Chapter 6

TUNING CACHES TO APPLICATIONS FOR LOW-ENERGY EMBEDDED SYSTEMS

Ann Gordon-Ross[1], Chuanjun Zhang[2], Frank Vahid[1,3], and Nikil Dutt[4]
[1]Department of Computer Science and Engineering, University of California, Riverside; [2]Department of Electrical Engineering, University of California, Riverside; [3]Also with the Center for Embedded Computer Systems at UC Irvine; [4]Center for Embedded Computer Systems, School of Information and Computer Science, University of California, Irvine.

Abstract: The power consumed by the memory hierarchy of a microprocessor can contribute to as much as 50% of the total microprocessor system power, and is thus a good candidate for power and energy optimizations. We discuss four methods for tuning a microprocessors' cache subsystem to the needs of any executing application for low-energy embedded systems. We introduce on-chip hardware implementing an efficient cache tuning heuristic that can automatically, transparently, and dynamically tune a configurable level-one cache's total size, associativity and line size to an executing application. We extend the single-level cache tuning heuristic for a two-level cache using a methodology applicable to both a simulation-based exploration environment and a hardware-based system prototyping environment. We show that a victim buffer can be very effective as a configurable parameter in a memory hierarchy. We reduce static energy dissipation of on-chip data cache by compressing the frequent values that widely exist in a data cache memory.

Key words: Cache; configurable; architecture tuning; low power; low energy; embedded systems; on-chip CAD; dynamic optimization; cache hierarchy; cache exploration; cache optimization; victim buffer; frequent value.

1. INTRODUCTION

The power consumed by the memory hierarchy of a microprocessor can contribute to 50% or more of total microprocessor system power[1]. Such a large contributor to power is a good candidate for power and energy optimization. The design of the caches in a memory hierarchy plays a major role in the memory hierarchy's power and performance.

Tuning cache design parameters to the needs of a particular application or program region can save energy. Cache design parameters include: cache size, meaning the total number of data byte storage; cache associativity, meaning the number of tag and data ways simultaneously read per cache access; cache line size, meaning the number of bytes in a block when moving data between cache and the next memory level; and victim buffer use, meaning a small fully-associative buffer storing recently-evicted cache data lines. Every application has different cache requirements that cannot be efficiently satisfied with one predetermined cache configuration. For instance, different applications have vastly different spatial and temporal locality and thus have different requirements[2] with respect to cache size, cache line size, cache associativity, victim buffer configuration, etc. In addition to tunable cache parameters, widely existing frequent values in data caches for some applications can enable data encoding within the cache for reduced power consumption. We define ***cache tuning*** as the task of choosing the best configuration of cache design parameters for a particular application, or for a particular phase of an application, such that performance, power and/or energy are optimized.

New technologies enable cache tuning. Core-based processors allow a designer to choose a particular cache configuration[3-7]. Some processor designs allow caches to be configured during system reset or even during runtime[2,8,9].

Manual tuning of the cache is hard. A single-level cache may have many tens of different cache configurations, and interdependent multi-level caches may have thousands of cache configurations. The configuration space gets even larger if other dependent configurable architecture parameters are considered, such as bus and processor parameters. Exhaustively searching the space may be too slow even if fully automated. With possible average energy savings of over 40% through tuning[2,10], we sought to develop automated cache tuning methods.

In this chapter, we discuss four methods of cache tuning for energy savings. We discuss an in-system method for automatically, transparently, and dynamically tuning a level-one cache; an automatic tuning methodology for two-level caches applicable to both a simulation-based exploration environment or a hardware-based prototyping environment; a configurable victim buffer; and a data cache that encodes frequent data values.

2. BACKGROUND – TUNABLE CACHE PARAMETERS

Many methods exist for configuring a single level of cache to a particular application during design time and in-system during runtime. Cache configuration can be specified during design time for many commercial soft cores from MIPS[6], ARM[5], and Arc[4] and for environments such as Tensilica's Xtensa processor generator[7] and Altera's Nios embedded processor system[3].

Configurable cache hardware also exists to assist in cache configuration. Motorola's M*CORE[9] processors offer way configuration which allows the ways of a unified data/instruction cache to individually be specified as either data or instruction ways. Additionally, ways may be shut down entirely. Way shut-down is further explored by Albonesi[8] to reduce dynamic power by an average of 40%. An adaptive cache line size methodology is proposed by Veidenbaum et al.[11] to reduce memory traffic by more than 50%.

Exhaustive search methods may be used to find optimal cache configurations, but the time required for an exhaustive search is often prohibitive. Several tools do exist for assisting designers in tuning a single level of cache. Platune[12] is a framework for tuning configurable system-on-a-chip (SOC) platforms. Platune offers many configurable parameters beyond just cache parameters, and prunes the search space by isolating interdependent parameters from independent parameters. The level one cache parameters, being dependent, are explored exhaustively.

Heuristic methods exist to prune the search space of the configurable cache. Palesi et al.[13] improves upon the exhaustive search used in Platune by using a genetic algorithm to produce comparable results in less time. Zhang et al.[14] presents a cache configuration exploration methodology wherein a cache exploration component searches configurations in order of their impact on energy, and produces a list of Pareto-optimal points representing reasonable tradeoffs in energy and performance. Ghosh et al.[15] uses an analytical model to efficiently explore cache size and associativity and directly computes a cache configuration to meet the designers' performance constraints.

Few methods exist for tuning multiple levels of a cache hierarchy. Balasubramonian et al.[10] proposes a hardware-based cache configuration management algorithm to improve memory hierarchy performance while considering energy consumption. An average reduction in memory hierarchy energy of 43% can be achieved with a configurable level two and level three cache hierarchy coupled with a conventional level one cache.

3. A SELF-TUNING LEVEL ONE CACHE ARCHITECTURE

Tuning a cache to a particular application can be a cumbersome task left for designers even with the advent of recent computer-aided design (CAD) tuning aids. Large configuration spaces may take a designer weeks or months to explore and with a small time-to-market, lengthy tuning iterations may not be feasible. We propose to move the CAD environment on-chip, eliminating designer effort for cache tuning. We introduce on-chip hardware implementing an efficient heuristic that automatically, transparently, and dynamically tunes the cache to the executing program to reduce energy[16].

3.1 Configurable Cache Architecture

The on-chip hardware tunes four cache parameters in the level-one cache: cache line size (64, 32, or 16 bytes), cache size (8, 4, or 2 Kbytes), associativity (4, 2, or 1-way), and cache way prediction (on or off). Way prediction is a method for reducing set-associative cache energy, in which one way is initially accessed, and other ways accessed only upon a miss.

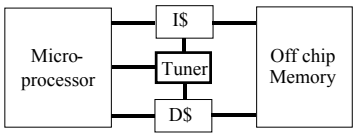

Figure 6-1. Self-tuning cache architecture

The exploration space is quite large, necessitating an efficient exploration heuristic implemented with specialized *tuning* hardware, as illustrated in Figure 6-1. The tuning phase may be activated during a special software-selected tuning mode, during startup of a task, whenever a program phase change is detected, or at fixed time intervals. The choice of approach is orthogonal to the design of the self-tuning architecture itself.

The cache architecture supports a certain range of configurations[2]. The base level-one cache of 8 Kbytes consists of four banks that can operate as four ways. A special configuration register allows the ways to be concatenated to form either a direct-mapped or 2-way set associative 8 Kbyte cache. The configuration register may also be configured to shut down ways, resulting in a 4 Kbyte direct-mapped or 2-way set associative cache or a 2 Kbyte direct-mapped cache. Specifically, due to the bank layout for way shut down, 2 Kbyte 2- or 4-way set associative and 4 Kbyte 4-way

set associative caches are not possible using the configurable cache hardware.

3.2 Heuristic Development Through Analysis

A naïve tuning approach would simply try all possible combinations of configurable parameters in an arbitrary order. For each configuration, the miss rate can be measured and used to estimate the energy consumption of the particular cache configuration. After all configurations are executed, the approach would simply choose the configuration with the lowest energy consumption. However, such an exhaustive method may involve the inspection of too many configurations. Therefore, we wish to develop a cache tuning heuristic that minimizes the number of configurations explored.

When developing a good heuristic, the parameter (cache size, line size, associativity, or way prediction) with the largest impact in performance and energy would likely be the best parameter to search first. We analyzed each parameter to determine the parameter's impact on miss rate and energy by fixing three parameters and varying the third.

We observed that varying the cache size had the largest average impact on energy and miss rate – changing the cache size can impact the energy by a factor of two or more. From our analysis, we developed a search heuristic that first determines the best cache size, determines the best line size, then the best associativity, and finally, if the best associativity is greater than one, our heuristic determines whether to use way prediction or not.

3.3 Search Heuristic

The heuristic developed based on the importance of parameters is summarized below:

1. Begin with a 2 Kbyte, direct-mapped cache with a 16 byte line size. Increase the cache size to 4 Kbytes. If the increase in cache size causes a decrease in energy consumption, increase the cache size to 8 Kbytes. Choose the cache size with the best energy consumption.
2. For the best cache size determined in step 1, increase the line size from 16 bytes to 32 bytes. If the increase in line size causes a decrease in energy consumption, increase the line size to 64 bytes. Choose the line size with the best energy consumption.
3. For the best cache size determined in step 1 and the best line size determined in step 2, increase the associativity to 2 ways. If the increase in associativity causes a decrease in energy consumption, increase the

associativity to 4 ways. Choose the associativity with the best energy consumption.

4. If step (3) determined the best associativity to be greater than 1, determine if enabling way prediction results in energy savings.

The cache tuning heuristic can be implemented 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.

3.4 Experiments and Results

We simulated numerous Powerstone[9] and MediaBench[18] benchmarks using SimpleScalar[19], a cycle-accurate simulator that includes a MIPS-like microprocessor model, to obtain the number of cache accesses and cache misses for each benchmark and configuration explored.

For power dissipation, we considered both 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. We obtain the energy of a cache hit from our own CMOS 0.18 μm layout of our configurable cache (we found our energy values correspond closely with CACTI values). We obtain the off-chip memory access energy from a standard Samsung memory, and the stall energy from a 0.18 μm MIPS microprocessor. Furthermore, we obtained the power consumed by our cache tuner, through simulation of a synthesized version of our cache tuner written in VHDL.

Table 6-1. 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 the I-cache and D-cache.

Ben.	I-cache cfg	No.	D-cache cfg	No.	I-cache E%	D-cache 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%

Ben.	I-cache cfg	No.	D-cache cfg	No.	I-cache E%	D-cache E%
jpeg	8K_4W_32B	8	4K_2W_32B	7	6%	94%
pjpeg	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	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%

Table 6-1 shows the results of our search heuristic, for instruction and data cache configurations. Our search heuristic is quite effective: it searches on average only 5.8 configurations, compared to 27 configurations for an exhaustive approach. Furthermore, our heuristic finds the optimal configuration in nearly all cases. For the two data cache configurations where the heuristic does not find the optimal, *pjpeg* and *mpeg2*, the configuration found is only 5% and 12% worse than the optimal, respectively. On average, the dynamic self-tuning cache can reduce memory-access energy by 45% to 55%. Additionally, be observed 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. 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.

To determine the area and power overhead of our cache tuner, 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[20], 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 tuner can be shutdown after the best configuration is determined, thereby minimizing the effects of additional static power dissipation due to the tuner.

4. AUTOMATIC TUNING OF A TWO-LEVEL CACHE ARCHITECTURE – THE TCAT

In the previous section, we described an automatic method for tuning a single level of cache in system during run-time. We extend the single level cache tuner to tune two-level caches to embedded applications for reduced energy consumption[21]. This method is applicable to both a simulation-based exploration environment and a hardware-based prototyping environment. We present the *two-level cache tuner,* or *TCaT* – a heuristic for searching the huge solution space of possible configurations. The heuristic interlaces the exploration of the two cache levels and searches the various cache parameters in a specific order based on their impact on energy.

4.1 Configurable Cache Architecture

The configurable caches in each of the two cache levels explored here are based on the configurable cache architecture described for a single level configurable cache in Section 3.1. The target architecture for our two-level cache tuning heuristic contains separate level one instruction and data caches and separate level two instruction and data caches. For the first level cache, we explore the same search space as the single level cache tuner: cache line size (64, 32, or 16 bytes), cache size (8, 4, or 2 Kbytes), and associativity (4, 2, or 1-way). For the second level of cache, we expand the cache size to a possible 64, 32, or 16 Kbytes while the line size and associativity parameters are the same. We do not explore way prediction with the TCaT.

An exhaustive exploration of all cache configurations for a two level cache hierarchy is too costly. For a single level separate instruction and data cache design, an exhaustive exploration would explore a total of 28 different cache configurations. However, the addition of a second level of hierarchy raises the number of cache configurations to 432.

Nevertheless, for comparison purposes, we determined the optimal cache configuration for each benchmark by generating exhaustive data. It took over one month of continual simulation time on an UltraSparc compute server to generate the data for our nine benchmarks.

In addition, we have chosen a **base cache** hierarchy configuration consisting of an 8 Kbyte, 4-way set associative level-one cache with a 32 byte line size, and a 64 Kbyte 4-way set associative level two cache with a 64 byte line size – a reasonably common configuration.

4.2 Initial Two-Level Cache Tuning Heuristic – Search Each Level Independently

Initially, we extended the heuristic described in Section 3.3 for a two-level cache by tuning the level-one cache while holding the level-two cache at the smallest size, then tuning the level-two cache using the same heuristic.

We applied the initial heuristic to the benchmarks and found that this heuristic did not perform well for two levels (the original heuristic was intended for only one level, where it works well). The cache configuration determined by our initial heuristic consumed, on average over all benchmarks, 1.41 times more energy than the optimal configuration. In the worst case, our initial heuristic found a cache configuration using 2.7 times more energy than the optimal configuration. In one benchmark, the initial heuristic found a cache configuration that was *worse* than the base cache.

The naïve assumption that the two levels of cache could be configured independently was the reason that our initial heuristic did not perform well for a two level system. In a two-level cache hierarchy, the behavior of each cache level directly affects the behavior of the other level. For example, the miss rate of the level one cache does not solely determine the performance of the level two cache. The performance of the level two cache is also determined by what *values* are missing in the level one cache. To fully explore the dependencies between the two levels, we decided to explore both levels simultaneously.

4.3 The Two-Level Cache Tuner - TCaT

To more fully explore the dependencies between the two cache levels, we expanded our initial heuristic to interlace the exploration of the level one and level two caches. Instead of entirely configuring the level one cache before configuring the level two cache, the interlaced heuristic explores one parameter for both levels of cache before exploring the next parameter, while adhering to the parameter ordering of the initial heuristic. The basic intuition behind our heuristic is that interlacing the exploration allows for better modeling and tuning of the interdependencies between the different levels of cache hierarchy. We applied the interlaced heuristic to the benchmarks and found that the interlaced heuristic performed much better than the initial heuristic, but there was still much room for improvement.

We examined the cases where the interlaced heuristic did not yield the optimal solution. We discovered that in these cases, the optimal was not being reached for two reasons. First, the initial heuristic did not fully explore each parameter. For instance, if an increase from a 2 Kbyte to 4 Kbyte cache size did not yield an improvement in energy, an 8 Kbyte cache size was not

examined. The second reason the optimal configuration was not being found was not due to a failure in the heuristic, but rather due to the limitations set on certain cache configurations by the configurable cache itself. For example, in the level two cache, if a 16 Kbyte cache is chosen as the best size, the only associativity available is a direct-mapped cache. With no energy improvement by increasing the cache from a 16 Kbyte direct-mapped to a 32 Kbyte direct-mapped cache, no other associativities are searched by the previous heuristics. To allow for all associativities to be searched, we added a final adjustment to the associativity search step of the interlaced heuristic with full parameter exploration. The final adjustment allows the cache size to be increased for both the level one and level two caches in order to search larger associativities. We refer to this final heuristic as the two-level cache tuner - the TCaT.

4.4 Experiments and Results

The experimental setup and energy calculations are the same as those described in Section 3.4. We explored nine different benchmarks obtained from MediaBench[18] and EEMBC[22] benchmarks suites.

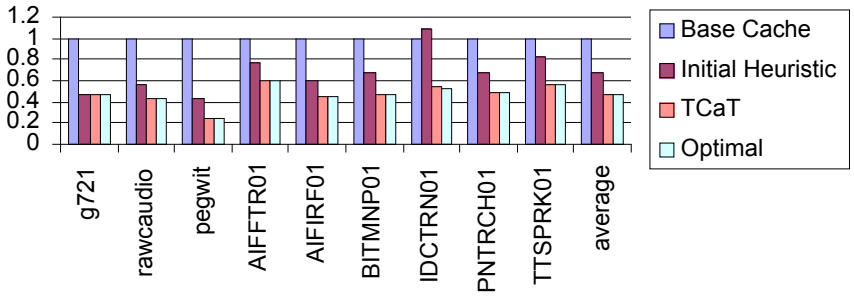

Figure 6-2. Energy consumption for the initial heuristic cache configuration, the TCaT cache configuration, and the optimal cache configuration, normalized to the base cache configuration for each benchmark.

Figure 6-2 shows the results for the initial heuristic and the TCaT for each benchmark. The energy consumptions have been normalized to the base cache configuration for each benchmark's cache hierarchy. The results show that the TCaT finds the optimal cache configuration in most cases. Compared to the base cache configuration and averaged over all benchmarks, the initial heuristic achieves an average energy savings of 32% while the TCaT achieves an average energy savings of 53%. Additionally, we found that for every benchmark, there is no loss of performance due to

cache configuration for optimal energy consumption. In fact, the benchmarks receive an average of a 28% speedup, which we found was due to the tuning of the cache line size.

Furthermore, the TCaT reduces the configuration search space significantly. The exhaustive approach for separate instruction and data caches for a two level cache hierarchy explores 432 cache configurations. The improved heuristic explores only 28 cache configurations, or only 6.5% of the search space. This reduction in the search space speeds up both a simulation approach and a hardware-based prototyping platform approach.

5. USING A VICTIM BUFFER IN AN APPLICATION SPECIFIC MEMORY HEIRARCHY

In addition to tuning cache parameters such as cache size, line size, and associativity, the cache subsystem can include a configurable victim buffer which can be beneficial in systems with a direct-mapped cache. Direct-mapped caches are popular in embedded microprocessor architecture due to their simplicity and good hit rates for many applications. A victim buffer is a small fully-associative cache, whose size is typically 4 to 16 cache lines, residing between a direct-mapped L1 cache and the next level of memory. The victim buffer holds lines discarded after an L1 cache miss. The victim buffer is checked whenever there is an L1 cache miss, before going to the next level memory. If the desired data is found in the victim buffer, the data in the victim buffer is swapped back to the L1 cache. Jouppi[23] reported that a four-entry victim buffer could reduce 20% to 95% of the conflict misses in a 4 Kbyte direct-mapped data cache. Albera and Bahar[24] evaluated the power and performance advantages of a victim buffer in a high performance superscalar, speculative, out-of-order processor. They showed that adding a victim buffer to an 8 Kbyte direct-mapped data cache results in 10% energy savings and 3.5% performance improvements on average for the Spec95 benchmark suite.

A victim buffer improves the performance and energy of a direct-mapped cache *on average*, but for some applications, a victim buffer actually degrades performance without much or any energy savings, as we will show later. Such degradation occurs when the victim buffer hit rate is low. Checking a victim buffer requires an extra cycle after an L1 miss. If the victim buffer hit rate is high, that extra cycle actually prevents dozens of cycles for accessing the next level memory. But if the buffer hit rate is low, that extra cycle does not save much and thus is wasteful. Whether a victim buffer's hit rate is high or low is dependent on what application is running.

Such performance overhead may be one reason that victim buffers are not always included in embedded processor cache architectures.

In this section, we will show that treating the victim buffer as a configurable memory parameter to a direct-mapped cache is superior to either using a direct-mapped cache without a victim buffer or using a direct-mapped cache with an always-on victim buffer[25]. Furthermore, we show that a victim buffer parameter is even useful with a cache that itself is highly parameterized.

5.1 Victim Buffer as a Cache Parameter

We consider adding a victim buffer to both core-based and pre-fabricated platform based design situations.

A core-based approach involves incorporating a processor (core) into a chip before the chip has been fabricated, either using a synthesizable core (soft core) or a layout (hard core). In either case, most core vendors allow a designer to configure the level 1 cache's total size (typical sizes range from no cache to 64 Kbyte), associativity (ranging from direct mapped to 4 or 8 ways), and sometimes line size (ranging from 16 bytes to 64 bytes). Other parameters include use of write through, write back, and write allocate policies for writing to a cache, as well as the size of a write buffer. Adding a victim buffer to a core-based approach is straightforward, involving simply including or not including a buffer into the design.

A pre-fabricated platform is a chip that has already been designed, but is intended for use in a variety of possible applications. To perform efficiently for the largest variety of applications, recent platforms come with parameterized architectures that a designer can configure for his/her particular set of applications. Recent architectures include cache parameters[2,8,9] that can be configured by setting a few configuration register bits. We therefore developed a configurable victim buffer that could be turned on or off by setting bits in a configuration register.

5.2 Experiments and Results

The experimental setup and energy calculations are the same as those described in Section 3.4. The benchmarks examined include programs from the Powerstone[9], MediaBench[18], and Spec2000[26] benchmark suites.

5.2.1 Victim Buffer with a Direct-Mapped Cache

Figure 6-3 shows the performance and energy improvements when adding an always-on victim buffer to a direct-mapped cache. Performance is

the program execution time. Energy is estimated as described in section 3.4. 0% represents the performance and energy consumption of an 8 Kbyte direct-mapped cache. From Figure 6-3, we see that a victim buffer improves both performance and energy for some benchmarks, like *mpeg, epic*, and *adpcm*. For other benchmarks, energy is not improved but performance is degraded, as for *vpr, fir,* and *padpcm*. A victim buffer should be excluded or turned off for these benchmarks. Some benchmarks, like *jpeg, parser*, and *auto2*, yield some energy savings at the expense of some performance degradation using a victim buffer – a designer might choose whether to include/exclude or turn on/off the buffer in these cases depending on whether energy or performance is more important.

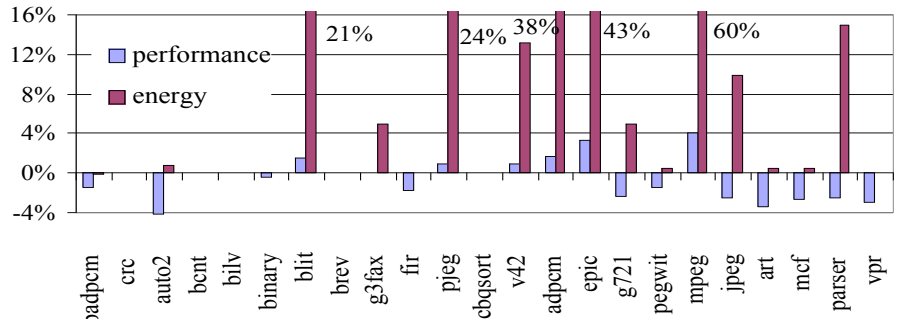

Figure 6-3. Performance and energy improvements when adding a victim buffer to an 8 Kbyte direct-mapped cache. Positive values mean the victim buffer improved performance or energy, with 0% representing an 8 Kbyte direct-mapped cache without a victim buffer. Benchmarks with both bars positive should turn on the victim buffer, while those with negative performance improvement and little or no energy improvement should turn off the victim buffer.

5.2.2 Victim Buffer with a Parameterized Cache

Figure 6-4 shows the performance and energy improvement of adding a victim buffer to a parameterized cache having the same configurability described by Zhang et. al[2]. 0% represents the performance and energy of the original configurable cache when tuned optimally to a particular application. The bars represent the performance and energy of the configurable cache when optimally tuned to an application assuming a victim buffer exists and is always on. The optimal cache configurations for a given benchmark are usually different for each of the two cases (no victim buffer versus always-on victim buffer).

We see that, even though the configurable cache already represents significant energy savings compared to either a 4-way or direct-mapped cache[2], a victim buffer extends the savings of a configurable cache by a large amount for many examples. For example, a victim buffer yields an additional 32%, 43%, and 23% energy savings for benchmarks *adpcm, epic,* and *mpeg2*. The savings of *adpcm* and *epic* come primarily from the victim buffer that reduces the visits to off-chip memory. The saving of *epic* comes primarily from the victim buffer enabling us to configure the configurable cache to use less associativity without increasing accesses to the next memory level. Yet, for other benchmarks, like *adpcm*, *auto2* and *vpr*, the victim buffer yields performance overhead with no energy savings and thus should be turned off.

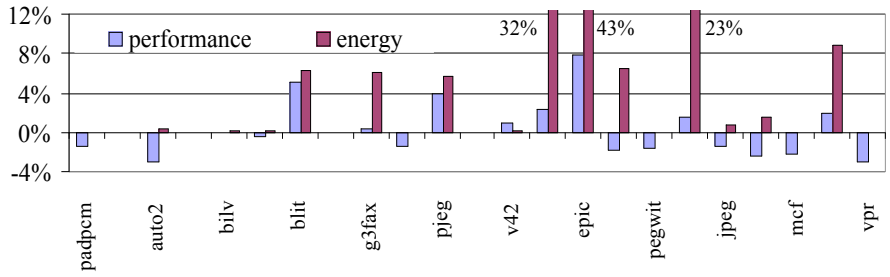

Figure 6-4. Performance and energy improvements when adding a victim buffer to an 8 Kbyte configurable cache. 0% represents a configurable cache without a victim buffer, tuned optimally to the particular benchmark.

6. LOW STATIC-POWER FREQUENT-VALUE DATA CACHES

Recently, a frequent value (FV) low power data cache design was proposed based on the observation that a major portion of data cache accesses involves frequent values, which can be dynamically captured[27]. Frequent values are encoded in the cache, occupying only a few bits.

We improve upon previous FV data caches by reducing static power by shutting off the unused bits in the larger sub-array for encoded frequent values[28]. Since frequent values are stored in encoded form using only the few bits in the smaller sub-array, the remaining bits in the larger sub-array serve no purpose as long as the value stays frequent. Such shutoff may be beneficial since FVs occupy many words in data caches[27].

Furthermore, the original FV low power cache design suffers from an extra cycle when reading non-FVs[27], which account for 68% of all data

cache accesses, resulting in a 5% increase in execution time. We used circuit design to remove the extra cycle.

6.1 Overview of Original FV Cache Design

In this section, we give a brief overview of the original FV data cache designed by Yang and Gupta[27].

The FV cache was proposed based on the observation that a small number of distinct frequently occurring data values often occupy a large portion of program memory data spaces and therefore account for a large portion of memory accesses[27]. This frequent value phenomenon was exploited in designing a data cache that trades off performance with energy efficiency.

From the perspective of the frequent value cache, data values are divided into two categories: a small number of frequent values, in our case 32 FVs, and all remaining values that are referred to as non-frequent values. The frequent values are stored in encoded form and therefore can be represented in 5 bits; the non-frequent values are stored in unencoded form in 32 bit words. Additionally, a flag bit is needed for each word in the cache to determine if the value stored in that location is encoded or not. The set of frequent values remains fixed for a given program run.

When reading a word from the cache, initially we simply read from the low-bit array. Since every word read out contains a flag bit, the flag is examined to determine what comes next. The flag being 1 means the desired word is in un-encoded form, so the remaining bits should be read out from the high-bit array to form the original value. On the other hand, the flag being 0 means that the desired word is a frequent value and stored in encoded form. In this case, the access proceeds to decode the value. Since the access to the high-bit array is avoided, cache activity is reduced.

A write to the FV cache is performed as follows. Before a value is written, it is first encoded through an encoder. If encoding is successful, it means that the value is a frequent value and thus a 5-bit code is stored in the low-bit array and the flag bit is cleared. In this case, accessing the high-bit array is avoided. If the encoding fails, the value to be written is a non-frequent value and thus both low-bit and high-bit data arrays are accessed as well as the flag bit being set. Note that writing non-FVs does not need to take two cycles as does reading non-FVs, because the value is encoded early in the pipeline and thus the decision of driving one array or two is clear before the access.

6.2 Improving the FV Cache Design

The FVs are not only *accessed* frequently, but also *distributed* widely in caches[29]. This phenomenon provides a good opportunity for reducing static power. Our approach is the following. Since the 32-bit FVs are encoded in 5 bits, the remaining 27 bits do not store any useful information. Therefore, they can be shut down to save static power and as long as a value stays frequent, static power is saved. The overall savings depend on the occupancy of FVs in the cache. Our studies show that on average nearly half of the cache content contains FVs, which indicates the benefit of reducing static power through finding FVs.

The flag bits are initially set to 1, which means initially all words are non-FVs. Any data to the data cache is checked with the FV encoder. If the word is an FV, the corresponding flag bit is set to 0 and this cache word is encoded and stored in the 5-bit array. At the same time, the flag bit turns off the 27-bit portion of the word. Similarly, on reading FVs, only the 5-bit portion is read and the 27-bit portion is gated off using the flag bit. On a non-FV read or write, the flag bit is set to 0 and the original 32 bits are written into the cache as usual. Our new circuit design improves the original FV cache design in that there is no extra delay in determining accesses of the 27-bit portion.

6.3 Designers' Choices of Using the FV Cache

We have described a low static power FV cache. When utilized into a processor system, the FV cache can be designed with different degrees of complexity and flexibility. In this section, we provide three approaches that are suitable for a variety of processors targeting different types of applications. Essentially, the complexity comes from how FVs are identified and if they are allowed to vary for different applications. As always, the more flexibility the processor provides, the more complex the FV cache is.

The first approach is appropriate to application specific processors. Since only a single type of application runs on the processor, its FVs tend to be stable over time. In such cases, the FVs can be first obtained from a profiling run through simulations, and then synthesized into the cache as part of the cache data storage. The advantage of this approach is that once the FVs are hard coded on-chip, the cache does not perform operations other than reads. Thus, the logic of this component is simple and can be designed to consume minimum power.

The second approach extends the first one with the ability of changing the FVs according to different applications. This approach is suitable for a multi-task environment in which the processor runs multiple programs

instead of single program. Each program's FVs are still obtained off-line. Instead of synthesizing the FVs on-chip, a register file may be used to store FVs so that they can be rewritten on each activation of a different program. The size of the register file depends on the number of FVs of interest to the designer, which is heavily dependent on each program's behavior.

The third approach provides the maximum flexibility in maintaining FVs. According to a previous study[29], some programs' FVs are sensitive to different inputs. This suggests that another dimension of varying FVs might be added into the design. Since it is infeasible to profile every program on all possible inputs to catch FVs, detecting FVs on-line would be useful. Thus, on top of the second approach, the register file could be extended to dynamically capture FVs using extra logic. In the scheme proposed by Yang and Gupta[27], an inexpensive hardware FV finder was developed that monitored cache accesses. The FV finder was turned on for only the first 5% of memory accesses assuming that the total memory access numbers are known a priori. After that, the FVs were captured in the finder and transmitted to the cache so that the cache starts operating as an FV cache. The energy overhead of the finder was estimated to be 0.3%-6.1% of the L1 D-cache (8 Kbyte to 64 Kbyte caches were tested). The area overhead is similar to our second approach, and thus modest. One potential issue is that the FV finder described detects frequently *accessed* values, which may or may not correspond to frequently *distributed* values in memory, though they usually are the same. We leave an FV finder for frequently distributed values for future work.

6.4 Experiments and Results

To determine the benefits of our FV cache architecture in reducing static energy, we ran 11 SPEC2000[26] benchmarks through the SimpleScalar tool set[19]. We used a 4-issue out-of-order processor simulator with a 32 Kbyte L1 instruction and data cache. The benchmarks were fast-forwarded for 1 billion instructions and executed for 500 million instructions afterwards, using reference inputs.

6.4.1 Static Energy Savings

Our main goal is to reduce the static energy consumed by the data cache without losing performance. As mentioned earlier, the overall static energy saving depends on the average coverage of FVs inside data cache. Through experiments, we found that there are abundant FVs in the L1 data cache at any time for Spec 2000 benchmarks, as shown in Figure 6-5. The percentage shown is the average for the 500 million instructions execution time. On

average, 49.2% of the total words are FVs, with the highest being 77.0% for benchmark *mcf* and the lowest 9.4% for benchmark *ammp*. The static energy savings are proportional to the number of FVs in the data cache. Thus, the corresponding static energy savings on average are 35% (49.2%_27/33_86%) considering that 27 bits out of 33 bits (we need a flag bit per 32-bit word) are shut off and 86% of static power can be saved using a pMOS Gated-V_{dd}. When compared with the conventional 32-bit per word cache, the static energy savings can be calculated as 100%- (100%-35%)*33/32 = 33%.

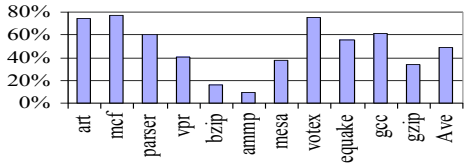

Figure 6-5. Percentage of data cache words that are FVs

6.4.2 Performance Improvement

Our second achievement is the performance improvement over the original FV data cache design. Recall that the original FV cache performance overhead was due to the prolonged non-FV accesses. The more non-FV accesses, the slower the execution and the less the overall power savings (less energy savings), since the system would consume more energy when the program runs longer. We measured the average percentage of cache hits that are FVs, as shown in Figure 6-6(a). On average, the hit rate on data FVs is 32% with the highest being 62.7% for *votex* and the lowest 11.4% for *mcf*. Therefore, we can see that on average, 68% of cache accesses are non-FVs.

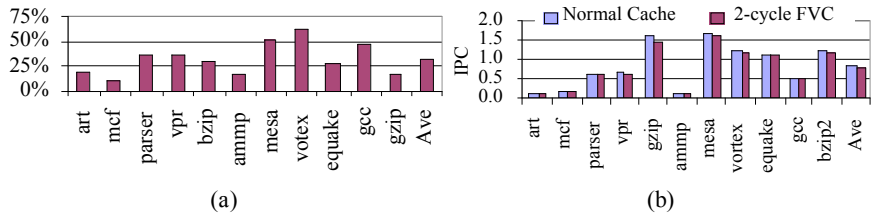

Figure 6-6. (a) Hit rate of FVs in data cache; (b) Performance (IPC) degradation of two-cycle FV cache

With our improved circuitry (1-cycle latency for non-FVs as well as for FVs), we are able to maintain the same execution speed as the base case. To see how much performance we have gained over the original FV cache, we measured the IPCs for a normal cache and a 2-cycle FV cache and plot them in Figure 6-6(b). The IPC for our improved design is the same as the normal cache. Figure 6-6(b) shows the slowdowns of the original FV cache design, which is the same value as our performance improvement. We can see that there is a 5.2% difference in the averaged IPCs between the original FV cache and our improved version. This also means that in addition to the static energy we saved by shutting off partial FV words, we also saved more dynamic energy than the original FV cache design.

Another feature in our new design is that it is *safe* in the sense that it does not increase power consumption significantly even when FVs are not abundant. Thus, our improved FV cache design is an appealing approach in reducing both static and dynamic energy of caches.

7. ACKNOWLEDGEMENTS

This work was supported by the National Science Foundation (CCR-0203829, CCR-9876006) and by the Semiconductor Research Corporation (2003-HJ-1046G).

8. REFERENCES

1. S. Segars. Low power design techniques for microprocessors, International Solid State Circuit Conference, February 2001.
2. C. Zhang, F. Vahid, and W. Najjar. A highly-configurable cache architecture for embedded systems. 30th Annual International Symposium on Computer Architecture, June 2003.
3. Altera, Nios Embedded Processor System Development, http://www.altera.com/corporate/news_room/releases/products/nr-nios_delivers_goods.html.
4. Arc International, www.arccores.com.
5. ARM, www.arm.com.
6. MIPS Technologies, www.mips.com.
7. Tensilica, Xtensa Processor Generator, http://www.tensilica.com/.
8. D. H. Albonesi. Selective Cache Ways: On Demand Cache Resource Allocation. Journal of Instruction Level Parallelism, May 2002.
9. A. Malik, W. Moyer, and D. Cermak. A Low Power Unified Cache Architecture Providing Power and Performance Flexibility. International Symposium on Low Power Electronics and Design, 2000.
10. R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory Heirarchy Reconfiguration For Energy and Performance in General-Purpose Processor Architecture. 33rd International Symposium on Microarchitecture, December 2000.

11. A. Veidenbaum, W. Tang, R. Gupta, A. Nicolau, and X. Ji. Cache Access and Cache Time Model. IEEE Journal of Solid-State Circuits, Vol 31, No 5, 1996.
12. T. Givargis and F. Vahid. Platune: A Tuning Framework For System-On-a-Chip Platforms. IEEE Transactions on Computer Aided Design, November 2002.
13. M. Palesi and T. Givargis. Multi-Objective Design Space Exploration Using Genetic Algorithms. International Workshop on Hardware/Software Codesign, May 2002.
14. C. Zhang and F. Vahid. Cache Configuration Exploration on Prototyping Platforms. 14th IEEE International Workshop on Rapid System Prototyping , June 2003.
15. A. Ghosh and T. Givargis. Cache Optimization For Embedded Processor Cores: An Analytical approach. International Conference on Computer Aided Design, November 2003.
16. C. Zhang, F. Vahid, and R. Lysecky. A Self-Tuning Cache Architecture for Embedded Systems. Design Automation and Test in Europe Conference (DATE), February 2004.
17. M. Powell, A.Agarwal, T. Vijaykumar, B. Falsafi, and K. Roy. Reducing Set-Associative Cache Energy via Way-Prediction and Selective Direct Mapping, 34th International Symposium on Microarchitecture, 2001.
18. C. Lee, M. Potkonjak, and W.H. Mangione-Smith. MediaBench: A Tool For Evaluating and Synthesizing Multimedia and Communication Systems. Proc 30th Annual International Symposium on Microarchitecture, December 1997.
19. D. Burger, T. Austin, and S. Bennet. Evaluating Future Microprocessors: The Simplescalar Toolset. University of Wisconsin-Madison. Computer Science Department Tech. Report CS-TR-1308, July 2000.
20. http://www.mips.com/products/s2p3.html, 2003.
21. A. Gordon-Ross, F. Vahid, and N. Dutt. Automatic Tuning of Two-Level Caches to Embedded Applications. Design Automation and Test in Europe Conference (DATE), February 2004.
22. EEMBC, the Embedded Microprocessor Benchmark Consortium, www.eembc.org.
23. N. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers, Proceedings of International Symposium on Computer Architecture, 1990.
24. G. Albera and R. Bahar. Power/performance Advantages of Victim Buffer in High-Performance Processors, IEEE Alessandro Volta Memorial Workshop on Low-Power Design, 1999.
25. C. Zhang and F. Vahid. Using a Victim Buffer in an Application-Specific Memory Hierarchy. Design Automation and Test in Europe Conference (DATE), February 2004.
26. http://www.specbench.org/osg/cpu2000.
27. J. Yang and R. Gupta. Energy Efficient Frequent Value Data Cache Design, Int. Symp. on Microarchitecture, Nov. 2002.
28. C. Zhang, J. Yang, and F. Vahid. Low Static-Power Frequent-Value Data Caches. Design Automation and Test in Europe Conference (DATE), February 2004.
29. J. Yang and R. Gupta. "Frequent Value Locality and its Applications," ACM Transactions on Embedded Computing Systems (inaugural issue), Vol. 1, No. 1, pages 79-105, November 2000.