A Self-Tuning Cache Architecture for Embedded Systems

Chuanjun Zhang
Department of Electrical Engineering
University of California, Riverside
czhang@ee.ucr.edu

Frank Vahid* and Roman Lysecky
Department of Computer Science and Engineering
University of California, Riverside
{vahid/rlysecky}@cs.ucr.edu,
(* also with the Center for Embedded Computer Systems, UC Irvine)

Abstract

Memory accesses can 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 pre-fabricated 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 configurable 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. We carefully designed the heuristic to avoid any cache flushing, since flushing is power and performance costly. By simulating numerous Powerstone and MediaBench benchmarks, we show that such a dynamic self-tuning cache can reduce memory-access energy by 45% to 55% on average, and as much as 97%, compared with a four-way set-associative base cache, completely transparently to the programmer.

Keywords

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 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, etc., 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 [6] incorporated a configurable unified set-associative cache whose four ways could be individually shutdown to reduce dynamic power during cache accesses. We have designed a highly configurable cache [13][14] 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). The benefits of optimally tuning a cache is quite significant, resulting in an average of over 40% savings for Powerstone [6] and MediaBench [5] benchmarks, and up to 70% on certain benchmarks.

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 parameters, but such simulation is often cumbersome to setup. 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 runtime behavior where the data changes dynamically. Recently, some design automation aids have evolved to assist the designer in the tuning task [4]. 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 [10] used an adaptive strategy to adjust cache line size dynamically to an application. Albonesi [1] proposed dynamically turning off the cache ways to reduce dynamic energy dissipation. Balasubramonian [2] 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 in [10], the instruction per cycle (IPC) in [1], and miss rate, IPC, and branch frequency in [2], these dynamic strategies increase/decrease or turn on/off the single aspect of the cache that is tunable.

In our work, we tune four cache parameters: cache line size (64, 32 or 16 bytes), cache 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 cache way prediction (on or off). 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 time periods. The choice of approach is orthogonal to the design of the self-tuning architecture itself.

The paper is organized as follows. We briefly describe energy evaluation in Section 2. In Section 3, we describe our self-tuning

1530-1591/04 $20.00 (c) 2004 IEEE

$$E_{total} = E_{dynamic} + E_{static}$$
$$E_{dynamic} = Cache_{total} * E_{hit} + Cache_{Misses} * E_{miss}$$
$$E_{miss} = E_{offchip\_access} + E_{uP\_stall} + E_{cache\_block\_fill}$$
$$E_{static} = Cycles_{total} * E_{static\_per\_cycle}$$

Equation 1: Equations for calculating total energy due to memory accesses.

$$E_{tuner} = P_{tuner} * Time_{total} * NumSearch$$

Equation 2: Equation for calculating energy consumption of the heuristic cache tuner.

strategy involving a search heuristic. We provide the results of our search heuristic in Section 4. We conclude in Section 5.

2. 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. Energy equals power times time. Dynamic energy consumption causes most of the total energy dissipation in micrometer-scale technologies, but static energy dissipation will contribute an increasingly larger portion of total energy dissipation 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. Furthermore, we compute energy dissipation of the cache tuner using Equation 2.

We simulated numerous Powerstone [6] and MediaBench [5] benchmarks using SimpleScalar [3], a cycle-accurate simulator that includes a MIPS-like microprocessor model, to obtain total cache accesses, $Cache_{Total}$, and cache misses, $Cache_{Misses}$, for each benchmark. We obtain the energy of a cache hit, E_{hit}, from our own CMOS 0.18 µm layout of our configurable cache (we found our energy values correspond closely with CACTI [9] values). We obtain 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.

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 1, that can be enabled and disabled by software. Our goal is for the tuning process and the required additional

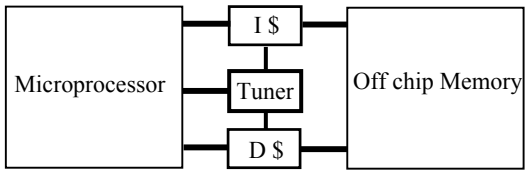

Figure 1: Self-tuning cache architecture.

hardware to be as efficient as possible in terms of the size, power, and performance.

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 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., 100x100x100= 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. Searching the cache configurations in an arbitrary order may require flushing the cache after each configuration to ensure correct cache behavior, which is very time and power costly. Without flushing, the new configuration could have items in the wrong places, yielding 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

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 the energy consumed by off-chip memory accesses. Figure 2 provides the energy dissipation of on-chip cache, off-chip memory, and total memory energy dissipation for the benchmark *parser* from Spec 2000 [12]. When the cache size is increased from 1 Kbyte to 1 Mbyte, the miss rate (not shown due to space limits) dramatically decreases, which

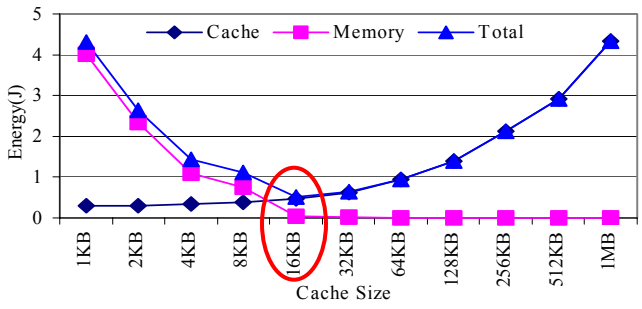

Figure 2: Energy dissipation of on-chip cache, off-chip memory and the total for benchmark *parser* at cache size from 1 Kbyte to 1 Mbyte.

results in a decrease in off-chip memory accesses and a decrease in off-chip memory energy consumption. As shown in Figure 2, while the energy dissipation of off-chip memory decreases rapidly as we increase the cache size from 1 Kbyte to 16 Kbytes, 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 increase energy dissipation significantly. Unfortunately, this tradeoff point is different for every application, and such tradeoffs exist not only for cache size, but also for cache associativity and line size. For example, for many applications, when associativity is increased from 4-way to 8-way, the performance improvement is very limited, but the energy dissipation is greatly increased because more data and tag ways are accessed concurrently. Therefore, in developing our searching heuristic to find the best cache configuration, for each possible cache parameter we attempt to iteratively adjust each parameter, with the goal 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 to configure first. We executed 13 of Motorola's Powerstone benchmarks [6] and 6 MediaBench benchmarks [5] for all 27 possible cache configurations. Although there are 3 cache parameters each with 3 possible values, and way prediction as on or off, there are less than 3*3*3*2=54 configurations, because not all configurations are possible – size is decreased by shutting down ways, so a 4-way 2 Kbyte cache is not possible, for example. 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.

Figure 3 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 4 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 Kbytes, 4 Kbytes, and 2 Kbytes, line sizes as 16 bytes, 32 bytes, and 64 bytes, and associativity as 1-way, 2-way, and 4-way. The energy dissipation for way prediction isn't shown, as way prediction doesn't impact miss rate.

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 shaded bars 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 Kbyte 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 shaded 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, 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.

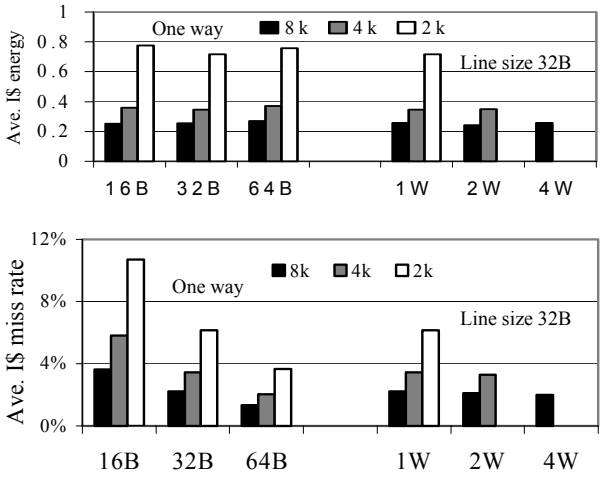

Figure 3:Average instruction cache miss rate (top) and normalized instruction fetch energy (bottom) of the benchmarks.

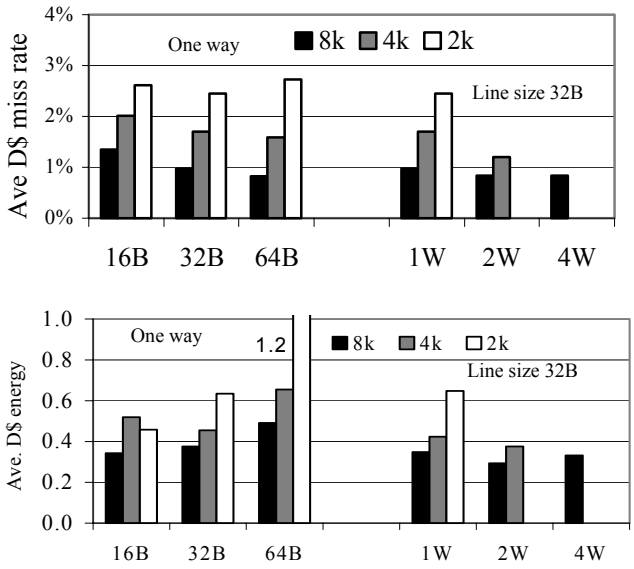

Figure 4: Average data cache miss rate (top) and normalized data fetch energy (bottom) of the benchmarks.

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. We'll illustrate the concept using a trivially small 8 byte memory for simplicity. Figure 5 illustrates an 8 byte memory, a 4 byte cache configured as 1-way and 2-way, and a 2 byte 1-way cache. When decreasing the cache size from 4 bytes to 2 bytes as shown in Figure 5(b) and (c), an original hit may turn into a miss after the cache entries are shutdown. For example, addresses 000 (index=00, tag=00) and 110 (index=10, tag=11) are both hits before shutdown, but will be mapped to the same block indexed by 0, resulting in a miss. While the width of the tag is fixed (in this example the tag is two bits wide), the width of the index changes as the cache's configuration

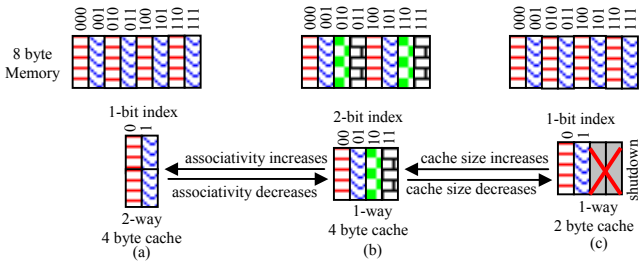

Figure 5: Cache flush analysis when changing cache associativity and cache size. Tag is always two-bits wide. Address space in memory will be mapped to the cache with the same pattern.

does. For 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 100 and 010 are mapped to the cache block indexed at 0. After the cache size is increased, the address 100 will be mapped to index 00, and the address 010 will be mapped to index 10, so there may be an extra miss. However, no write back is necessary in this case, and we thus avoid flushing.

For associativity, increasing associativity is preferable over decreasing, as shown in Figure 5(a) and (b). 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 000 and 010 are both hits before the increase of the associativity, then both addresses will still be hits after the associativity is increased. However, decreasing the associativity may turn a hit into a miss, increasing the miss rate. 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 one bit 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, and 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 [8]. 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 inputs to the heuristic are:

- Cache size: $C[i]$, $1 \le i \le n$, n is the number of possible cache size; $n=3$ in our configurable cache, where $C[1]$ = 2 Kbyte, $C[2]$ = 4 Kbyte, and $C[3]$ = 8 Kbyte;
- Cache associativity: $A[j]$, $1 \le j \le m$, m is the number of possible cache associativities; m=3 in our configurable cache, where $A[1]$ = 1 way, $A[2]$ = 2 ways, and $A[3]$ = 4 ways;
- Cache line size: $L[k]$, $1 \le k \le p$, p is the number of possible cache line sizes; $p=3$ in our configurable cache, where $L[1]$ = 16 bytes, $L[2]$ = 32 bytes, and $L[3]$ = 64 bytes; and

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}.

Figure 6: Search heuristic for determining the best cache configuration.

- Way prediction: $W[1]$ = off, $W[2]$ = on.

Figure 6 provides psuedocode for our search heuristic, which we use to determine the best cache configuration. Our heuristic starts with a 2 Kbyte 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 bytes 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.

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.

We can describe the efficiency of our search heuristic as follows. Suppose there are n configurable parameters, and each parameter has m values, then there are a total of m^n different combinations, assuming the m values of the n parameters are independent. However, our heuristic only searches $m*n$ combinations at most. Suppose we have 10 parameters of which each has 10 values. Brute force searching searches 10,000,000,000 combinations, while the heuristic searches 100 instead.

We can also use the heuristic to search through a multilevel cache memory system. Suppose we have 16 Kbyte 8-way instruction and data caches with line sizes of 8, 16, 32, and 64 bytes. Suppose there is also a second level unified L2 cache, which is a 256 Kbyte 8-way cache with a line size of 64, 128, 256, and

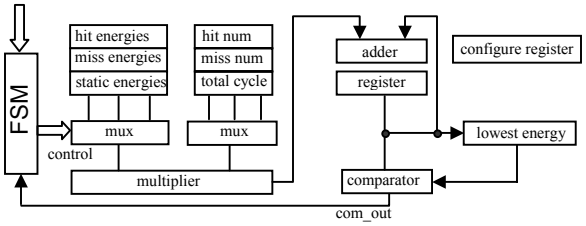

Figure 7**:** FSMD of the cache tuner

512 bytes. The total solution space is 40*40*40 = 64000. However, by using our heuristic, we search 10+10+10 = 30 combinations at most.

3.5 Implementing the Heuristic in Hardware

We could implement our cache tuning heuristic in either software or hardware. 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 runtime behavior of the application but also affect the cache behavior, possibly resulting in the search heuristic choosing a non-optimal cache configuration. Therefore, we prefer a hardware-based approach that does not significantly impact overall area or power.

Implementing the search heuristic in hardware is achieved using a simple state machine controlling a simple datapath, shown in Figure 7. In the datapath, there are eighteen registers. We use three of the registers to collect runtime information, the total number of cache hits and misses, and the 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 bytes, 32 bytes, 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 Kbyte, 4 Kbyte, and 2 Kbyte caches, respectively. All fifteen 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 configuration 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.

Figure 8 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

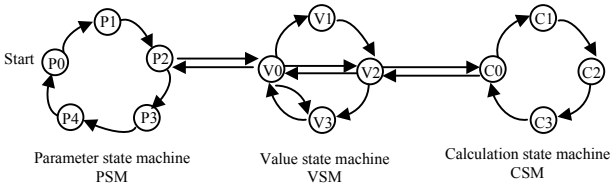

Figure 8: FSM of the cache tuner.

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 a 2 Kbyte cache; the third state, V2, is for a 4 Kbyte cache, and V3 is for an 8 Kbyte 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 the 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 8, the solid lines show state transitions in the three state machines, respectively. The PSM states, P1, P2, P3, and P4 depend on VSM, although only P2 to V0 is drawn. In the same way, VSM states, V1, V2, and V3 depend on CSM, but only dependence of V2 on C0 is drawn.

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, *and involves no cache flushing*. Furthermore, the heuristic finds the optimal configuration in nearly all cases. For the two data cache configurations where the heuristic doesn't find the optimal, *pjpeg* and *mpeg2*, the configuration found is only 5% and 12% worse than the optimal, respectively. Additionally, our results demonstrate that way prediction is only beneficial for instruction caches and that only a 4-way set associative instruction cache has lower energy consumption when way prediction is used. In general, way prediction is beneficial when considering a set associative cache. However, 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. The optimal data cache configuration of *mpeg2* is an 8 Kbyte 2-way set associative cache with a line size of 16 bytes. However, the heuristic selects a 4 Kbyte 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 3.29% using a 2 Kbyte cache, 0.82% using a 4 Kbyte cache, and 0.58% using a 8 Kbyte cache. By increasing the cache size from 2 Kbytes to 4 Kbytes, we achieve a 4X miss rate reduction. By increasing the cache size further to 8 Kbytes, we only achieve a further reduction in miss rate of 1.4X. Larger caches consume more dynamic and static energy. Hence, a larger cache is only preferable if the improved miss rate results in 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 Kbyte cache is not large enough to overcome the 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 Kbyte cache is significantly reduced when the associativity is increased from direct mapped to 2 way set associative, which results in a 5X reduction in miss rate.

Table 1: Results of search heuristic. *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

Ben.	I-cache cfg.	No.	D-cache cfg.	No.	I-E%	D-E%
padpcm	8K_1W_64B	7	8K_1W_32B	7	23%	77%
crc	2K_1W_32B	4	4K_1W_64B	6	70%	30%
auto	8K_2W_16B	7	4K_1W_32B	6	3%	97%
bcnt	2K_1W_32B	4	2K_1W_64B	4	70%	30%
bilv	4K_1W_64B	6	2K_1W_64B	4	64%	36%
binary	2K_1W_32B	4	2K_1W_64B	4	54%	46%
blit	2K_1W_32B	4	8K_2W_32B	8	60%	40%
brev	4K_1W_32B	6	2K_1W_64B	4	63%	37%
g3fax	4K_1W_32B	6	4K_1W_16B	5	60%	40%
fir	4K_1W_32B	6	2K_1W_64B	4	29%	71%
jpeg	8K_4W_32B	8	4K_2W_32B	7	6%	94%
pjepg	4K_1W_32B	6	4K_1W_16B	*5*	51%	49%
	optimal		*4K_2W_64B*			
ucbqsort	4K_1W_16B	6	4K_1W_64B	6	63%	37%
tv	8K_1W_16B	7	8K_2W_16B	7	37%	*63%*
adpcm	2K_1W_16B	5	4K_1W_16B	5	64%	36%
epic	2K_1W_64B	5	8K_1W_16B	6	39%	61%
g721	8K_4W_16B_P	8	2K_1W_16B	3	15%	85%
pegwit	4K_1W_16B	5	4K_1W_16B	5	37%	63%
mpeg2	4K_1W_32B	6	4k_2w_16B	*6*	40%	60%
	optimal		*8K_2W_16B*			
	Average:	5.8	Average:	5.4	45%	55%

When our heuristic is determining the best cache size, the heuristic does not predict what will happen when associativity is increased. Therefore, the heuristic did not choose the optimal cache configurations in the cases of *mpeg2* and *pjpeg*.

We also developed and compared several other search heuristics. One particular search heuristic 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 sub-optimal configurations consumed up to 7% more energy.

We sought to develop tuning hardware that imposes little area and power overhead. The cache tuner consists of a few registers, a small custom circuit implementing the state machine (synthesized to hardware), an arithmetic unit capable of performing addition, a slow multiplier (fast multipliers are not necessary since the equations are only occasionally computed), a small control circuit that uses the arithmetic unit to compute energy, and a comparator. We designed the cache tuner hardware using VHDL and synthesized the tuner using Synopsys Design Compiler. The total tuner size was about 4,000 gates, or 0.039 mm^2 in 0.18 μm CMOS technology. Compared to the reported size of the MIPS 4Kp with caches [7], this represents an increase in area of just over 3%. The power consumption of the cache tuner is 2.69 mW at 200 MHz, which is only 0.5% of the power consumed by a MIPS processor. Furthermore, we only use the tuning hardware during the tuning stage; the hardware can be shutdown after the best configuration is determined.

From gate level simulations of the cache tuner, we determined the total number of cycles used to finish one cache configuration is 164 cycles. Executing at 200 MHz, where the average number of cache configurations searched is 5.4, the average energy consumption of the cache tuner is only 11.9 nJ. 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 (recall we avoided flushing by careful ordering of the search), we computed the energy consumption of the benchmarks when cache size is configured in the order of 8 Kbytes down to 2 Kbytes. The average energy consumption due to writing back dirty data ranges from 9.48 μJ to 21 mJ with an average 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 findings the best configuration automatically, thus relieving designers from the burden of having to perform simulations or manual physical measurements to determine the best configuration. 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 tuning approches. Moving traditional desktop CAD algorithms to on-chip hardware will likely become increasingly common as chip transistor capacities continue to increase.

6. Acknowledgements

This work was supported by the National Science Foundation (CCR-0203829, CCR-9876006) and by the Semiconductor Research Corporation (CSR 2002-RJ-1046G).

7. References

[1] D.H. Albonesi, "Selective Cache Way: On-Demand Cache Resource Allocation," Journal of Instruction Level Parallelism, 2000.

[2] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas, "Memory Hierarchy Reconfiguration for Energy and Performance in General-Purpose Processor Architectures," 33rd International Symposium on Microarchitecture, 2000

[3] D. Burger and T.M. Austin, "The SimpleScalar Tool Set, Version 2.0," Univ. of Wisconsin-Madison Computer Sciences Dept, Technical Report #1342, 1997.

[4] T. Givargis and F. Vahid, "Platune: A Tuning Framework for System-on-a-Chip Platforms," IEEE Transaction. on CAD, Vol. 21, No. 11, 2002.

[5] C. Lee, M. Potkonjak and W. Mangione-Smith, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems," International Symposium on Microarchitecture, 1997.

[6] A. Malik, B. Moyer, and D. Cermak, "A Low Power Unified Cache Architecture Providing Power and Performance Flexibility," International Symposium on Low Power Electronics and Design, 2000.

[7] http://www.mips.com/products/s2p3.html, 2003.

[8] M. Powell, A.Agaewal, T. Vijaykumar, Babak Falsafi, and K. Roy, "Reducing Set-Associative Cache Energy via Way-Prediction and Selective Direct Mapping," 34th International Symposium on Microarchitecture, 2001.

[9] G. Reinmann and N.P. Jouppi, "CACTI2.0: An Integrated Cache Timing and Power Model," COMPAQ Western Research Lab, 1999.

[10] A. Veidenbaum, W. Tang, R. Gupta, A. Nicolau, and X. Ji, "Adapting cache line size to application behavior," International Conference on Supercomputing, 1999.

[11] S. Segars, "Low Power Design Techniques for Microprocessors," IEEE International Solid-State Circuits Conference Tutorial, 2001.

[12] http://www.specbench.org

[13] C. Zhang, F. Vahid, W. Najjar. Energy Benefits of a Configurable Line Size Cache for Embedded Systems. International Symposium on VLSI Design, 2003.

[14] C. Zhang, F. Vahid, W. Najjar. A Highly Configurable Cache Architecture for Embedded Systems. In the 30th ACM/IEEE International Symposium on Computer Architecture, 2003.