A Self-Tuning Cache Architecture for Embedded Systems

CHUANJUN ZHANG, FRANK VAHID, and ROMAN LYSECKY
University of California, Riverside

Memory accesses often account for about half of a microprocessor system's power consumption. Customizing a microprocessor cache's total size, line size, and associativity to a particular program is well known to have tremendous benefits for performance and power. Customizing caches has until recently been restricted to core-based flows, in which a new chip will be fabricated. However, several configurable cache architectures have been proposed recently for use in prefabricated microprocessor platforms. Tuning those caches to a program is still, however, a cumbersome task left for designers, assisted in part by recent computer-aided design (CAD) tuning aids. We propose to move that CAD on-chip, which can greatly increase the acceptance of tunable caches. We introduce on-chip hardware implementing an efficient cache tuning heuristic that can automatically, transparently, and dynamically tune the cache to an executing program. Our heuristic seeks not only to reduce the number of configurations that must be examined, but also traverses the search space in a way that minimizes costly cache flushes. By simulating numerous Powerstone and MediaBench benchmarks, we show that such a dynamic self-tuning cache saves on average 40% of total memory access energy over a standard nontuned reference cache.

Categories and Subject Descriptors: B.3 [**Memory Structures**]: Design Styles—*Cache memories*

General Terms: Design, Experimentation, Performance

Additional Key Words and Phrases: Cache, configurable, architecture tuning, low power, low energy, embedded systems, on-chip CAD, dynamic optimization

1. INTRODUCTION

Using a prefabricated microprocessor platform in an embedded system product provides strong time-to-market advantages over fabricating an application-specific integrated circuit (ASIC). With on-chip configurable logic available on many platforms today, the attractiveness of prefabricated platforms over ASICs expands to even more situations. A drawback of a prefabricated platform is that key architectural features, such as cache size, cannot be synthesized such that

This research was supported by the National Science Foundation (grants CCR-0203829) and by the Semiconductor Research Corporation.

Authors' addresses: C. Zhang, Department of Electrical Engineering, University of California, Riverside, CA 92521; F. Vahid and R. Lysecky, Department of Computer Science and Engineering, University of California, Riverside, CA 92521.

Frank Vahid is also with the Center for Embedded Computer Systems at UC Irvine.

Permission to make digital/hard copy of part of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication, and its date of appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permision and/or a fee.
© 2004 ACM 1539-9087/04/0500-0407 $5.00

they are tuned to the application. While microprocessor manufacturers could previously provide a variety of prefabricated ICs spanning the continuum of desired architectures, providing such variety becomes increasingly difficult as microprocessors coexist with numerous other coprocessors, configurable logic, peripherals, and so on, in today's era of system-on-a-chip platforms.

A solution is for key architectural features to be designed with built-in configurability, enabling designers to configure those features to a particular application. Motorola's M*CORE designers [Malik et al. 2000] incorporated a configurable unified set-associative cache whose four ways could be individually shutdown to reduce dynamic power during cache accesses. Additionally, the M*CORE's cache can be configured as an instruction cache, data cache, or unified cache. The M*CORE's designers showed an average 30% of total power savings by tuning the cache for their Powerstone benchmarks and as much as 40% savings on certain benchmarks. Albonesi [1999] also proposed a configurable cache whose ways could be shutdown to reduce dynamic power, showing an average 40% savings in overall cache energy dissipation on Spec95 benchmarks through tuning. Veidenbaum et al. [1999] proposed a cache whose line sizes could be adaptively modified, resulting in a reduction in memory traffic by over 50%. Balasubramonian et al. [2000] proposed a cache that could be configured as a single or as a two-level cache, achieving savings of 43% of memory hierarchy energy when their configurable cache is used as an L2/L3 cache coupled with a conventional L1 cache.

We have designed a highly configurable cache with three parameters that designers can configure: total size (8, 4, or 2 Kbytes), associativity (4, 2, or 1-way for 8 Kbytes, 2 or 1-way for 4 Kbytes, and 1-way only for 2 Kbytes), and line size (64, 32, or 16 bytes). We will briefly discuss our configurable cache in Section 2; the detailed design can be found in Zhang et al. [2003a, 2003b].

Tuning is presently a cumbersome task imposed on the designer, who in most cases must manually determine the best configuration. A designer can use simulation to determine the best cache, but such simulation is often cumbersome to setup, since modeling a system's environment can be harder than modeling the system itself. Simulations can also be extremely slow, requiring tens of hours or days to simulate just seconds of an application, and represents an extra step in the designer's tool flow. Furthermore, simulating an application typically uses a fixed set of input data during execution. Such a simulation approach cannot capture actual run-time behavior where the data changes dynamically. Recently, some design automation aids have evolved to assist the designer in the tuning task [Givargis et al. 2002]. While tuning fits into existing hardware design flows reasonably well, such simulation-based tuning does not fit in well with standard, well-established embedded software design flows, which instead primarily consist of compile, download, and execute.

Several researchers have proposed dynamically tuning cache parameters. Veidenbaum et al. [1999] used an adaptive strategy to adjust cache line size dynamically to an application. Kaxiras's cache line decay method [Kaxiras et al. 2001] dynamically turns off cache lines that have not been visited for a designated period, reducing the leakage energy dissipation. Albonesi et al. [1999] proposed dynamically turning off cache ways to reduce dynamic

energy dissipation. Balasubramonian et al. [2000] dynamically detects the phase change of an application and configures the hierarchy of the caches to improve the memory hierarchy performance and therefore reduce dynamic energy dissipation. However, these dynamic strategies each manipulate only one cache parameter, like cache line size, cache size, and cache hierarchy. Based on monitoring some predetermined criteria, such as cache miss rate and memory-to-L1 cache data traffic volume [Veidenbaum et al. 1999], the time interval between two visits to a cache line [Kaxiras et al. 2001], the instruction per cycle (IPC) [Albonesi et al. 1999], and miss rate, IPC, and branch frequency [Balasubramonian et al. 2000], these dynamic strategies increase/decrease or turn on/off the single aspect of the cache that is tunable.

In our work, we tune not one but four cache parameters: cache line size, cache size, associativity, and cache way prediction. The space of configurations is much larger, and hence we propose a method of dynamically tuning the cache in a very efficient manner. Our method uses some additional on-chip hardware that dynamically tunes our configurable cache to an executing program. The tuning could be applied using different approaches, perhaps being applied only during a special software-selected tuning mode, during the startup of a task, whenever a program phase change is detected, or at fixed periods. The choice of approach is orthogonal to our design of the self-tuning architecture itself.

In this paper, we introduce the tuning problem, describe our approach to developing an efficient heuristic for searching the configuration space, and provide experimental results showing the effectiveness of our methods. The paper is organized as follows. We briefly describe our configurable cache architecture and benefits in terms of energy dissipation in Section 2. In Section 3, we describe our self-tuning strategy involving a search heuristic. We provide the results of our search heuristic in Section 4. We conclude the paper in Section 5.

2. CONFIGURABLE CACHE ARCHITECTURE

Our configurable cache architecture includes four parameters: cache size, which can be configured as 8 Kbytes, 4 Kbytes, or 2 Kbytes; associativity, which can be 4-way, 2-way, or 1-way (direct mapped); line size, which can be 16 bytes, 32 bytes, or 64 bytes; and way prediction, which can be turned off or on (but is always off if the cache is configured as direct mapped).

2.1 Associativity: Way Concatenation

The configuration of associativity is implemented by a technique we call *way concatenation*, as shown in Figure 1. The base cache consists of four banks that can operate as four ways. By configuring a small register, the ways can be effectively "concatenated," resulting in either a two-way or direct-mapped 8 Kbytes cache. The way concatenation logic is very simple, consisting of eight logic gates. The four outputs of the configure circuit, c0, c1, c2, and c3, control the opening and closing of each way. We replaced the two inverters used in a traditional cache by NAND gates. One inverter is the word line driver; the other inverter is the output inverter of the comparator. Based on our layout in 0.18-μm CMOS technology, by appropriately sizing the NAND gates we can

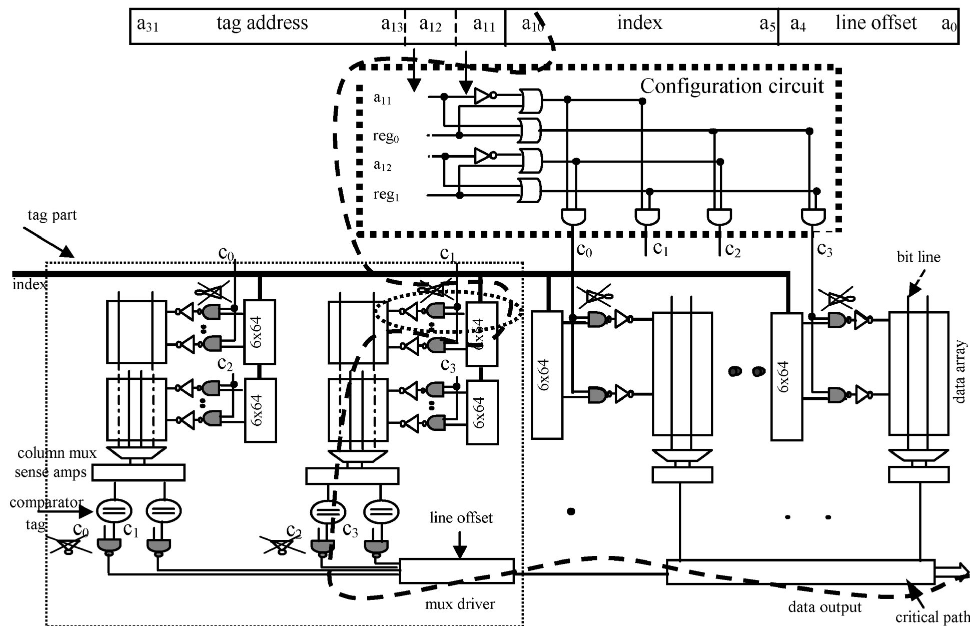

Fig. 1. A way-concatenable four-way set-associative cache architecture with the critical path shown.

ensure that our changes do not impact the cache's critical path. The area overhead of the configuration circuit is negligible compared with the total cache area [Zhang et al. 2003]. It should be noted that full tags (address bits from a_{11} to a_{31}) are checked when the associativity is configured as 1-way, 2-way, or 4-way. Although two bits of the tag do not need to be checked when the cache is configured as direct mapped, the overhead of checking those bits is negligible and a full tag check simplifies the cache's control circuits.

2.2 Cache Size: Way Shutdown

We also permit the tuning of cache size through way shutdown. Individual ways can be shutdown, resulting in a 4 Kbytes cache that can be either 2-way or direct mapped, or a 2 Kbytes direct mapped cache. The shutdown logic uses sleep transistors to reduce static power dissipation, which is becoming increasingly important [Agarwal et al. 2002], and increases the critical path by roughly 5% with less than a 1% increase in area.

2.3 Line Size: Line Concatenation

Designers can also select the cache line size as 16, 32, or 64 bytes, by configuring a small register in the cache controller. We implement line size configurability using a base physical line size of 16 bytes, with the larger line sizes implemented logically as multiple physical lines [Zhang et al. 2003], as illustrated in Figure 2.

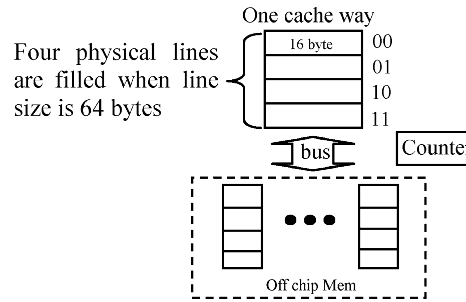

Fig. 2. Architecture of a line size configurable cache.

2.4 Way Prediction

When multiple ways are activated, we can also configure the cache to support way prediction. Way prediction first accesses only one way of the cache. If there is a miss in the first way, the cache will then access the remaining ways. Way prediction [Inoue et al. 1999; Powell et al. 2001] saves energy of per cache access by only accessing one way initially, with the drawback of requiring an extra cycle when there is a misprediction.

2.5 Energy Evaluation

Power dissipation in CMOS circuits is comprised of two main components, static power dissipation due to leakage current and dynamic power dissipation due to logic switching current and the charging and discharging of the load capacitance. Dynamic energy consumption contributes to most of the total energy dissipation in micrometer-scale technologies, but static energy dissipation will contribute an increasingly larger portion of total energy consumption in nanometer-scale technologies. Therefore, we consider both types of energies.

We should not disregard energy consumption due to accessing off-chip memory, since fetching instructions and data from off-chip memory is energy costly because of the high off-chip capacitance and large off-chip memory storage. Additionally, when accessing the off-chip memory, the microprocessor may stall while waiting for the instruction and/or data, and such waiting still consumes some energy. Thus, we calculate the total energy due to memory accesses using equation (1), which considers all these factors. We compute the energy dissipation of the cache tuner using equation (2).

$$\begin{aligned} E_{total} &= E_{dynamic} + E_{static} \\ E_{dynamic} &= Cache_{total} * E_{hit} + Cache_{Misses} * E_{miss} \\ E_{miss} &= E_{offchip_access} + E_{uP_stall} + E_{cache_block_fill} \\ E_{static} &= Cycles_{total} * E_{static_per_cycle} \end{aligned} \tag{1}$$

$$E_{tuner} = P_{tuner} * Time_{total} * NumSearch \tag{2}$$

To see the potential benefits of tuning a cache to an application, we simulated all possible configurations of our configurable cache for numerous

Powerstone [Malik et al. 2000] and MediaBench [Lee et al. 1997] benchmarks using SimpleScalar [Burger et al. 1997], a cycle-accurate simulator that includes a MIPS-like microprocessor model, to obtain the total cache accesses, $Cache_{total}$, and cache misses, $Cache_{misses}$. We obtained the energy of a cache hit, E_{hit}, from our own CMOS 0.18-μm layout of our configurable cache (incidentally, we found our energy values correspond closely with CACTI [Reinman 1999] values). We obtained the off-chip memory access energy, $E_{off_chip_access}$, from a standard Samsung memory and the stall energy, E_{uP_stall}, from a 0.18-μm MIPS microprocessor. Our total energy, E_{total}, captures all energy related to memory accesses, which is the value of interest when configuring the cache. Furthermore, we obtained the power consumed by our cache tuner, which we will describe later, through simulation of a synthesized version of our cache tuner written in VHDL. From the simulation, we also obtained the time required by the tuner to search the cache configurations.

Some researchers have done work on building analytical cache performance models. Agarwal et al. [1989] proposed an analytical model of cache performance, which uses parameters extracted from address traces of programs to estimate the cache performance at different cache parameters. Specifically, the miss rate of a given trace for a program is described as a function of cache size, associativity, line size, and other parameters. The analytical cache performance model not only saves simulation time, which may be prohibitively long, but also provides insight to the dependence of miss rate on the program and workload parameters. As far as we know, we have not seen an analytical model of energy dissipation for a specific application at different cache parameters. CACTI [Reinman et al. 1999] only provides an analytical energy model of per cache access. To get the energy dissipation for a specific application, we must run simulations to obtain the necessary information, such as miss rate and number of accesses to both instruction and data caches. One possible way to simplify the energy calculation is to combine these analytical cache models with the CACTI model, obtaining the miss rate from the cache performance models and energy dissipation per cache access from CACTI. However, we cannot use these cache performance models on-chip to dynamically predict the performance. While developing a model capable of dynamically estimating the performance and energy dissipation of a specific application at varying cache parameters could be very useful in a variety of applications, this will be left as future work.

2.6 Benefits of a Configurable Cache

As illustrated in Figure 3, the energy benefits of optimally tuning a cache to an application is quite significant, resulting in an average of over 40% memory access energy savings for Powerstone and MediaBench benchmarks, and up to 70% on certain benchmarks. While certain cache configurations will result in a decrease in performance, our configurable cache typically only results in a performance overhead of less than 2%.

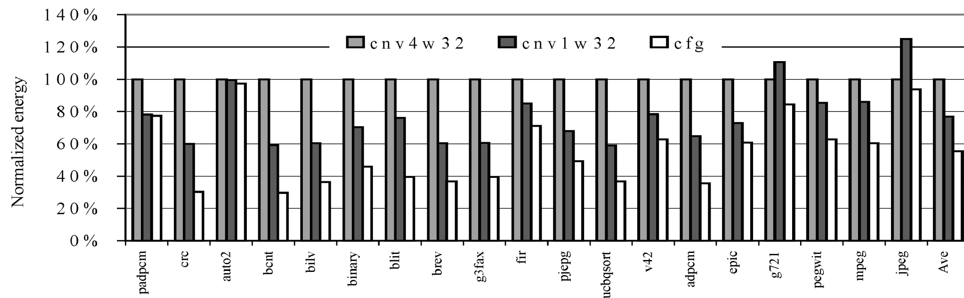

Fig. 3. Memory access energy of Powerstone and MediaBench benchmarks, for our 8 Kbytes configurable cache (cfg) tuned to the best configuration for each benchmark, compared to a conventional 4-way set-associative 8 Kbytes cache with a line size of 32 bytes (cnv4w32) and a conventional direct mapped 8 Kbytes cache with a line size of 32 bytes (cnv1w32). Energies are normalized to the cnv4w32 value for each example. Note that tuning the cache to each example (cfg) yields very significant energy reductions.

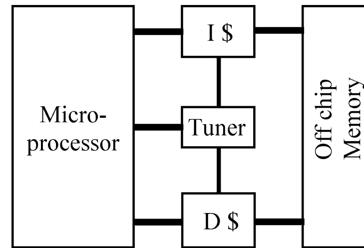

Fig. 4. Cache self-tuning hardware.

3. SELF-TUNING STRATEGY

3.1 Problem Overview

Given the many different possible configurations of our cache, our goal is to automatically tune a configurable cache dynamically as an application executes, thus eliminating the need for tuning via simulation or manual platform configuration and measurement. We accomplish this using a small amount of additional hardware, as shown in Figure 4, that can be enabled and disabled by software. Our goal is for the tuning process and required additional hardware to be as size, power, and performance efficient as possible.

A naive tuning approach exhaustively tries all possible cache configurations, in some arbitrary order. For each configuration, the approach measures the cache miss rate and estimates a configuration's energy from this miss rate. After trying all configurations, the approach selects the lowest energy configuration seen. Such an exhaustive approach has two main drawbacks. First, an exhaustive search method may involve too many configurations. While our configurable cache has 27 configurations, increasing the number of values of each parameter could easily result in over 100 configurations. Consider also that many other components within the system may have configurable settings as well—such as a second level of cache, a bus, and even the microprocessor

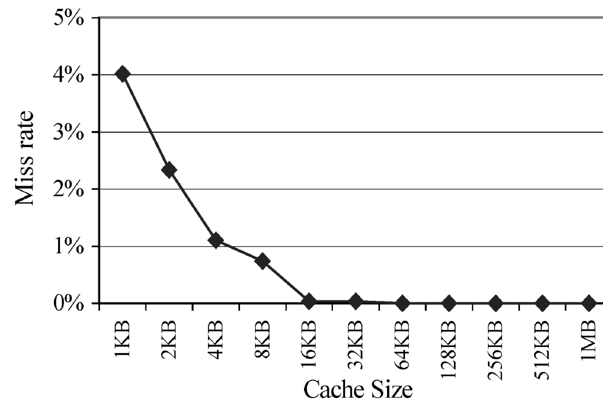

Fig. 5. Miss rate of benchmark *parser* at cache sizes from 1 Kbytes to 1 Mbytes.

itself. If we tune our system by considering all possible configurations, the number of configurations of our cache multiplies the configuration numbers for other components, quickly reaching millions of possible configurations (e.g., $100 \times 100 \times 100 = 1{,}000{,}000$). Thus, we need an approach that minimizes the number of configurations examined. The second drawback is that the naive approach may require too many cache flushes, which are very time and power costly. Without flushing, the new cache configuration could yield incorrect results.

Therefore, we want to develop a tuning heuristic that minimizes the number of cache configurations examined and minimizes or eliminates cache flushing, while still finding a near-optimal cache configuration

3.2 Heuristic Development Through Analysis

We develop the heuristic through analyzing the impaction of each cache parameter to the energy dissipation. From equation (1), the total energy consumption of memory accesses is comprised of two main elements, specifically the energy dissipated by on-chip cache, which includes dynamic cache access energy and static energy, and energy consumed by off-chip memory accesses.

Figure 5 shows the miss rate of the benchmark *parser* from Spec 2000 [SPEC] considering cache sizes ranging from 1 Kbytes to 1 Mbytes. Figure 6 provides the energy dissipation of on-chip cache, off-chip memory, and total energy dissipation of the benchmark *parser*. When the cache size is increased from 1 to 16 Kbytes, the miss rate dramatically decreases, which results in a decrease in off-chip memory accesses and a decrease in off-chip memory energy consumption. As we further increase the cache size, the energy consumption of off-chip memory decreases very little. However, the energy dissipated by the on-chip cache continues to increase as the cache size increases. Therefore, the increase in on-chip cache energy dissipation will eventually outweigh the decrease in energy of the off-chip memory. For the benchmark *parser*, this turning point is at a cache size of 16 Kbytes at which increasing the cache size will not improve performance greatly but will risk increasing energy dissipation significantly.

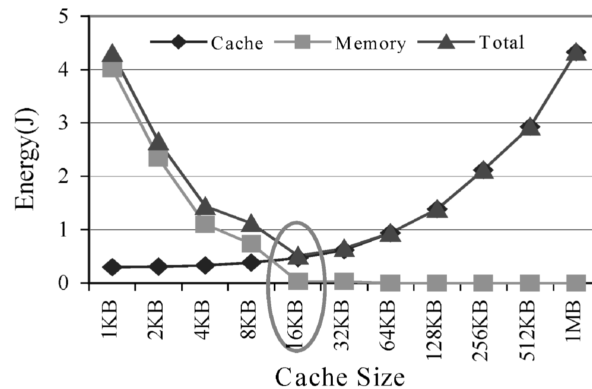

Fig. 6. Energy dissipation on on-chip cache, off chip memory and the total of benchmark *parser* at cache size from 1 Kbytes to 1 Mbytes.

Unfortunately, this tradeoff point is different for every application and exists not only for cache size but also for cache associativity and line size. For many applications, when associativity is increased from 4-way to 8-way, the performance improvement is very limited, but the energy dissipation is increased greatly because more data and tag ways are accessed concurrently. Therefore, in developing our search heuristic to find the best cache configuration, for each possible cache parameter we attempt to iteratively adjust each parameter with the intention of increasing cache performance as long as a decrease in total energy dissipation is observed.

To help us develop the heuristic for efficiently searching the configuration space, we first analyzed each parameter—cache size, associativity, line size, and way prediction—to determine their impacts on miss rate and energy. The parameter with the greatest impact would likely be the best parameter to configure first. We executed 13 of Motorola's Powerstone benchmarks and six MediaBench benchmarks for all 27 possible cache configurations. Although there are three cache parameters each with three possible values, and way prediction as on or off, there are less than $3 \times 3 \times 3 \times 2 = 54$ configurations, because not all configurations are possible—e.g., size is decreased by shutting down ways, so a 4-way 2 Kbytes cache is not possible. A common way to evaluate the impact of several variables is to fix some variables and vary the others. We therefore fix three parameters and vary the fourth one.

Figure 7 shows the average instruction miss rate of all the benchmarks simulated and the average energy consumption of the instruction cache for the examined configurations.

Figure 8 shows the average data miss rate of all the benchmarks simulated and the average energy consumption of the data cache. Total cache sizes are shown as 8, 4, and 2 Kbytes, line sizes as 16, 32, and 64 bytes, and associativity as 1-way, 2-way, and 4-way. The energy dissipation of a set-associative cache with way prediction is not shown, as way prediction does not impact the miss rate.

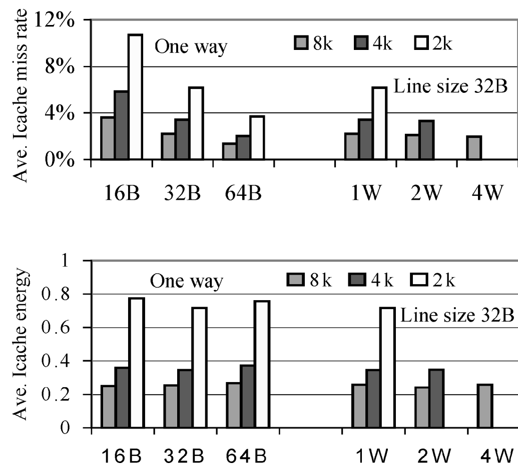

Fig. 7. Average instruction cache miss rate (top) and normalized instruction fetch energy (bottom) of the benchmarks.

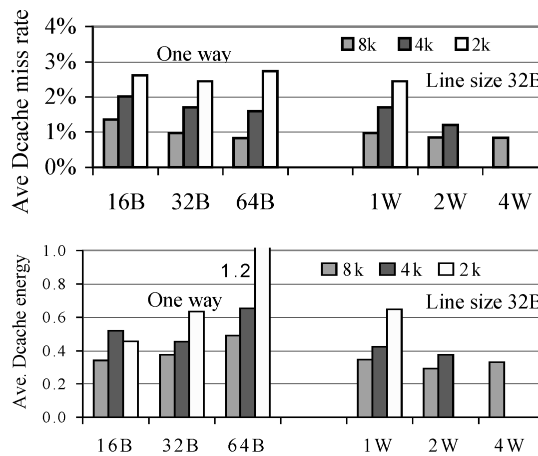

Fig. 8. Average data cache miss rate (top) and normalized data fetch energy (bottom) of the benchmarks.

By looking at the varying bar heights in each group of bars, we see in general that total cache size has the biggest average impact on energy and miss rate—changing cache size can impact energy by a factor of two or more. By looking at the difference in the same colored bar for different line sizes, we notice very little energy variation for different instruction cache line size. However, we do see more variation in data cache energy due to line size, especially for a 2 Kbytes cache. This result is not surprising, since data addresses tend not to have as strong of a spatial locality compared with instruction addresses. Finally, by examining the same colored bars for different associativity, we notice very little change in energy consumption, indicating that associativity has a smaller impact on energy consumption than either cache size or line size. From our analysis, we developed a search heuristic that finds the best cache size first,

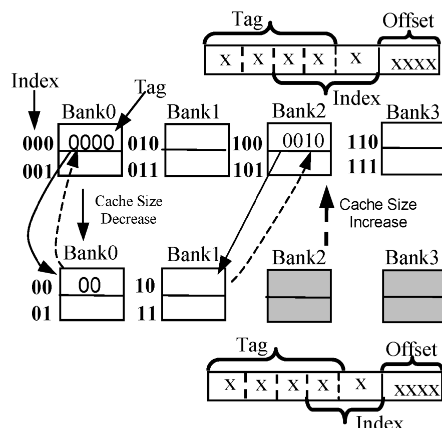

Fig. 9. Cache size configuration: increasing size avoids flushing, while decreasing size does incur flushing. The tag's width is fixed, which is four bits in this example.

determines the best line size, determines the best associativity, and finally if the best associativity is more than one, our heuristic determines whether to use way prediction.

3.3 Minimizing Cache Flushing

In the previous section, we determined a heuristic order in which to vary the parameters. However, the order in which we vary the values of each parameter also matters—one order may require cache flushing and/or incur extra misses, while a different order may not.

For cache size, starting with the smallest cache and increasing the size is preferable over decreasing the size, as illustrated in Figure 9. When decreasing the size, an original hit may turn into a miss after the cache memory bank is shutdown. For example, addresses 00000 (index=000, tag=0000) and 00100 (index=100, tag=0010) are both hits before shutdown but will be mapped to the same block indexed by 00, resulting in a miss. While the width of the tag is fixed, the width of the index changes as the cache configuration changes. For the data cache, we have to write back such items when the data in the shutdown ways is dirty. Such flushing is expensive in terms of power and time.

Alternatively, increasing the cache size does not require flushing. For example, before the cache size is increased, addresses 00100 and 00000 are mapped to the cache block indexed at 00. After the cache size is increased, the address 00100 will be mapped to index 100, possibly incurring an extra miss. However, no write back is necessary for the data cache in this case, and we thus avoid flushing.

For associativity, increasing associativity is preferable over decreasing, as shown in Figure 10. When associativity is increased, there will be no extra misses or errors incurred because more ways are activated to read the data. For example, if addresses 00000 and 00100 are both hits before the increase in associativity, then both addresses will still be hits after the associativity is increased. However, decreasing the associativity may turn a hit into a miss,

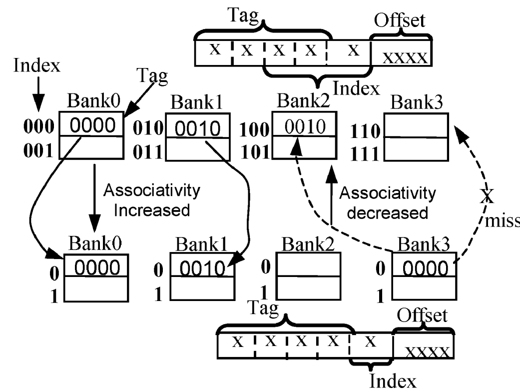

Fig. 10. Cache flushing analysis when associativity is increased or decreased. The tag's width is fixed, which is four bits in this example.

increasing the miss rate. For example, if address 00000 is a hit at bank 3, and the associativity is decreased to a direct mapped cache, the cache will try to locate the address in bank 0, resulting in a miss. In either case, the cache does not need to flush the data and no errors will occur if we design the configurable cache to always check the full tag, instead of reducing the tag to two bits in the direct mapped case. Furthermore, reducing the cache's tag to two bits when configured as a direct mapped cache yields no significant power advantage. Therefore, checking the full tag is reasonable.

In determining the best line size, increasing or decreasing the line size will result in the same behavior, since we use a physical line size of 16 bytes. Therefore, no extra misses will occur and no flushing is needed.

We only use way prediction in a set-associative cache. The accuracy of way prediction depends on each application. Generally, prediction accuracy for a set-associative instruction cache is around 90% and around 70% for a data cache [Powell 2001]. An incorrect prediction will incur extra energy dissipation and an extra cycle to read the data.

3.4 Search Heuristic

Based on the above analyses, we use a heuristic to search for the best cache parameters. The input of the heuristic is:

- Cache size: $C[i], 1 \leq i \leq n, n$ is the number of possible cache size, $n = 3$ in our configurable cache, where $C[1] = 2$ Kbytes, $C[2] = 4$ Kbytes, and $C[3] = 8$ Kbytes;
- Cache associativity: $A[j], 1 \leq j \leq m, m$ is the number of possible cache associativity, $m = 3$ in our configurable cache, where $A[1] = 1$-way, $A[2] = 2$-way, and $A[3] = 4$-way;
- Cache line size: $L[k], 1 \leq k \leq p, p$ is the number of possible cache line size, $p = 3$ in our configurable cache, where $L[1] = 16$ bytes, $L[2] = 32$ bytes, and $L[3] = 64$ bytes; and
- Way prediction: $W[1] =$ off, $W[2] =$ on.

```
Cache Tuning Heuristic Algorithm
    Input: cache size: C[i], cache associativity: A[i], cache line size: L[j], way
           prediction: W[1] = off ,W[2]= on
    Output: the best Cache size C, associativity A, line size L and way prediction
           status
begin: A = A[1], L = L[1], W = W[1], E[0] = 0
    for i =1 to n do
        energy calculation using Equation 1:
            E[i] = f(C[i],A,L,W)
        if E[i]<E[i-1] break
    end
    C_best= C[i],E[1] = E[i]
    for j =2 to p do
        energy calculation using Equation 1:
            E [j] = f (C_best , A, L[j], W)
        if E [j]<E [j-1] break
    end
    L_best= L[j],E[i] = E[j]
    for k = 2 to m do
        energy calculation using Equation 1:
            E[k] = f(C_best, A[k] , L_best , W)
        if E[k]<E[k-1]  break
    end
    A_best= A[k], E[0]=E[k]
    if A_best = 1 then
        W_best  = W[1]
    else
        W = W[2]
        if E[1] = f(C_best, A_best , L_best , W) <E[0] then
                W_best = W[2]
output: C_best, A_best , L_best, W_best.
```

Fig. 11. Search heuristic for determining best cache configuration.

Figure 11 provides pseudocode for our search heuristic, which we use to determine the best cache configuration. Our heuristic starts with a 2 Kbytes direct mapped cache where the line size is 16 bytes. We then gradually increase the total cache size to our largest possible size of 8 Kbytes as long as increasing the size of the cache results in a decrease in total energy. After determining the best cache size, we begin increasing the line size from 16 to 32 bytes and finally 64 bytes. Once again, as we increase the line size of the cache, if we do not notice a decrease in energy consumption, we choose the best line size configuration we have seen so far. Similarly, we then determine the best associativity by gradually increasing the associativity until we see no further improvement in energy consumption. Finally, we determine if enabling way prediction results in any energy savings.

The best cache size, cache line size, associativity, and way prediction have been determined in an efficient manner. While our search heuristic is scalable to larger caches, which have more possible settings for cache size, line size, and

associativity, we have not analyzed the accuracy of our heuristic with larger caches but plan to do so as future work.

3.5 The Efficiency of the Heuristic

We can generalize the efficiency of our search heuristic in the following way. Suppose there are n configurable parameters, each parameter has m values, then there are total m^n different combinations, assuming the m values of the n parameters are independent of each other. However, the heuristic only searches $m \times n$ combinations at most. For example, suppose we have 10 parameters of which each has 10 values. Brute force searching searches 10,000,000,000 configurations, while our heuristic would only search 100 configurations instead.

We can also use the heuristic to search through a multilevel cache memory system. Suppose we have a 16 Kbytes 8-way instruction and data cache with line sizes of 8, 16, 32, and 64 bytes. Suppose there is also a second-level unified L2 cache, which is 256 Kbytes 8-way with line sizes of 64, 128, 256, and 512 bytes. The total solution space size is $40 \times 40 \times 40 = 64{,}000$. However, by using our heuristic, we search $10 + 10 + 10 = 30$ configurations at most.

3.6 Implementing the Heuristic in Hardware

We could implement our cache tuning heuristic in either software or hardware. However, in a software-based approach, the system processor would execute the search heuristic. Executing the heuristic on the system processor would not only change the run-time behavior of the application but also affect the cache behavior, possibly resulting in the search heuristic choosing a nonoptimal cache configuration. Therefore, a hardware-based approach that does not significantly impact the system's area or power consumption is more desirable.

We implemented the search heuristic in hardware using a simple state machine controlling a simple datapath, shown in Figure 12. In the datapath, there are 18 registers. We use three of the registers to collect run-time information, total cache hits, total cache misses, and total cycles. Six additional registers store the cache hit energy per cache access, which correspond to 8 Kbytes 4-way, 2-way, and 1-way; 4 Kbytes 2-way and 1-way; and 2 Kbytes 1-way configurations. The physical line size is 16 bytes, so the cache hit energy for different cache line sizes is the same. We use three registers to store the miss energy, which corresponds to line sizes of 16, 32, and 64 bytes, respectively. Because static power dissipation depends on the cache size only, we use three more registers to store the static power dissipation corresponding to 8, 4, and 2 Kbytes caches, respectively. All 15 registers are 16 bits wide. We also need one register to hold the result of energy calculations and another register to hold the lowest energy of the cache configurations tested. Both of these registers are 32 bits wide. The last register is the configure register that is used to configure the cache. We have four cache parameters to configure, where cache size, line size, and associativity have three possible values, and prediction can either be on or off. Therefore, the configure register is seven bits wide. The FSM controls the datapath using the signal "control" and the output of the comparator within the datapath is an input to the FSM.

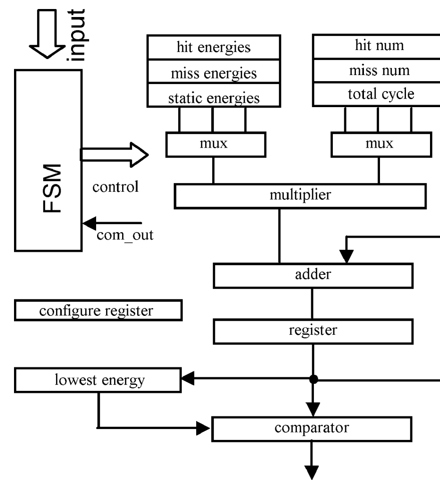

Fig. 12. FSM+D of the cache tuner. The "input" includes clock, reset, and start signal. The "control" is the output of FSM to control the registers and muxes; the output of the comparator is fed back to FSM.

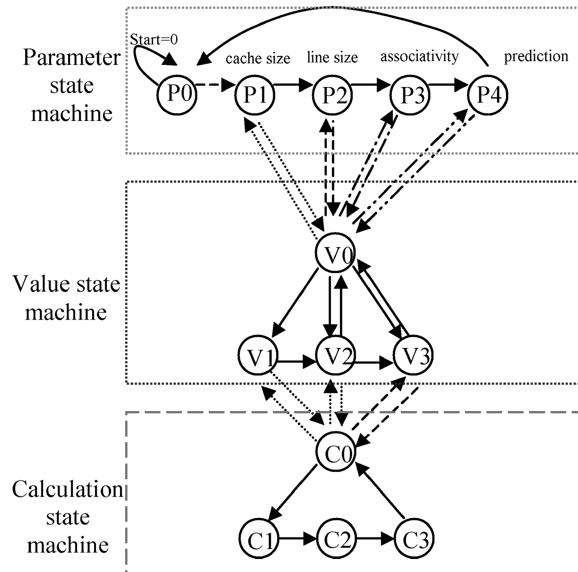

Fig. 13. FSM of the cache tuner.

Figure 13 shows the FSM of the cache tuner composed of three smaller state machines. The first state machine is for cache parameters, which we will refer to as the parameter state machine (PSM). The first state of the PSM is the start state, which has to wait for the start signal to start the cache tuning. The second state, state P1, is for tuning the cache size, where the best cache size is determined in this state. The other states P2, P3, and P4 are for cache line size, cache associativity, and way prediction, respectively. The second state

machine determines the energy dissipation for the many possible values of each cache parameter. We will refer to this state machine as the value state machine (VSM). The highest possible value of these cache parameters is three, so we use four states in the VSM. If the current state of PSM is P1, corresponding to determining the best cache size, the second state of the VSM will determine the energy of 2 Kbytes cache; the third state, V2, is for a 4 Kbytes cache, and V3 is for an 8 Kbytes cache. The first state, V0, is an interface state between PSM and VSM. If the PSM is P2, which is for line size tuning, then the second state of the VSM, V1, is for a line size of 16 bytes, the third state of VSM, V2, is for a line size of 32 bytes, and the last state, V3, is for a line size of 64 bytes. We also need a third state machine to control the calculation of the energy. Because we have three multiplications, and only one multiplier, we use a state machine that has four states to compute the energy. We call this state machine the calculate state machine (CSM). The first state is also an interface state between VSM and CSM.

In Figure 13, the solid lines show state transitions in the three state machines, respectively. The dotted lines show the dependence of upper level state machines on the lower level state machines, for example, the dotted lines between P1 and V0 shows that P1 must wait for VSM to finish before going to the next state, P2.

4. EXPERIMENTS

Table 1 show the results of our search heuristic, for instruction and data cache configurations. Our search heuristic only searches on average 5.8 configurations compared to 27 configurations that an exhaustive approach would analyze. Furthermore, the heuristic finds the optimal configuration in nearly all cases. Additionally, our results demonstrate that way prediction is only beneficial for instruction caches and that only a 4-way set-associative instruction cache had lower energy consumption when way prediction is used. While way prediction is typically beneficial when considering a set-associative cache, for the benchmarks we examined, the cache configurations with the lowest energy dissipation were mostly direct mapped caches where way prediction is not applicable.

For the benchmarks *mpeg2* and *pjpeg*, our heuristic search does not choose the optimal cache configuration, selecting configurations that 5% and 12% worse than the optimal configuration, respectively. The optimal data cache configuration of *mpeg2* is an 8 Kbytes 2-way set-associative cache with a line size of 16 bytes, whereas our heuristic selects a 4 Kbytes 2-way set-associative cache with a line size of 16 bytes. For a direct mapped cache with a line size of 16 bytes, the miss rate of the data cache for *mpeg2* is 0.82% using a 4 Kbytes cache and 0.58% using an 8 Kbytes cache. By increasing the cache size from 4 to 8 Kbytes, we only achieve a reduction in miss rate of 1.4×. However, a larger cache is only preferable if the improved miss rate results in a large enough reduction in energy consumption in the off-chip memory to overcome the increased energy consumption of the larger cache. For *mpeg2*, the reduced miss rate achieved using an 8 Kbytes cache is not large enough to overcome the

Table I. Results of Search Heuristic

Ben.	I-cache cfg.	No.	D-cache cfg.	No.	E%
padpcm	8K_1W_64B	7	8K_1W_32B	7	23%
crc	2K_1W_32B	4	4K_1W_64B	6	70%
auto	8K_2W_16B	7	4K_1W_32B	6	3%
bcnt	2K_1W_32B	4	2K_1W_64B	4	70%
bilv	4K_1W_64B	6	2K_1W_64B	4	64%
binary	2K_1W_32B	4	2K_1W_64B	4	54%
blit	2K_1W_32B	4	8K_2W_32B	8	60%
brev	4K_1W_32B	6	2K_1W_64B	4	63%
g3fax	4K_1W_32B	6	4K_1W_16B	5	60%
fir	4K_1W_32B	6	2K_1W_64B	4	29%
jpeg	8K_4W_32B	8	4K_2W_32B	7	6%
pjepg	4K_1W_32B	6	4K_1W_16B	5	51%
	optimal		*4K_2W_64B*		
ucbqsort	4K_1W_16B	6	4K_1W_64B	6	63%
tv	8K_1W_16B	7	8K_2W_16B	7	37%
adpcm	2K_1W_16B	5	4K_1W_16B	5	64%
epic	2K_1W_64B	5	8K_1W_16B	6	39%
g721	8K_4W_16B_P	8	2K_1W_16B	3	15%
pegwit	4K_1W_16B	5	4K_1W_16B	5	37%
mpeg2	4K_1W_32B	6	4k_2w_16B	6	40%
	optimal		*8K_2W_16B*		
	Average:	5.8	Average:	5.4	45%

Ben. is the benchmark considered, *cfg*. is the cache configuration selected, *No*. is the number of configurations examined by our heuristic, and E% is the energy savings of both I-cache and D-cache.

added energy consumed by the cache itself, and we therefore select a cache size of 4 Kbytes. When cache associativity is considered, the miss rate of the 8 Kbytes cache is significantly reduced when the associativity is increased to a 2-way set-associative cache, which results in a 5× reduction in miss rate. When our heuristic is determining the best cache size, we cannot predict what will happen when associativity is increased. Therefore, our heuristic did not choose the optimal cache configurations in the cases of *mpeg2* and *pjpeg*.

We also tried several different search heuristics to compare with our final choice. One particular search heuristic we analyzed searched in the order of line size, associativity, way prediction, and cache size. This heuristic did not find the optimal configuration in 11 out of 18 examples for the instruction cache and in 7 out of 18 examples for the data cache. For both caches, the suboptimal configurations consumed up to 7% more energy.

We also required that the tuning hardware impose as little of an overhead on area and overall system power. We implemented our cache tuner using VHDL and synthesized the tuner using Synopsys Design Compiler. The total size of our cache tuner is roughly 4,000 gates, or 0.039 mm^2, using a 0.18-μm CMOS technology. Compared to the reported size of the MIPS 4 Kp with caches [MIPS Technologies Inc. 2003], this represents an increase in area of just over a 3%.

From gate-level simulations of the cache tuner, we determined the total number of cycles used to find the best cache configuration is 164. Additionally, the

power consumed by our cache tuner is 2.69 mW executing at 200 MHz. Therefore, our cache tuner, which searches an average of 5.4 configurations to find the best configuration, has a very low energy consumption of 11.9 nJ on average. Compared with the total energy dissipation of the benchmarks, which ranged from 0.16 mJ to 3.03 J with an average of 2.34 J, the energy dissipation of the cache tuner is negligible.

In order to show the impact that data cache flushing would have had, we calculated the energy consumption of the benchmarks when the cache size is configured in the order of 8 down to 2 Kbytes. The average energy consumption due to writing back dirty data is 5.38 mJ. Thus, if we search the possible cache size configurations from largest to smallest, the additional energy dissipation due to cache flushes would be 480,000 times larger than that of our cache tuner.

5. CONCLUSIONS

A configurable cache enables tuning of the cache to a particular program, which can significantly reduce memory access power that often accounts for half a microprocessor system's power. Our self-tuning on-chip CAD method relieves designers from the burden of having to perform simulations or manual physical measurements to determine the best configuration, finding the best configuration automatically. Our heuristic minimizes the number of configurations examined during tuning, and minimizes the need for cache flushing. Energy savings of such a cache average 40% compared to a standard cache. The self-tuning cache can be used in a variety of approaches.

REFERENCES

AGARWAL, A., HOROWITZ, M., AND HENNESSY, J. 1989. An analytical cache model. *ACM Trans. Comput. Syst. 7*, 2, 184–215.

AGARWAL, A., LI, H., AND ROY, K. 2002. DRG-cache: A data retention gated-ground cache for low power. In *Proceedings of 39thDesign Automation Conference*, New York, NY, June 2002. ACM, 473–478.

ALBONESI, D. H. 1999. Selective cache way: On-demand cache resource allocation. In *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, Los Alamitos, CA, USA. IEEE Computer Society, 248–259.

BALASUBRAMONIAN, R., ALBONESI, D., BUYUKTOSUNOGLU, A., AND DWARKADAS, S. 2000. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *Proceedings 33rd Annual IEEE/ACM International Symposium on Microarchitecture*, Piscataway, NJ, USA. IEEE, 245–257.

BURGER, D. AND AUSTIN, T. M. 1997. The SimpleScalar Tool Set, Version 2.0. Technical Report #1342, Department of Computer Sciences, University of Wisconsin-Madison.

GIVARGIS, T. AND VAHID, F. 2002. Platune: A tuning framework for system-on-a-chip platforms. *IEEE Trans. CAD 21*, 11.

INOUE, K., ISHIHARA, T., AND MURAKAMI, K. 1999. Way-predictive set-associative cache for high performance and low energy consumption. In *Proceedings of International Symposium on Low Power Electronic Design*.

KAXIRAS, S., HU, Z., AND MARTONOSI, M. 2001. Cache decay: Exploiting generational behavior to reduce cache leakage power. In *28th Annual International Symposium on Computer Architecture*.

LEE, C., POTKONJAK, M., AND MANGIONE-SMITH, W. 1997. MediaBench: A tool for evaluating and synthesizing multimedia and communications systems. In *International Symposium on Microarchitecture*.

MALIK, A., MOYER, B., AND CERMAK, D. 2000. A low power unified cache architecture providing power and performance flexibility. In *International Symposium on Low Power Electronics and Design*.

MIPS TECHNOLOGIES INC. 2003. http://www.mips.com/products/s2p3.html.

POWELL, M., AGAEWAL, A., VIJAYKUMAR, T., FALSAFI, B., AND ROY, K. 2001. Reducing set-associative cache energy via way-prediction and selective direct mapping. In *34th International Symposium on Microarchitecture*.

REINMAN, G. AND JOUPPI, N. P., 1999. *CACTI2.0: An Integrated Cache Timing and Power Model*. COMPAQ Western Research Lab.

VEIDENBAUM, A., TANG, W., GUPTA, R., NICOLAU, AND JI. X., 1999. Adapting cache line size to application behavior. In *Proceedings of the 1999 International Conference on Supercomputing.* ACM, New York, NY, USA, 145–154.

SEGARS, S. 2001. Low power design techniques for microprocessors. In *IEEE International Solid-State Circuits Conference Tutorial*.

SPEC. 2000. Standard Performance Evaluation Corporation. http://www.specbench.org.

ZHANG, C., VAHID, F., AND NAJJAR, W. 2003a. Energy benefits of a configurable line size cache for embedded systems. In *Proceedings of IEEE Computer Society Annual Symposium on VLSI. New Trends and Technologies for VLSI Systems Design*, Florida, USA. IEEE Computer Society, Los Alamitos, CA, USA, 87–91.

ZHANG, C., VAHID, F., AND NAJJAR, W. 2003b. A highly configurable cache architecture for embedded systems. In *Proceedings of the 30th ACM/IEEE International Symposium on Computer Architecture*, San Diego, CA. 136–146.

Received January 2003; revised June 2003; accepted August 2003