VLSI DESIGN
2001, Vol. 12, No. 2, pp. 151–165
Reprints available directly from the publisher
Photocopying permitted by license only

Investigating Memory System Energy Behavior Using Software and Hardware Optimizations*

G. ESAKKIMUTHU[†], H. S. KIM[‡], M. KANDEMIR[¶],
N. VIJAYKRISHNAN[§] and M. J. IRWIN[#]

Computer Science and Engineering Department, The Pennsylvania State University, University Park, PA 16802

(Received 20 June 2000; In final form 3 August 2000)

Memory system usually consumes a significant amount of energy in many battery-operated devices. In this paper, we provide a quantitative comparison and evaluation of the interaction of two hardware cache optimization mechanisms and three widely used compiler optimization techniques used to reduce the memory system energy. Our presentation is in two parts. First, we focus on a set of memory-intensive benchmark codes and investigate their memory system energy behavior due to data accesses under hardware and compiler optimizations. Then, using four motion estimation codes, we look at the influence of compiler optimizations on the memory system energy considering the overall impact of instruction and data accesses.

Keywords: Memory system energy; Block buffering; Sub-banking; Compiler optimizations; On-chip caches; Sub-banks

INTRODUCTION

Hardware and software techniques to reduce energy consumption have become an essential part of current system designs [27]. Such techniques have particularly targeted the memory system due to the prevalent use of data-dominated signal and video applications in mobile devices. Various low power circuit techniques [4], energy efficient memory and cache architectures [1, 9, 12, 7, 19], and power-aware compilation techniques [16, 3] have been proposed. However, there is still much work to be done in understanding the interaction of hardware and software optimizations and in evaluating their relative energy gains.

* This project was funded in part by the Pittsburgh Digital Greenhouse through a grant from the Commonwealth of Pennsylvania, Department of Community and Economic Development.
[†]e-mail: geethanj@cse.psu.edu
[‡]Corresponding author. Tel.: (814) 863-7325, Fax: (814) 865-3176, e-mail: hykim@cse.psu.edu
[¶]e-mail: kandemir@cse.psu.edu
[§]e-mail: vijay@cse.psu.edu
[#]e-mail: mji@cse.psu.edu

This paper explores the interaction of hardware and software optimizations by considering the memory system energy savings obtained when using energy-efficient cache architectures and compiler optimization techniques. Specifically, we set out to answer following two questions.

- As far as data caches are concerned, what is the influence of compiler optimizations and hardware enhancements on on-chip cache/off-chip memory energy? Apart from presenting a comprehensive energy evaluation of hardware and compiler techniques, such a study will also reveal how these optimizations interact with each other.
- How do the variations in energy due to data accesses and energy due to instruction accesses compare when performance-oriented compiler optimizations are applied?

Consequently, our presentation is in two parts. In the first part of our presentation (Section 2), we first apply the hardware and software optimization techniques individually using the original codes and base cache configurations, respectively, in order to evaluate their relative energy savings in the memory system. This evaluation is performed using a suite of data-intensive codes that operate on multi-dimensional arrays which are typical in many embedded video and signal processing applications. Further, energy consumption is estimated using an energy model for a memory system that consists of an on-chip data cache and an off-chip main memory.

Next, we evaluate various energy-efficient on-chip cache configurations with different combinations of cache sub-banking and block buffering optimizations [19, 11]. These techniques help to reduce the cache energy by limiting the data access to just one bank of a cache or to just the previously accessed cache line. We find that these optimizations result in an energy saving of 44–89% on an average in the data cache. After that, the energy consumption of the memory system (considering data cache and off-chip memory) was evaluated after applying three widely used code optimization techniques, namely, linear loop transformations (*e.g.*, loop permutation, loop skewing, *etc.*), loop tiling and loop unrolling [14, 21, 22]. In this case, we find that the energy consumption in the data cache increases on an average by 51%. This is due to the increase in data accesses as a result of applying these optimizations in conjunction with scalar replacement [14]. However, considering just the cache energy consumption is misleading as the entire memory system energy reduces by 42–79% after applying these transformations. This effect is due to the decreased number of accesses to off-chip memory due to improved locality after applying the optimizations. These results also corroborate a popular belief that most significant energy gains can be obtained at the higher levels of system design [8].

We find that the compiler optimizations reduce the overall memory system energy consumption by up to 75% (79%) as compared to the maximum of 4% (15%) reduction achieved when using 4 K (16 K) energy-efficient cache architectures. This result shows the importance of energy-aware software in system design. It must also be emphasized that the hardware optimizations are still important for reducing the on-chip energy consumption. As a final set of experiments in this part, we apply the hardware and software optimizations concurrently and find that the combination of the considered hardware and software optimizations results in up to 88% savings in energy consumption of the memory system due to data accesses.

In the second part of our presentation (Section 3), we focus on the energy consumption considering the entire memory system including both instruction and data accesses. This is important as many of the high-level compiler optimizations are oriented more towards improving the data locality rather than instruction locality and code size. Consequently, aggressive data locality optimizations may be detrimental from the code size and energy consumption for instruction accesses perspectives. These compiler optimizations are particularly useful for embedded image and video

applications that operate on large multidimensional arrays. Our results obtained using four motion estimation codes show that aggressive state-of-the-art compiler optimizations generally decrease the energy consumed due to data accesses while increasing the energy consumed due to instruction accesses. These results indicate the need for more research on techniques that simultaneously improve data and instruction locality/code size.

Overall, the results from this study will help compiler designers to make appropriate tradeoffs based on relative importance of the on-chip energy consumption and entire system energy consumption. Further, system designers can benefit from our results when selecting a combination of hardware and software optimizations (instruction and data) in designing an energy-efficient system.

Hardware Optimizations

A variety of hardware optimizations have been proposed to reduce the energy consumption [4]. In this paper, we focus on two cache optimizations, namely, block buffering [19, 9, 12] and cache sub-banking [19, 9, 12], as the cache is one of the major energy consuming components in current processors. We choose these cache optimizations since the effectiveness of block buffering is influenced by software optimizations while that of sub-banking is not. Also, neither technique affects performance in a noticeable way.

In the block buffering scheme, the previously accessed cache line is buffered for subsequent accesses. If the data within the same cache line is accessed on the next data request, only the buffer needs to be accessed. This avoids the unnecessary and more energy consuming access to the entire cache data array. Thus, increasing temporal locality of the cache line through software techniques can save more energy. In the cache sub-banking optimization, the data array of the cache is divided into several sub-banks and only the sub-bank where the desired data is located is accessed. This optimization reduces the per access energy consumption and is *not* influenced by locality optimization techniques. We also evaluate cache configurations that combine both these optimizations. In such a configuration, each sub-bank has an individual buffer. Here, the scope for exploiting locality is limited as compared to applying only block buffering as the number of words stored in a buffer reduces. However, it provides the additional benefits of sub-banking for each cache access.

Compiler Optimizations

In this section, we introduce the compiler optimizations evaluated (*i.e.*, linear loop transformations, tiling, and unrolling) and discuss their expected impact on energy consumption. While these optimizations are known to improve the memory performance characteristics of programs enormously [23], their impact on energy consumption needs further evaluation. Among the various high-level compiler transformations, we choose to target loop optimizations for two reasons. First, the multimedia and signal processing applications operate on multi-dimensional array structures that benefit from such optimizations. Second, these optimizations are widely used by commercial and academic optimizing compilers [22, 2].

Linear Loop Transformations

The linear loop transformations attempt to improve cache performance, instruction scheduling, and iteration-level parallelism by modifying the traversal order of the iteration space of the loop nest. The simplest form of loop transformation, called loop interchange [23, 21], can improve data locality (cache utilization) by changing the order of the loops. For example, consider the code shown in Figure 1a. Here, array `V` is copied to array `U` one column at a time. Since the C programming language stores multi-dimensional arrays in row-major order, a new cache line must be brought in for each element of column `i`. For a large value of `n`, the cache lines holding elements

```
for(i=0;i<N;i++)
 for(j=0;j<N;j++)
  U[j][i]=V[j][i];
```
(a)

```
for(j=0;j<N;j++)
 for(i=0;i<N;i++)
  U[j][i]=V[j][i];
```
(b)

```
for(i=0;i<n;i++)
 for(j=0;j<n;j++)
  for(k=0;k<n;k++)
   U[i][j]+=V[i][k]*W[k][j];
```
(c)

```
for(it=0;it<n;it+=t)
 for(jt=0;jt<n;jt+=t)
  for(kt=0;kt<n;kt+=t)
   for(i=it;i<min(it+t,n);i++)
    for(j=jt;j<min(jt+t,n);j++)
     for(k=kt;k<min(kt+t,n);k++)
      U[i][j]+=V[i][k]*W[k][j];
```
(d)

```
for(i=0;i<n;i+=b)
 for(j=0;j<n;j++)
  for(k=0;k<n;k++) {
   U[i][j]+=V[i][k]*W[k][j];
   U[i+1][j]+=V[i+1][k]*W[k][j];
   .........................;
   U[i+b-1][j]+=V[i+b-1][k]*W[k][j]; }
```
(e)

FIGURE 1 (a) A simple loop nest that performs array copying. (b) Loop-optimized version of (a). (c) Matrix-multiplication nest. (d) Tiled version of (c). (e) Loop-unrolled version of (c).

at the beginning of the column may be displaced from the cache before the next column is accessed, which, in turn, causes cache lines to be loaded multiple times. This problem can be handled by interchanging the order of the loops, which gives us the optimized code in Figure 1b that copies one row at a time. Note that this code causes all data in the same cache line to be accessed before the line can be displaced from the cache, thereby improving the cache performance. We expect this optimization to particularly benefit the block buffering optimization discussed earlier.

Loop Tiling

Another important technique used to improve cache performance is blocking, or tiling [21, 10]. When it is used for cache locality, arrays that are too big to fit in the cache are broken up into smaller pieces (to fit in the cache). As an example, consider the matrix multiplication code given in Figure 1c. If `U`, `V`, and `W` all fit in the cache, performance should be good. But if they are too big, performance might drop substantially, in which case an optimizing compiler tiles this nest as shown in Figure 1d (`t` denotes the tile size). As long as the tiles from the three arrays fit in the cache, we can expect a good performance [22]. We also expect a decrease in power consumed in memory, due to better data reuse after applying tiling. It must be noted that in comparison to linear loop transformations, tiling exploits temporal locality across multiple loop levels.

Loop Unrolling

Consider the code shown in Figure 1e. This is the same as the code in Figure 1c except that the outermost loop has been unrolled `b` times (assuming that `b` divides `n` evenly). The advantage of doing so is that `b` iterations of the `i` loop are executed in the inner loop and `W[k] [j]` only needs to be loaded into a register once for all these computations. More specifically, the same `W[k] [j]` value needs to be loaded once for each of the `U[i+0] [j], U[i+1] [j],...,U[i+b-1] [j]` computations (a total of `b` times). Thus, the outer loop unrolling can reduce the number of times that the elements of the array `W` need to be loaded by a factor of `b`. From the energy perspective, fewer accesses to the memory means less energy consumption.

It should be mentioned that these optimizations are generally applied in conjunction with scalar replacement, a transformation that converts array references to scalar variables. A negative impact of this is the increase in number of assignment statements, which may potentially increase the number of data accesses. However, loop unrolling compensates this effect by minimizing the number of load/store operations in the inner loop position. Although not addressed in this paper, it should be noted that these optimizations typically result in more complex array subscript functions and loop bounds which in turn can increase the overall energy consumption in the data path and instruction caches.

Experimental Strategy

In order to evaluate the effectiveness and interaction of the hardware and software optimizations, the C versions of benchmarks shown in Table I were used. All these codes are representative of the multi-dimensional array domain, the domain that many signal and video processing applications belong to. In this study, we zoom in on the `mxm` benchmark results when varying the different parameters and finally summarize the behavior across all benchmarks in the end. To determine the energy consumed by these codes, we obtained memory reference traces while executing the benchmarks using the *SimplePower* cycle-accurate simulator [24]. These traces were then analyzed using a *configurable memory system simulator* that was built in-house. The memory system simulator allows the configuration of cache sizes, block sizes, associativities, write and replacement policies, the number of cache sub-banks and cache block buffers used. In particular, we used the various cache configurations shown in Table II. *Henceforth, all the reported results will use the configuration numbers shown in this table.* Also incorporated in our memory system simulator is the on-chip cache energy model proposed in [11] using 0.8μ technology parameters [20], the off-chip main memory energy per access cost of the Cypress CY7C1326-133 chip and the I/O pad energy costs [18].

To evaluate the impact of compiler optimizations on the overall energy consumption, we used a high-level compilation framework based on loop (iteration space) and data (array layout) transformations. For this study, the framework proposed in [13] was enhanced with iteration space tiling and loop unrolling. Our enhanced framework takes as input a code written in C and applies the enabled optimizations to generate the optimized high-level C code. The tiling technique employed is similar to one explained in [22] and selects a suitable tile size for a given code, input size, and cache configuration. The loop unrolling algorithm carefully weighs the advantages of increasing data reuse and the disadvantages of larger loop body in selecting an optimal degree of unrolling (the parameter `b` in Fig. 1e) and is similar in spirit to the technique discussed in [22]. We generated the optimized codes after applying the loop transformation, tiling and unrolling optimizations individually and also when applied in combination.

Influence of Optimizations

Hardware Optimizations

First, we evaluate the influence of varying the number of sub-banks and block buffers on the cache energy consumption when using the original (unoptimized) codes. Figures 2 and 3 show the energy consumed in the data cache when different configurations shown in Table II are used for 4 K and 16 K caches, respectively, for the `mxm` benchmark. We find that increasing the number of sub-banks from one to two provides an energy saving of 45% for the data cache accesses. An additional 22% saving is obtained when the number of

TABLE I Programs used in the experiments

Program	Source	# of arrays	Input size (KB)	Instr. count
`tomcatv`	Specfp95	9	119	4,868,048
`nasa 7/btrix`	Specfp92	29	4,312	44,430,039
`nasa 7/mxm`	Specfp92	3	120	47,168,707
`nasa 7/vpenta`	Specfp92	9	114	2,274,945
`adi`	Livermore	6	241	6,062,860
`dtdtz (aps)`	Perfect club	17	1,605	42,119,337
`bmcm (wss)`	Perfect club	11	126	89,539,244
`psmoo (tfs)`	Perfect club	1	204	16,955,980
`eflux (tfs)`	Perfect club	5	297	12,856,306

TABLE II Hardware configurations used in this study. Indicates number of block buffers and sub-banks used. One sub-bank indicates that sub-banking is not employed

Configuration	# of block buffers	# of sub-banks
1	0	1
2	0	2
3	0	4
4	1	1
5	2	1
6	4	1
7	8	1
8	1	2
9	2	2
10	4	2
11	8	2
12	1	4
13	2	4
14	4	4
15	8	4

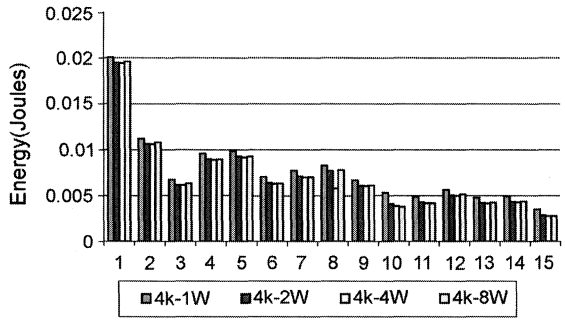

FIGURE 2 Energy consumed in *data cache* by the original mxm code for different hardware optimization configurations when using 4 K data caches with 32 byte blocks.

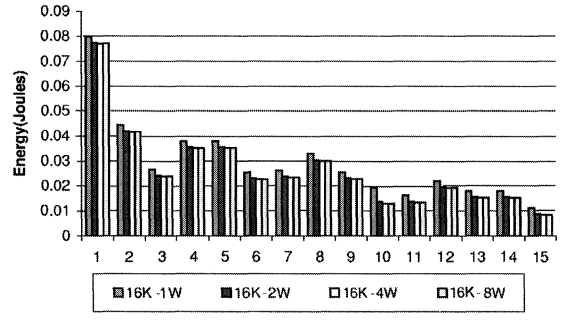

FIGURE 3 Energy consumed in *data cache* by the original mxm code for different hardware optimization configurations when using 16 K data caches with 32 byte blocks.

sub-banks is increased from two to four. It must be observed that the savings are not linear as one may expect. This is because the energy cost of the tag arrays still remain the same independent of the degree of sub-banking. Further, a small amount of additional energy results from the more complex decoding as number of sub-banks increase.

We also varied the number of block buffers used to cache the previously accessed cache lines. We find that adding a single block buffer reduced the energy by up to 50%. This reduction is achieved by capturing the locality of the buffered cache line, thereby avoiding accesses to the entire data array. However, access patterns in many applications can be regular and repeating across a varied number of different cache blocks. In order to capture this effect, we varied the number of block buffers to two, four, and eight as well. We did not increase the number of block buffers beyond eight as the increased cost of accessing larger block buffers started to dominate any energy gains that would be achieved by the additional block buffers. Also, increasing the number of buffers further could create performance problems. We observed that, for the mxm benchmark, an additional 17% (as compared to a single buffer) energy saving can be achieved using four buffers.

We also found that using a combination of eight block buffers and four sub-banks, the energy consumed in 4 K (16 K) data cache could be reduced on an average by 88% (89%). Thus, such hardware techniques can reduce the energy consumed by processors with on-chip caches. However, if we consider the entire memory system including the off-chip memory energy consumption, the energy savings from these techniques amount to only 4% (15%) when using a 4 K (16 K) data cache. Thus, it may be necessary to investigate optimizations at the software level to supplement these optimizations.

Software Optimizations

Figures 4 and 5 show the total energy consumption of the memory system after applying the

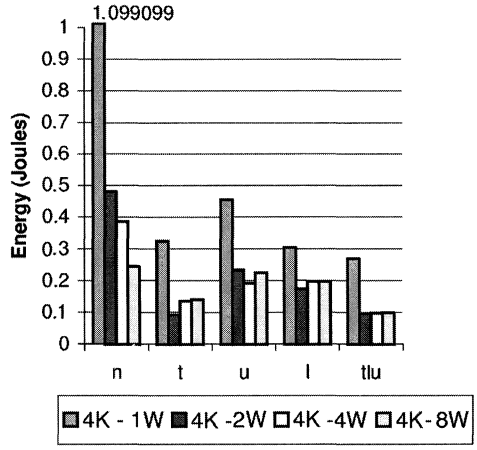

FIGURE 4 Energy consumed in the *entire memory system* by optimized mxm codes when using a 4 K data cache (no block buffers and sub-banking) with 32 byte blocks and varying associativities.

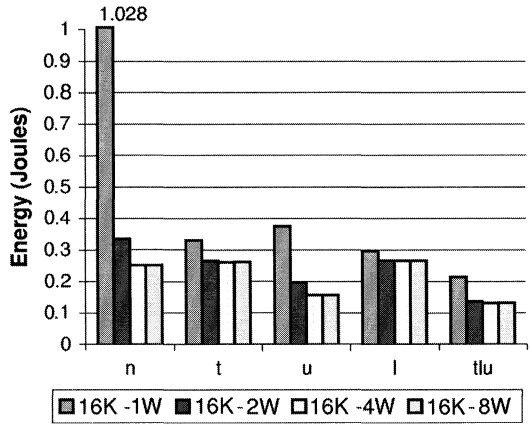

FIGURE 5 Energy consumed in the *entire memory system* by the optimized mxm codes when using a 16 K data cache (no block buffers and sub-banking) with 32 byte blocks and varying associativities.

compiler optimizations on 4 K and 16 K data caches with *no* block buffering or sub-banking. In these and the following figures in this paper, `t`, `l`, `u` and `tlu` denote pure tiling, pure loop transformations, pure unrolling, and the concurrent application of all these three optimizations, respectively. We find that the maximum energy gains of these optimizations are observed when using data caches with lower associativities (1-way and 2-way) and the entire memory energy reductions using a 4 K (16 K) direct mapped data cache range from 58–75% (63–79%) for the different optimizations. As the cache associativities increase, the influence of conflict misses reduces significantly, thereby, downplaying the importance of these locality optimizations. In fact, the energy could increase in such cases as a result of applying the tiling and loop transformation when they are accompanied by scalar replacement. It should be noted that the optimized versions results in a larger energy consumption when using a 16 K cache as compared to using 4 K cache. This is because they exhibit good locality with a 4 K cache already and the incremental locality improvement obtained using a 16 K cache is overshadowed by the increased per-access cost of the larger cache. On the other hand, the unoptimized code takes advantage of larger cache size as its locality is very poor. Further, we find that the combination of all the three optimizations when applied together achieves a maximum of 75% energy saving for 4 K cache configurations investigated.

Figures 6 and 7 show the energy consumed in the data caches after applying the compiler optimizations on 4 K and 16 K data caches with *no* block buffering or sub-banking. We find that the tiling and loop transformations, generally, increase the cache energy consumption due to the

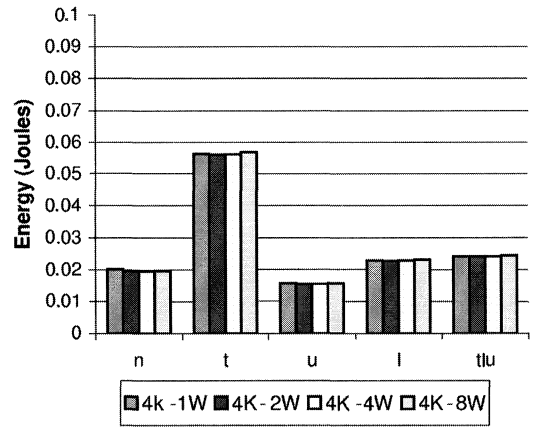

FIGURE 6 Energy consumed by the optimized mxm codes in 4 K *data cache* (no block buffers and sub-banking) with 32 byte blocks and varying associativities.

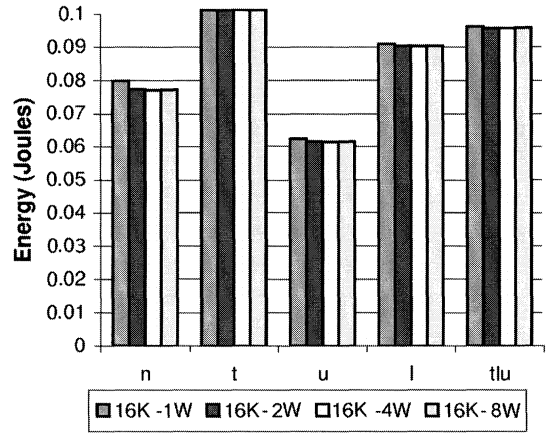

FIGURE 7 Energy consumed by optimized mxm codes in 16 K *data cache* (no block buffers and sub-banking) with 32 byte blocks and varying associativities.

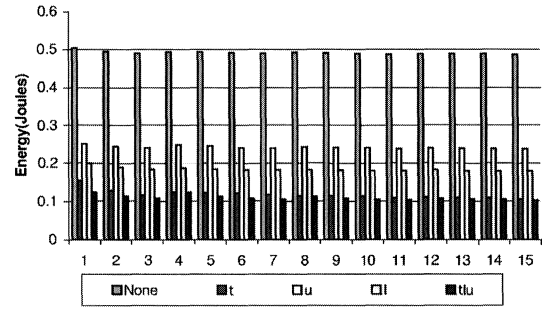

FIGURE 8 Energy consumed in the *entire memory system* by the optimized mxm codes with varying hardware and software optimizations when using a 4 K, 2-way data cache with 32 byte blocks.

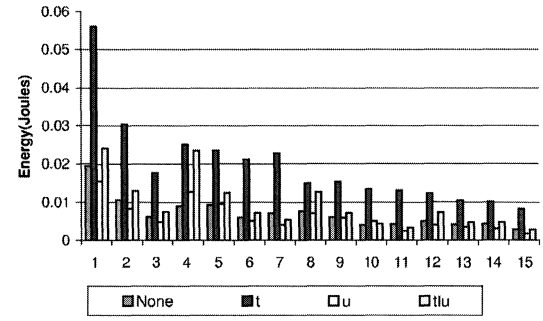

FIGURE 9 Energy consumed by the optimized mxm codes for different hardware and software optimizations in a 4 K 2-way associative *data cache* with 32 byte blocks.

additional memory references introduced by most probably scalar replacement. This effect is pretty dominant in the tiling optimization that results in a 186% increase in energy for 4 K data caches on an average. However, this effect is overshadowed by the drastic reduction in the number of more energy consuming off-chip accesses to the main memory (see Figs. 4 and 5). We also observe that the unrolling optimization is suitable for reducing both the cache energy and the overall energy in the mxm code.

Combined Optimizations

Next, we evaluated the interaction of the hardware and software optimizations. Figure 8 shows the memory system energy consumption when a combination of different software and hardware optimizations is applied. The corresponding variation in on-chip data cache energy consumption for these optimizations is given in Figure 9. It can be observed from Figure 8 that tiling performs the best across different 4 K cache configurations among the three individual compiler optimizations applied. Since, we observed earlier that tiling increases the cache energy consumption dramatically, sub-banking and block buffering are of particular importance here. For the tiled code, moving from a base data cache configuration to one with eight block buffers and four sub-banks reduces the overall memory system energy by around 10%. Thus, *it is important to use a combination of hardware and software optimizations in designing an energy-efficient system.*

Further, we observed that the linear loop transformed codes exploited the block buffers better than the original code and other optimizations. For example, when using two (eight) block buffers in a 4 K 2-way cache, the block buffer hit rate was 69% (82%) as compared to the 55% (72%) for the unoptimized mxm code. Thus, it is also important to choose the software optimizations such that they provide the maximum benefits from the available hardware optimizations. So far, we have presented the results only for the mxm code to discuss the influence and interaction of the software and hardware energy optimization techniques. Figure 10 captures this influence for all

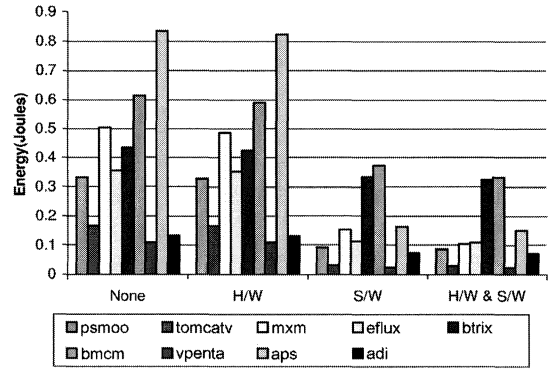

FIGURE 10 Energy consumed when using a 4K 2 way data cache with a combination of hardware and software optimizations. None refers to the base hardware and original code; S/W refers to base hardware and optimized code generated when all optimizations are enabled; H/W refers to data cache with eight block buffers and four sub-banks, and original code; HW/SW refers to data cache with eight block buffers and four sub-banks and optimized code generated when all optimizations are enabled.

the studied benchmarks. These results show that there is a significant potential for reducing energy through a proper combination of hardware and software optimization techniques. The pure hardware optimizations (eight block buffers and four sub-banks) provided up to 4% energy savings, with an average saving of 2% across all benchmarks. In contrast, the pure software optimization approach (`tlu`), provided at least a 23% energy saving, with an average of 62%. Further, a combination of hardware and software optimizations provides an average of 64% energy saving. *It is thus observed that the compiler optimizations provide most of the savings in the memory system energy consumption, while hardware optimization can be critical if one focuses on on-chip cache energy consumption.*

A COMPARATIVE EVALUATION OF ENERGY DUE TO INSTRUCTION AND DATA ACCESSES

Most of the compiler optimizations are oriented toward performance by improving data locality, while instruction sequences generally become more complex. Also, static and dynamic code size can become larger as a result of these optimizations. In this section, we look at the influence of compiler optimizations on the memory system energy considering the overall impact of both instruction and data accesses.

Experimental Strategy

The experimental methodology used in this section is as follows. First, we selected four *motion estimation codes* [26] written in C to conduct experiments as these codes are representative of many embedded image processing applications. A 144×176 frame size and a 16×16 block size were used in all experiments. The goal of motion estimation is to search (within a predefined search area called *reference frame*) for a block that best matches, according to a certain criterion, a given block in the current image frame [25]. The displacement between the coordinates of the block in the current frame and the matched block in the reference frame is called a *motion vector*. Brief descriptions of these four codes are as follows:

Full Search

In full search block-matching motion estimation, each reference macro-block is compared to all candidate macro-blocks in the search area to determine the best match [6]. It is able to find the best matched block, but requires a significant amount of computation. This is the most data-intensive version of the codes we used.

3Step-logarithmic Search

In a logarithmic search motion estimation, the search is accomplished hierarchically in $v = \lceil \log_2 d_m \rceil$ steps, where the search region spans d_m pixels in each direction [25]. During each step $s (0 \leq s \leq v-1)$, a partial motion vector is determined by comparing the total absolute differences evaluated at exactly nine displacement vectors with a local step size of 2^{v-s-1}. The final motion vector can be found as the sum of the partial motion vectors.

Hierarchical Motion Estimation

The hierarchical block matching algorithm is based on the image pyramid [17]. The next level of the original picture can be formed by low-pass filtering of the original image and sub-sampling the resultant picture by two in each dimension. In the hierarchical method, a motion vector at the highest level is calculated first. Then, the search area is displaced by this motion vector and the procedure is repeated up to the lowest level of hierarchy. The final motion vector is the sum of all motion vectors found in all stages of the hierarchy.

Parallel Hierarchical One-dimensional Search (PHODS)

In this algorithm, instead of finding the two-dimensional motion vector directly, two one-dimensional displacements are found in parallel on the two axes independently within the search area [5].

First, the motion estimation codes are compiled using the MIPSpro compiler without any high-level optimizations. However, the compiler performs a number of low-level (back-end) optimizations, that include common subexpression elimination, algebraic simplification, copy propagation, constant propagation, loop-invariant optimization, and scalar replacement. In addition to these optimizations, MIPSpro also optimizes references and definitions for external variables and performs software pipelining [14]. We refer to this version as `base`. Note that the base case uses almost all major state-of-the-art *low-level optimizations* that can be found in a modern optimizing compiler.

The memory reference pattern of the resulting base codes is obtained from the execution trace and fed to the configurable memory system simulator mentioned in Experimental Strategy Section. The results obtained from the memory simulator are then used to evaluate the memory system energy.

Figures 11 and 13 show the energy expended in the memory system during data and instruction accesses, respectively, for the base version (Note

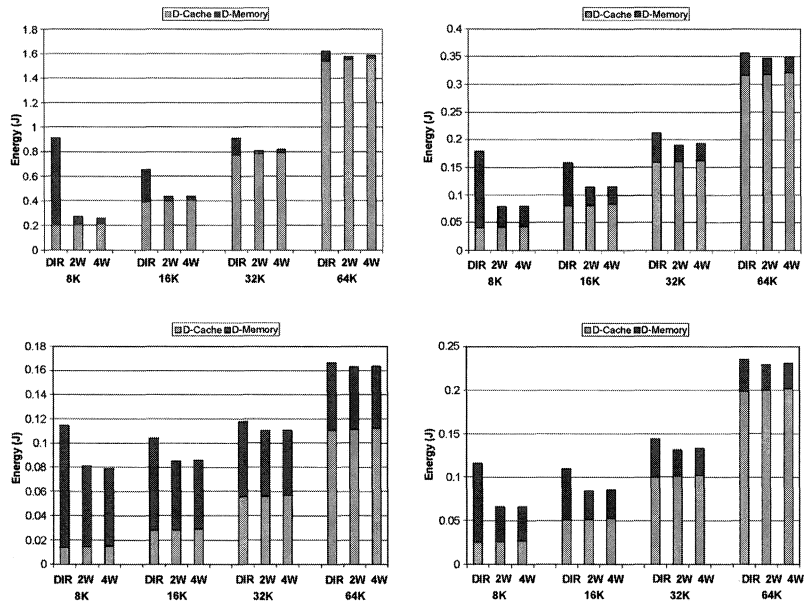

FIGURE 11 Energy (J) consumption (base version) of data accesses for different cache configurations with varying cache sizes (from left to right and top to bottom, `full_search`, `3step_log`, `hier`, and `parallel_hier`). The cache block size is 32 B for all configurations and sizes.

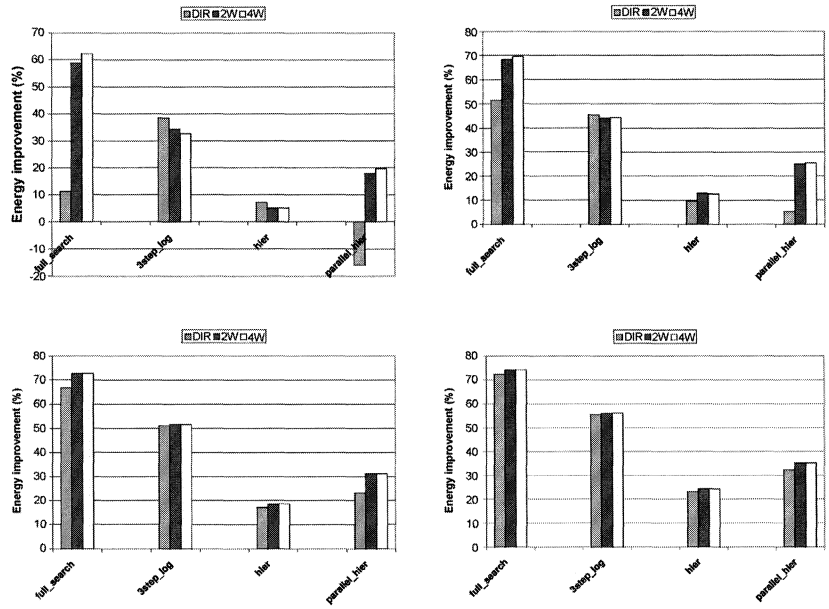

FIGURE 12 Energy reduction (%) due to high-level compiler optimizations for data accesses using different cache configurations (from left to right and top to bottom, cache size of 8 KB, 16 KB, 32 KB, and 64 KB). The cache block size is 32 B for all configurations and sizes.

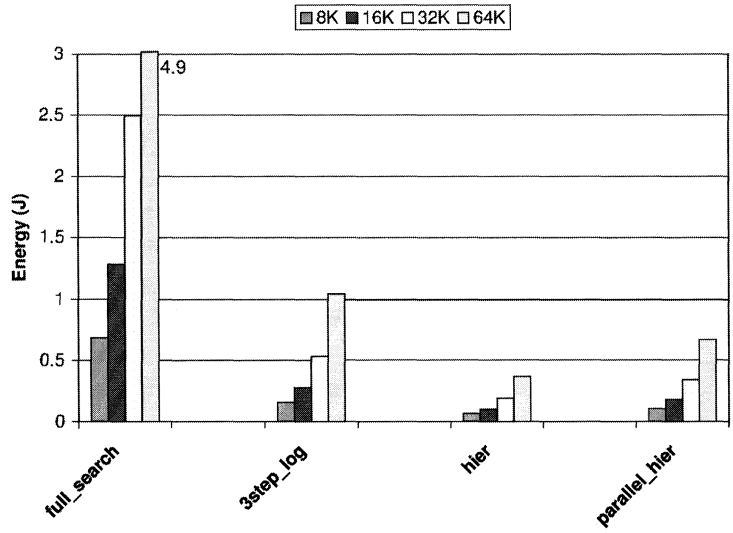

FIGURE 13 Energy (J) consumption (base version) due to instruction accesses for two-way set-associative caches. The cache block size is 32 B for all sizes.

that the y-axes of different graphs are not the same). Among the different algorithms employed to perform the motion estimation, the most data-intensive full search code consumes about 8 times more energy for data accesses than the most energy-efficient PHODS algorithm when using an 8 K direct-mapped data cache. *This shows the importance of choosing appropriate algorithms for*

energy savings. Further, we observe that, for the direct-mapped data caches, the energy expended during data accesses reduces when cache size is increased from 8 KB to 16 KB. But, this trend changes with further increase in cache size. This behavior is due to the significant reduction in cache misses when cache size increases from 8 KB to 16 KB resulting in fewer energy-expensive memory accesses. However, for cache sizes larger than 16 KB, the increased per-access cache energy cost (due to a larger capacitive load) starts to dominate any benefits from fewer cache misses.

It was also observed that beyond an instruction cache size of 8 KB, most of the instruction accesses are captured in the cache. Thus, the number of instruction cache misses is small and most of the instruction related energy is consumed in accessing the instruction cache. Further, *it is observed that the energy cost for instruction accesses is comparable to the energy consumed by data accesses for most configurations. This observation is important as most of the state-of-the-art compiler optimizations currently target only improving data accesses.*

Influence of Optimizations

Next, we attempt to apply linear loop transformations, loop unrolling and loop tiling to the motion estimation codes. In optimizing the motion estimation codes, the compiler could not find any opportunities to apply tiling due to imperfectly-nested nature of the loops in these codes. In two of the codes, however, it successfully applied loop unrolling with an unroll factor of 5 and 6. When we analyze the resulting optimized C codes, we also observe that in all of them, there is an expansion in static code size as compared to the original. This is mainly due to loop unrolling and scalar replacement exercised by the compiler to improve cache and register performance.

Figure 12 shows the change in energy consumption due to data accesses after applying the high-level optimizations. It is observed that the energy reduction is most significant for the full search algorithm that is most data-intensive. This reduction is due to the significant decrease in number of data accesses as a result of improved locality. For example, scalar replacement converts memory references to register accesses. However, this also leads to an increase in dynamic instruction count. We can also see from Figure 12 that, except for one case, high-level compiler optimizations improve the data energy consumption for all motion estimation codes in all configurations. The average data energy reduction over all studied cache sizes is 30.9% for direct-mapped caches, 39.4% for 2-way caches and 39.8% for 4-way caches. Our experiments also show that in `hier` and `parallel_hier`, after the optimizations, there is an increase in the number of conflict misses (as we do not use array padding [15]). In particular, with `parallel_hier`, when the cache size is very small and cache is direct-mapped, these conflict misses offset any benefits that would otherwise be obtained from improved data locality, thereby degrading the performance from the energy perspective. Increasing the associativity eliminates this conflict miss problem.

It can be observed from Figure 14 that the energy consumed by instruction accesses increases on an average by 466%, 30% and 32% for the `3step_log`, `hier` and `parallel_hier` optimized codes, respectively. The main reason for this increase is the aggressive use of scalar replacement in these codes. While this optimization helps caches and registers to exploit temporal data locality, the use of scalar replacement in the inner loops of a given nest structure leads to significant increase in the dynamic instruction count. For example, in the optimized version of `hier`, dynamic instruction count increased to 62 million from 46 million. In contrast, the energy consumed by instruction accesses for `full_search` decreases by 13%. The data access pattern for `full_search` is more regular as compared to the other algorithms. Consequently, the MIPSpro optimizer was less aggressive with scalar replacement. Further, the application of loop unrolling on `full_search` reduced the number of branch instructions.

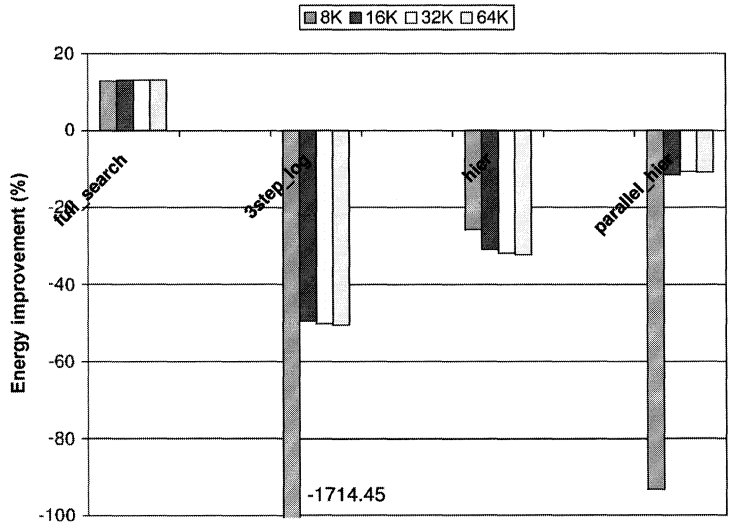

FIGURE 14 Energy reduction (%) for instruction accesses using two-way set-associative caches when high-level compiler optimizations are applied. The cache block size is 32 B for all sizes.

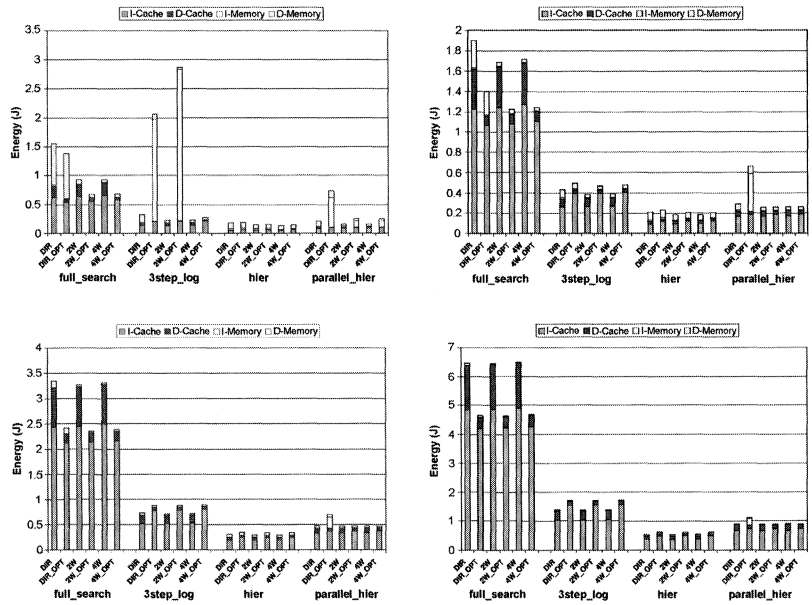

FIGURE 15 Energy consumption breakdown for different cache configuration without and with compiler optimizations (from left to right and from top to bottom, cache sizes of 8 KB, 16 KB, 32 KB, and 64 KB). The cache block size is 32 B for all configurations and sizes.

Figure 15 shows the overall impact of the optimizations considering both the instruction and data accesses. It is observed that the optimizations decrease the energy consumption by 26% for `full_search` on the average. However, due to the detrimental impact on energy consumed by instruction accesses, the overall energy consumption increased by approximately

153%, 11% and 43% for `3step_log`, `hier` and `parallel_hier` respectively.

CONCLUSIONS

The goal of the first part of this study is to investigate the interaction and influence of hardware and software optimizations on the data cache and off-chip memory energies due to data accesses. To achieve this goal, we select two effective cache optimizations and three widely used compiler optimizations and experiment with nine multi-dimensional array codes. We first examine pure hardware optimizations and find that the energy savings from these techniques amount to 4% (15%) when using a 4 K (16 K) data cache. Next, we apply pure software optimizations and find that up to 79% energy gains can be obtained. Further, a combination of the hardware and software optimizations reduces the memory system energy up to 88%. Our results show that the compiler optimizations provide a significantly higher energy savings as opposed to those gained using the pure hardware optimizations considered. However, a closer observation reveals that hardware optimization become more critical for on-chip cache energy reduction when executing optimized codes.

In the second part of the study, we evaluate the effect of high-level compiler optimizations on the entire memory system energy considering both instruction and data accesses. The results show that these optimizations are very effective in minimizing the energy consumed due to data accesses. However, they tend to increase the energy consumed due to instruction accesses significantly for the studied codes. These results indicate that further research is needed in optimizing the locality of instruction and data accesses simultaneously. In particular, we expect the techniques that integrate scalar replacement with locality optimizations (*e.g.*, loop unrolling, tiling, *etc.*) to become more important in design and implementation of future energy-efficient system.

References

[1] Albera, G. and Bahar, R. I. (1998). Power and performance tradeoffs using various cache configurations. In: *Proc. Power Driven Micro-Architecture Workshop in Conjunction with ISCA'98*.

[2] Blume, W., Doallo, R., Eigenmann, J. G. R., Hoeflinger, J., Lawrence, T., Lee, J., Padua, D., Paek, Y., Pottenger, B., Rauchwerger, L. and Tu, P. (1996). Advanced program restructuring for high-performance computers with polaris. *IEEE Computer*, pp. 78–82.

[3] Catthoor, F., Wuytack, S., Greef, E. D., Balasa, F., Nachtergaele, L. and Vandecappelle, A. (1998). *Custom Memory Management Methodology – Exploration of Memory Organization for Embedded Multimedia System Design*. Kluwer Academic Pub.

[4] Chandrakasan, A. and Brodersen, R. (1995). *Low Power Digital CMOS Design*. Kluwer Academic Publishers.

[5] Chen, L.-G., Chen, W.-T., Jehng, Y.-S. and Church, C.-T., An efficient parallel motion estimation algorithm for digital image processing. *IEEE Trans. Circuits and Systems for Video Tech.*, **1**(4), 378–385, Dec., 1991.

[6] Do, V. L. and Yun, K. Y. (1998). A low-power VLSI architecture for full-search block-matching motion estimation. *IEEE Trans. Circuits and Systems for Video Tech.*, **8**(4), 393–398.

[7] Kin, J. *et al.*, The filter cache: An energy efficient memory structure. In: *Proc. of MICRO'97*, December, 1997.

[8] Singh, D. *et al.*, Power conscious CAD tools and methodologies: a perspective. In *Proc. of the IEEE*, April, 1995.

[9] Ghose, K. and Kamble, M. B. (1999). Reducing power in superscalar processor caches using subbanking, multiple line buffers and bit-line segmentation. In: *Proc. International Symposium on Low Power Electronics and Design*, pp. 70–75.

[10] Irigoin, F. and Triolet, R., Super-node partitioning. In: *Proc. 15th Annual ACM Symp. Principles of Programming Languages*, pp. 319–329, San Diego, CA, January, 1998.

[11] Kamble, M. B. and Ghose, K. (1997). Analytical energy dissipation models for low power caches. In: *Proc. International Symposium on Low Power Electronics and Design*, pp. 143–148.

[12] Kamble, M. B. and Ghose, K. (1997). Energy-efficiency of VLSI caches: A comparative study. In: *Proc. of International Conference on VLSI Design*, pp. 261–267.

[13] Kandemir, M., Choudhary, A., Ramanujam, J. and Banerjee, P., Improving locality using loop and data transformations in an integrated framework. In: *Proc. MICRO-31*, Dallas, TX, December, 1998.

[14] Muchnick, S. S. (1997). *Advanced Compiler Design Implementation*. Morgan Kaufmann Pub., San Francisco, California.

[15] Rivera, G. and Tseng, C.-W., Data transformations for eliminating conflict misses. In: *ACM SIGPLAN Conf. Programming Language Design and Implementation*, pp. 38–49, June, 1998.

[16] Roy, K. and Johnson, M. C., Software design for low power. In: *Low Power Design in Deep Sub-micron Electronics*, pp. 433–459. Kluwer Academic Press, October, 1996, Eds. Mermet, J. and Nebel, W.

[17] Seferidis, V. and Ghanbari, M. (1992). Hierarchical motion estimation using texture analysis. In: 1992 *Proc. Int'l Conf. Image Processing and Its Applications*, pp. 61–64.

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

[19] Su, C.-L. and Despain, A. M. (1995). Cache design trade-offs for power and performance optimization: a case study. In: *Proc. International Symposium on Low Power Electronics and Design*, pp. 63–68.
[20] Wilton, S. and Jouppi, N. (1994). An enhanced access and cycle time model for on-chip caches. *Technical Report 93/5*, DEC WRL Research report.
[21] Wolf, M. and Lam, M., A data locality optimizing algorithm. In: *Proc. ACM SIGPLAN 91 Conf. on Programming Language Design and Implementation*, pp. 30–44, June, 1991.
[22] Wolf, M., Maydan, D. and Chen, D., Combining loop transformations considering caches and scheduling. In: *Proc. MICRO-29*, pp. 274–286, Paris, France, December, 1996.
[23] Wolfe, M. (1996). *High Performance Compilers for Parallel Computing*. Addison-Wesley Publishing Company.
[24] Ye, W. (1999). Architectural level power estimation and experimentation. Technical report, *Ph.D. Thesis, Comp. Sci. and Eng.*, The Pennsylvania State University.
[25] Yeo, H. and Hu, Y. H., A modular high-throughput architecture for logarithmic search block-matching motion estimation. *IEEE Trans. Circuits and Systems for Video Tech.*, **8**(3), 299–315, June, 1998.
[26] Zervas, N. D., Masselos, K. and Goutis, C. E. (1998). Code transformations for embedded multimedia applications: Impact on power and performance. In: *Proc. ISCA Power Driven Microarchitecture Workshop*.
[27] Esakkimuthu, G., Vijaykrishnan, N., Kandemir, M. and Irwin, M. J. (2000). Memory system energy influence of hardware–software optimizations. In: *Proc. International Symposium on Low Power Electronics and Design*, pp. 244–246.

Authors' Biographies

Esakkimuthu N Geethanjali is currently working on her Masters in Electrical Engineering at Pennsylvania state University, State College. She received her B.E. in Electronics and Communications from University of Mysore, India in the year 1997. Her research interests include low power VLSI Design, low power system design and Computer Architecture.

H. S. Kim received her B.S. of Computer Science (1994) from Kyunpook National University, Taegu, Korea and M.S. of Computer Science (1996) from Pohang Instutite of Science and Technology, Pohang, Korea. She is pursuing her Ph.D. at The Pennsylvania State University. Her research interests include low power computer architectures, hardware and software interaction in system design, and low power caches. She is a student member of the IEEE and the ACM.

Mahmut Kandemir received his Ph.D. from Syracuse University, Syracuse, NY in Electrical Engineering and Computer Science, in 1999, M.S. and B.S. from Istanbul Technical University, Istanbul, Turkey in Control and Computer Engineering in 1992 and 1988, respectively. He has been an assistant professor in the Computer Science and Engineering Department at the Pennsylvania State University since August, 1999. His main research interests are optimizing compilers, I/O intensive applications, and power-aware computing. He is a member of the IEEE and the ACM.

Vijaykrishnan Narayanan received his Ph.D. in computer science and engineering from the University of South Florida in July, 1998 and his B.E. from the University of Madras, India, in 1993. He has been an assistant professor of computer science and engineering at Pennsylvania State University since August, 1998. His current research interests are in the areas of Java implementation and performance issues, mobile system design, energy-efficient software and architectures, VLSI systems and computer architecture.

Mary Jane Irwin received the M.S. (1975) and Ph.D. (1977) degrees in computer science from The University of Illinois, Urbana-Champaign. She is a Distinguished Professor of Computer Science and Engineering at The Pennsylvania State University. Dr. Irwin has been on the faculty at Penn State since 1977, and served a three term as department head of the then Department of Computer Science (1991–1993). Her current research and teaching interests include computer architecture, the design of application specific VLSI processors, high speed computer arithmetic, and electronic design automation. For her research contributions she was named Fellow of the IEEE in 1995 and Fellow of ACM in 1996. She is a member of both ACM and the IEEE Computer Society and is an elected member of the Board of Directors of the Computing Research Association (CRA). Dr. Irwin served as the cochair of CRA's Committee on the Status of Women in Computer Science and Engineering (CRA-W) and currently serves as the chair of CRA-W's awards subcommittee. She was General Chair of DAC'99 (the 1999 Design Automation Conference).