based cache resizing scheme for embedded systems Proceedings of the 8th IEEE AnnualWorskhop on Interaction between Compilers and Computer Architectures (INTERACT-8).

[4] 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.

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

[6] 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.

[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] 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.

[9] 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.

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