Experimental Evaluation of A Compiler-Based Cache Energy Optimization Strategy

M. Kandemir
Pennsylvania State University
kandemir@cse.psu.edu

I. Kolcu
UMIST
ikolcu@co.umist.ac.uk

I. Kadayif
Pennsylvania State University
kadayif@cse.psu.edu

Abstract

In this paper, we present experimental results from an optimization strategy that aims at reducing per access energy cost for direct-mapped data caches. We have developed a compiler algorithm that uses access pattern analysis to determine those references that are certain to result in cache hits (called 'certain hits') in a virtually-addressed, direct-mapped data cache. After detecting such references, the compiler substitutes the corresponding load operations with 'energy-efficient loads' that access only data array of cache instead of both tag and data arrays. This tag access elimination, in turn, reduces the per access energy consumption for data accesses. Our experimental results indicate that certain hits constitute a large percentage of total hits. They also show that even our most conservative strategy improves the data cache energy consumption by 11% on the average.

1 Introduction

Many optimizing compilers do not perform any energy-specific optimizations, and cannot cope well with increasing energy demands put forward by embedded/portable as well as high-performance applications. Cache memories are known to consume a large percentage of on-chip energy in current systems. While there exists a large body of compiler-based work to manipulate access pattern of a given code to improve its cache performance [5], there are not many compiler techniques that try to improve cache energy consumption.

In this paper, we present detailed experimental results of a compiler-based cache energy optimization strategy. This strategy analyzes a given array-intensive application and marks some load operations as certain hits. These loads are then treated by the architecture differently than traditional load operations. More specifically, our optimization strategy consists of the two steps briefly explained in the following two subsections.

1.1 Identifying Certain Hits

The compiler analyzes an array-intensive application and determines which array accesses are certain to be hits at runtime (i.e., during execution). As an example, consider the following loop assuming a cache with four-element wide lines:

```
for(i=1;i<L;i++)
 { a[i] }
```

Normally, to handle the array access `a[i]`, the compiler generates only one load operation (omitting the potential load due to loop index variable `i`). However, a closer look at the memory accesses reveals that (assuming that the first element of the array is aligned with a cache line boundary), only the first iteration of every four successive loop iterations will miss in the cache. That is, the remaining three iterations (in every four successive iterations) will be hits. Consequently, our strategy should generate a different type of load (called energy-efficient load) for the array accesses that will generate hits.

1.2 Modifying Code

To expose array references with different types of load requirements (normal vs. energy efficient), we use loop unrolling. Loop unrolling reduces the number of iterations and replicates the body of the loop [5]. For example, unrolling the loop shown above generates the following code (note that the new loop step size is 4 and we assume that 4 divides L evenly):

```
for(i=1;i<L;i=i+4)
 {
  { a[i] }
  { a[i+1] }
  { a[i+2] }
  { a[i+3] }
 }
```

0-7803-7494-0/02/$17.00 © 2002 IEEE

Now, one can clearly see that we need to use normal load operations only for the first reference in the loop body. The remaining references can use energy-efficient loads.

In cases where it is not possible to align arrays in memory such that conflict misses are eliminated with 100% accuracy, the compiler can use a data space (array layout) transformation called array interleaving. This transformation takes multiple arrays, and maps them into a single array. This mapping should be one-to-one (i.e., each array element should be mapped into a unique place, and no two array elements should be mapped into the same place in the new array) and, after the mapping, the array references in the program and array declarations should be modified (i.e., transformed) accordingly. If, for example, in a given loop array references `a[i]`, `b[i]`, and `c[i]` all map to the same location in the cache, this can cause significant performance/energy penalties owing to cache conflicts. Consequently, if the compiler cannot guarantee conflict-free accesses, it should be conservative and assume that each access will result in a miss. Alternatively, the compiler can apply array interleaving as follows. Let us consider the following mappings of arrays a, b, and c into the common data (array) space u: `a[i]` $\longrightarrow$ `u[3i-2]`; `b[i]` $\longrightarrow$ `u[3i-1]`; and `c[i]` $\longrightarrow$ `u[3i]`. Then, the compiler replaces the array references in the code with these transformed references; and after this, our approach can be applied.

In order to implement energy-efficient loads, we also need some support from the hardware. More specifically, when an energy-efficient load operation is encountered, the tag check should be skipped, and directly data array should be accessed. This improves the cache access energy as tag arrays might be as large as data arrays. Our compiler-based optimization algorithm is summarized in [4]. In this work, we present an extensive experimental evaluation of this strategy using representative signal processing applications and some other array-intensive codes.

2 Experiments

2.1 Energy Calculations

The energy values reported in this section are calculated using an extended version of formulations given in These formulations are an extended form of the model given by Shiue and Chakrabarti [6]. Our extensions isolate the contributions of certain hits and the remaining hits and take into account write accesses (in addition to read accesses).

2.2 Benchmarks Codes and Methodology

We evaluate the proposed optimization strategy using kernels from the DSPstone benchmark suite as well as six larger array-intensive benchmarks. The original DSPstone codes that use pointers have been converted to their array-based counterparts using the approach presented in [2]. The important characteristics of our codes are given in Figure 1. `aps` and `wss` are from Perfect Club Benchmarks; `btrix`, `tomcatv`, and `vpenta` are from Spec Benchmarks; and `adi` is an array-intensive code that performs ADI computation.

Benchmark Name	Input Size	Number of Arrays
DSPstone		
`adpcm`	1.00MB	4
`convolution(conv)`	2.00MB	2
`dot_product(dot)`	2.00MB	2
`fft`	224.00KB	2
`fir`	2.18MB	2
`fir2dim`	512.00KB	4
`iir_biquad_N_sections(iir)`	5.10MB	5
`matrix`	192.00KB	3
`n_real_updates(real)`	1.60MB	4
`n_complex_updates(complex)`	1.60MB	4
Miscellaneous		
`adi`	235.55 KB	6
`aps`	2.75 MB	17
`btrix`	124.64 MB	29
`tomcatv`	273.52 KB	9
`vpenta`	322.88 KB	9
`wss`	122.30 KB	10

Figure 1. Benchmark codes and their important characteristics.

We obtain a breakdown of hits into different types using the following strategy. For each code, we first apply our compiler analysis written using the SUIF [1] experimental compiler system from Stanford University, and determine the number of certain hits and the total number of data references. Scalar variables are treated as single-element wide arrays. Then, we feed the same code to the Shade simulator, and obtain the actual number of hits and misses. The difference between the number of hits returned by the simulator and the number of certain hits determined by our compiler analysis gives us the remaining (other) hits (i.e., the hits that could not be detected by the compiler statically). After that, we pass the number of certain hits, the number of other hits, and the number of misses to our energy formulations.

Unless stated otherwise, the default data cache configuration is a 16KB direct-mapped cache with a line size of 16 bytes. We also assume a 64 bit virtual address and 5 status bits in all experiments. It should be mentioned that the compiler detection of certain (definite) hits are 100% accurate; we compared the compiler output with the simulator output (address by address) and verified the accuracy of the compiler analysis. Except for the last part of our experimental results, we do not assume that any array variable is register allocated.

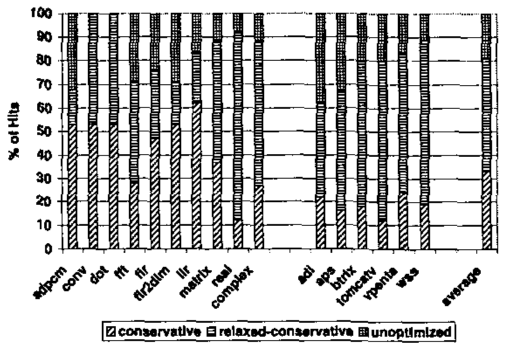

Figure 2. Percentage breakdown of different types of hits ('relaxed-conservative' means the difference between 'relaxed' and 'conservative').

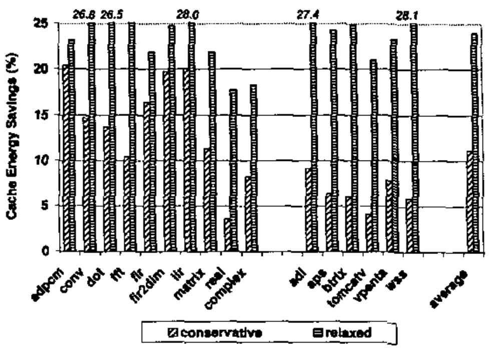

Figure 3. Data cache energy savings.

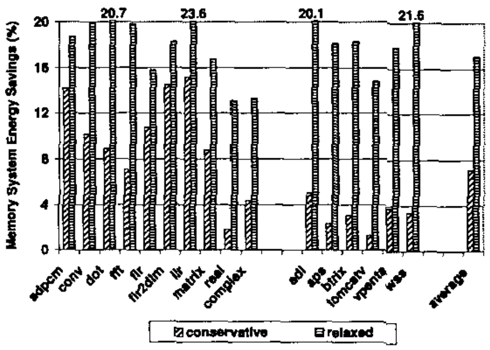

Figure 4. Memory system energy savings.

2.3 Results

Figure 2 shows the ratio of certain hits to total hits under two different schemes. In the first scheme, called the *conservative* scheme, only the successive uninterrupted accesses to a cache line are counted as certain hits. In the second scheme, which we call the *relaxed* scheme, the compiler tries to determine the distance between potentially conflicting references; only if this distance can be shown (with 100% accuracy) to be less than cache size, a certain hit is assumed. We see from these results that even in the conservative case, certain hits constitute a large percentage of total hits (around 33.5%), indicating there there is a significant scope for our optimization strategy.

Figure 3 gives the percentage savings in cache energy due to our optimization using a 16KB cache with a block size of 16 bytes for both the conservative and the relaxed strategies. Unless explicitly stated, all the energy improvement numbers reported here are *percentage reductions* with respect to the energy consumption of a conventional direct-mapped cache of the same size. We observe that, for the conservative scheme, the percentage benefits range from 3.5% and 20.4% (averaging in 11.13%). The relaxed strategy is, on the other hand, more aggressive and improves the cache energy consumption by 23.98% on the average. We also see from the figure that the larger benchmarks do not get much benefit from the conservative scheme; this is because of the large number of array references that prevent uninterrupted successive accesses to cache lines. When we employ the more aggressive relaxed scheme, however, this picture changes and the energy benefits become as much as 28.1%. It is also important to see how much memory system energy is saved as a result of our technique. Note that memory system energy includes the energies expended in the cache, address and data pads/buses, and off-chip main memory. The results reported in Figure 4 indicate that memory system energy savings range from 1.3% to 15.2% for the conservative strategy, and from 13.1% to 23.6% for the relaxed scheme.

To study the sensitivity of our approach to cache configuration, we performed another set of experiments where we measured the effectiveness of our approach under different data cache configurations. Note that since we focus only on direct-mapped caches, the configurations explored here differ from each other only in cache size and line (block) size. The results given in Figure 5 show that our strategy generates good results across all configurations experimented. We also observe that the cache line size has much more impact than the cache size as it contributes greatly to the width of the data array.

2.4 Impact of Optimization and Set-Associative Caches

Figure 6 shows the impact of array interleaving on energy benefits for our larger benchmarks. As shown in the figure, it improves the energy benefits of the relaxed strategy by 8.67% on average. This data transformation, however, did not bring much benefit in the DSPstone codes as the relaxed

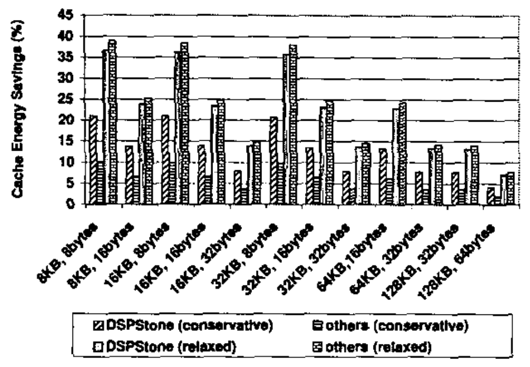

Figure 5. Cache energy savings for different configurations.

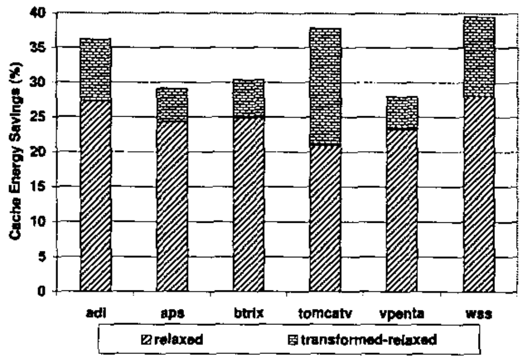

Figure 6. Impact of array interleaving ('transformed-relaxed' means the difference between 'transformed' and 'relaxed').

scheme was already able to optimize almost all array references successfully.

To compare the energy consumption of our approach to set-associative caches, the next set of experiments measured the energy benefits brought about by different set-associative caches over direct-mapped caches (of the same capacity). Figure 7 shows the memory system energy benefits for three set-associative caches (2-way, 4-way, and 8-way) (over a direct-mapped cache). All caches are 16KB with a line size of 16 bytes. For comparison, the figure also reproduces the energy benefits of our relaxed version. It is easy to see that the energy reductions generated by our energy optimization strategy are competitive with those coming from set-associative caches. In only two cases, a set-associative cache resulted in higher energy savings than our strategy. We also note that increasing associativity is not always beneficial from the energy perspective. This is because, in some cases, higher associativity does not bring the expected reductions in cache misses, and the higher per access energy cost of a set-associative cache makes it less efficient energy-wise.

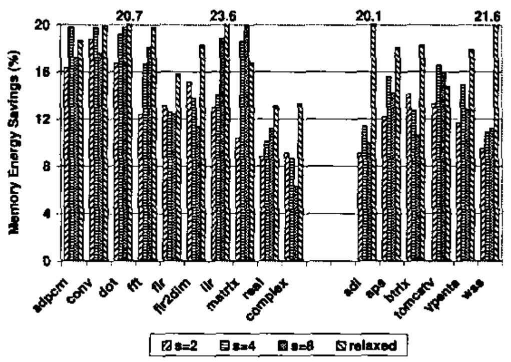

Figure 7. Comparison with set-associative caches.

2.5 Comparison with Block Buffering

A block buffer is a small line buffer inserted between the processor and the first-level cache to hold the most recently accessed cache line (block). If the cache line that resides in the block buffer is accessed again, this request can be satisfied from the block buffer, and the first level cache (both data and tag arrays) can be disabled during this access to save energy. The previous research [3] investigated different block buffering schemes, and evaluated their impact on energy and performance. Since block buffering is trying to achieve the same objective as our optimization strategy discussed in this paper (i.e., reducing per access cache energy cost), we wanted to compare these two techniques to evaluate their relative effectiveness.

Figure 8 shows the benefits due to block buffering over a direct-mapped cache without any optimization. We conducted experiments with different numbers of buffers (1, 2, and 4). For convenience, the benefits due to our techniques (both conservative and relaxed cases) are also repeated here. We observe from these results that, as expected, block buffering improves energy more than the conservative approach, since in block buffering, both the tag and data array accesses can be avoided (in a hit). Increasing the number of entries in the buffer improves the energy savings further up to a point beyond which the block buffer itself starts to consume a significant amount of energy. But, when we use the relaxed approach, we achieve, in most cases, larger savings than block buffering could, as our approach can optimize for large number of arrays simultaneously. In contrast, the optimization capability of block buffering is limited by the number of buffers.

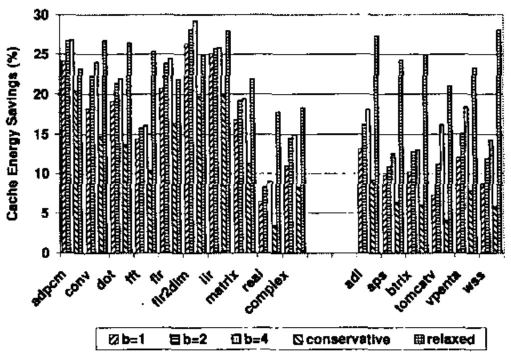

Figure 8. Impact of block buffering.

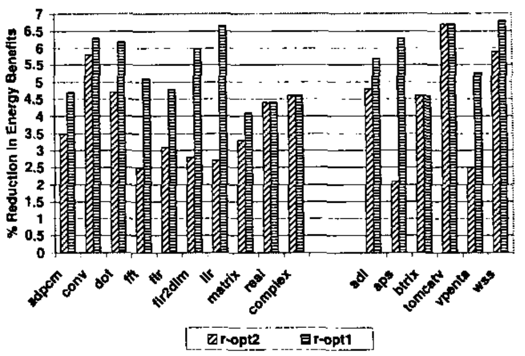

Figure 9. Impact of register allocation.

2.6 Impact of Register Optimization

A good register allocator captures in a register the successive accesses to a given variable [5]. This reduces the number of memory load operations, and can decrease the overall energy consumption. To study the interaction between register allocation and our optimization strategy, we first consider the best case from the register allocation perspective. In this case, we assume that all successive accesses to an array variable (except the first one) are always satisfied from a register. So, our strategy can only be used for the remaining array references. It is also possible to employ a less aggressive register allocation strategy where only the *uninterrupted* successive accesses (i.e., the successive accesses with no intervening variable access) are satisfied from registers. Obviously, these two cases can generate very different cache access patterns.

Placing array variables into registers reduces the effectiveness of our energy optimization strategy. This is due to two main reasons. First, in the existence of register allocation, there are fewer cache references to optimize. Second, in general, the references not allocated in any register form (collectively) a more irregular cache access pattern (as compared to the original access pattern). Figure 9 gives the percentage reductions in the effectiveness of our approach due to two different register allocation styles mentioned above. r-opt1 is a very aggressive allocator, and guarantees that a reuse of each variable is satisfied from a register. r-opt2 is less aggressive and satisfies from registers only the array accesses not intervened by accesses to other virtual lines. It can be observed from Figure 9 that the existence of register allocation reduces the benefits due to our strategy by 2.1%-6.8%. However, our strategy is still very successful in reducing the overall energy consumption.

3 Conclusions

This paper discusses the results from a cache energy optimization strategy based on access pattern analysis. The idea is to determine the accesses that will result in hits (with a hundred percent accuracy) in a direct-mapped data cache and skip tag array accesses for such references. We also discuss results from array interleaving, a data-centric transformation, that can enhance the effectiveness of our strategy and give results when register allocation is employed. Our results show that there is a large scope in array-intensive codes for such an optimization, and that the proposed technique is very effective in reducing cache energy consumption.

References

[1] S. P. Amarasinghe, J. M. Anderson, M. S. Lam, and C. W. Tseng. The SUIF compiler for scalable parallel machines. In Proc. *the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, February, 1995.

[2] B. Franke and M. F. P O'Boyle. Compiler transformation of pointers to explicit array accesses in DSP applications. In Proc. *International Conference on Compiler Construction*, Genoa, Italy 2001.

[3] K. Ghose and M. B. Kamble. Reducing power in superscalar processor caches using subbanking, multiple line buffers and bit-line segmentation. In Proc. *ISLPED'99*, San Diego, CA, 1999, pp. 70–75.

[4] M. Kandemir and I. Kolcu. Reducing cache access energy in array-intensive applications. In Proc. *the 5th Design Automation and Test in Europe Conference, (poster paper)* Paris, France, March, 2002.

[5] S. S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufmann Publishers, San Francisco, California, 1997.

[6] W.-T. Shiue and C. Chakrabarti. Memory exploration for low power embedded systems. *Technical Report CLPE-TR-9-1999-20*, Arizona State University.