Data Cache Energy Minimizations Through Programmable Tag Size Matching to the Applications*

Peter Petrov and Alex Orailoglu

Computer Science & Engineering Department
University of California, San Diego
(ppetrov,alex)@cs.ucsd.edu

ABSTRACT

An application-specific customization methodology for minimizing the energy dissipation in the data cache of embedded processors is presented in this paper. The data cache subsystem is one of the most power consuming microarchitectural parts of embedded processors. We target in this work particularly the data cache tag operations and show how an exceedingly small number of tag bits, if any, are needed to compute the miss/hit behavior for the vast majority of load/store instructions executed within application loops. The energy needed to perform the tag reads and comparisons can be thus dramatically reduced. We follow up this conceptual enhancement with a presentation of an efficient, reprogrammable implementation that utilizes application-specific information to apply the suggested energy minimization approach. The conducted experimental results confirm the expected significant decrease of energy dissipation for a set of important numerical kernels.

1. INTRODUCTION

An ever increasing and significant portion of the consumer electronic market nowadays is dominated by embedded systems. A large part of the functionality of such systems is typically implemented on a set of embedded processors. Major benefits of using embedded processors include improved time-to-market, flexible system implementation, and low-cost system design. The embedded processor cores impose though in turn significant penalties in terms of performance and power, mainly due to their generality.

With the advent of the mobile electronic system, such as cellphones, PDAs, and laptop computers, power consumption minimization is becoming one of the major quality requirements, since less power consumption of the product translates to longer battery life. Consequently, overall product quality is highly dependent on techniques for minimizing system power consumption. These techniques can be applied on various design abstraction levels, from circuit level to system architecture.

*This work is supported by an IBM Graduate Fellowship and NSF Grant 0082325.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ISSN'01, October 1-3, 2001, Montréal, Québec, Canada.
Copyright 2001 ACM 1-58113-418-5/01/0010 ...\$5.00.

Circuit-level power minimization techniques have been the dominant approach in designing energy efficient designs so far [1, 2]. However, in recent years, architecture-level approaches have attained popularity due to their ability to eliminate redundancies on a higher, microarchitectural level, thus resulting in even larger power optimizations [3, 4]. In [5], a small and energy efficient L0 data cache has been introduced in order to reduce the power consumption of the memory hierarchy. The price paid is an increased miss rate and longer access time. A power optimization technique applied during behavioral synthesis for memory intensive applications has been presented in [6]. The behavior of the memory access patterns is utilized to minimize the number of transitions on the address bus and decoder, thus reducing power consumption. In [4] an L0 instruction cache has been proposed with run-time techniques for accommodating only the frequently executed basic blocks. The small size of this cache translates directly to power consumption reductions. The speculative execution in modern high-end processors results in high instruction execution overhead. In [3] a technique for speculation control and pipeline gating has been presented for energy reduction in speculative processors. A new energy estimation framework for microprocessors has been proposed recently in [7]. The simulation environment employs a transition based power model and quickly achieves very precise power estimations.

In this paper, we propose a technique for application-specific customization of the data cache (D-cache) subsystem of embedded processor cores, one of the most power devouring components of the processor architecture. The proposed technique is particularly suitable for applications that contain data-intensive, numerical loops, a trait shared by a variety of DSP applications. We describe an architecture, capable of utilizing application-specific information in a microarchitecturally reprogrammable way. The technique enables re-customization in a post-manufacturing fashion, thus effectively covering a large class of real-life applications with no need for spinning new silicon.

Application-specific customization of embedded processor architectures is a technique that transfers application information to the processor architecture [8]. In this case, the microarchitecture performs informed decisions as to how to handle various architecture-specific actions. Fundamentally, this approach extends the communication link between compiler and processor architecture by transferring application information directly to the microarchitecture, while keeping the traditional compiler techniques unaffected. The static analysis information transfer is accomplished by utilizing a reprogrammable hardware implementation. The architecture we propose allows application changes to be applied in field by loading the new application information, in a manner similar to program reloading.

The proposed methodology utilizes information about the placement of the data being accessed within the application loops and more specifically the possible D-Cache conflicts and the minimal number

Figure 1: DM cache organization

of tag bits needed to identify these conflicts. The tag operations associated to the D-cache are designed for a worst case scenario and they utilize the entire effective address. These operations are very expensive in terms of power and usually carry a large amount of redundancy since conflicting references are frequently close to each other in the address space and a large part of their tags is identical. We show how the minimal necessary number of tag bits for identifying cache conflicts can be inferred from the data layout. In the extreme case of an application loop with a dataset that fits squarely in the D-cache, there would be no cache conflicts; hence no tag operations would be required at all. Only the cold misses when entering the loop require special attention. To handle the cold misses, the valid bits associated to the cache lines could be used. An efficient way of invalidating the cache lines corresponding to the loop prior to its execution is proposed, leading to a power efficient solution for handling the cold misses and enabling the application of the tag minimization framework that we propose.

We complement our methodology with a discussion of an efficient hardware implementation for the proposed D-cache customization. Not only is the hardware solution efficient, but is also reprogrammable, thus providing the flexibility to re-customize the embedded processor core in a post-manufacturing fashion. Consequently, the described hardware implementation constitutes a unified microarchitectural solution, capable of handling a large set of important applications through in-field re-customization, thus maintaining the market benefits of high-volume productions.

2. MOTIVATION

The D-cache is used to move the data closer to the processor core, so that the time needed to load data from memory is minimized and the performance gap between processor core and memory alleviated. A typical direct-mapped cache organization is shown in Figure 1. In this paper we consider only direct-mapped caches, but the approach can be easily extended to set-associative organizations. The referenced effective address is separated into *block index*, *cache index*, and *tag*. The block index is used to address a word within a cache line, while the cache index is used to address the cache line; the tag field checks whether there is a conflict with a memory location with the same cache index. The tag field of each cache line is stored in a separate tag memory array. Each time an access is performed to the D-cache, the tag associated to the cache line is read and compared to the tag of the effective address being referred. However, if the referred locations are close in the address space, a large part of these tag fields is identical. Consequently, a large amount of power is spent in reading, comparing and writing unnecessarily large tags.

In the domain of computationally intensive applications, such as DSP processing, application loops work on a set of data arrays. Performance considerations force manual optimization of these loops frequently. The loops rarely have any additional memory references, such as spill and fill code. The only data being addressed in the memory space is the actual data on which the loops operate. Figure 2 shows

```
for(i1=0;i1<64;i1++)
  for(i2=0;i2<64;i2++)
    for(i3=0;i3<64;i3++)
      C[i1][i3]+=A[i1][i2]*B[i2][i3];
```

Figure 2: Matrix multiplication code

a matrix multiplication loop. The loop operates on three matrices and the only accesses to data memory consist of the array references.

Figures 3a and 3b depict the memory layout of the three matrices *A*, *B*, and *C* from the example. The data memory space is divided into regions that correspond to one D-cache size and for each of these memory regions the tag part of the address is a constant. We denote these memory regions as *0-tag regions*. The left part of Figure 3 shows a configuration in which the data set of the example resides within a single 0-tag region. It is evident that in such a case there will be no conflicts within the arrays of data in the D-cache and there is no need to perform any tag operations. This is a straightforward consequence of the fact that a tag region has size exactly equal to the D-cache size and the tag part of the address is a constant. Figure 3a depicts a configuration in which the dataset spans two tag regions. In this case there *will* be conflicts in the D-cache; nonetheless, the tag fields of two conflicting addresses will differ on average by an exceedingly small number of bits. If the least significant bit of the tag for a given tag region is 0, then the address tag associated with the subsequent tag region in the memory will differ only in the least significant bit, which will be 1. In this way the tag regions in neighboring pairs for which the address tag differs only by one bit, the least significant bit, can be grouped. Such pairs of 0-tag regions are denoted as *1-tag regions*. The entire memory space is covered by a set of disjoint 1-tag regions.

A straightforward inference that can be drawn from the above observations is that there typically exists a large amount of redundancy in reading the entire tag from the tag array and comparing it to the effective address tag, given the proximity of the dataset references; a large number of identical bits in the tag components is to be expected, consequently. Given that application-specific information about the application loop dataset layout in the data memory is present during program execution, a large part of the tag redundancy can be eliminated, resulting in significant energy savings. The application information can be obtained during compile time and provided to the D-Cache microarchitecture in a reprogrammable way. Furthermore, certain compile-time optimization techniques can be utilized, to ensure that the actual loop dataset layout minimizes the number of required tag bits for identifying D-cache conflicts. For example, if possible, the compiler could try to place the data within a 0-tag region. If the data does not fit, the compiler can try incrementally larger tag regions until the required minimal number of tag bits is identified.

Subsequent sections in the paper present our methodology for eliminating the aforementioned tag redundancy in an application-specific

Figure 3: Memory layout

manner, thus achieving significant energy reductions. Since the flexibility of straightforward incorporation of application code changes is one of the paramount advantages of utilizing embedded processor cores, an efficient reprogrammable implementation that defines a unified architecture capable of capturing application information in-field is presented. This architecture preserves the generality of the embedded processor core in terms of functional programmability, while eliminating the inherent tag redundancy in the cache subsystem.

3. TAG UTILIZATION ANALYSIS

As demonstrated in the previous section, if the dataset of the frequently executed application fragments resides within a 0-tag region in the memory space, then no D-cache conflicts are possible; hence no tag operations are needed. When the loop dataset spans more than one tag region, a few tag bits are needed depending on the number and size of the tag regions the data spans. The structure and the position of the tag regions are determinative for extracting the redundancy in the tag fields. We explain subsequently the formation and structure of the tag regions within the main memory space.

3.1 Tag region formation

To understand the structure of the tag regions, let's consider a D-cache organization and memory address space that requires three bits of tag. This memory space can be divided into eight 0-tag regions with the tag field of the effective address a constant. A generalization of the 0-tag regions can be effected by noticing that all 0-tag regions can be grouped in pairs for which the tag field differs only in the least significant bit; for the first 0-tag region it contains the value of 0, while for the second one, the value is 1. We denote the region formed by the pair of such 0-tag regions as a 1-tag region. If a dataset resides within a 1-tag region, then only the least significant bit from the tag field needs to be used for conflict identification in the D-cache. In a similar vein, the 1-tag regions can be grouped into 2-tag regions. Generally, a $k + 1$ -tag region is formed by a pair of k -tag regions that differ in no more than the $k + 1$ least significant bits of the tag field. All the tag regions are nested within each other; for example a $(k + 1)$ -tag region contains two k -tag regions and each of the k -tag regions contains two $(k - 1)$ -tag regions in turn. The set of all k -tag regions, for any value of k , covers the entire memory space. A k -tag region corresponds to a portion of the memory space with size equal to the size of 2^k D-caches and tags differing only in the least significant k bits. The n -tag region that covers the complete memory corresponds to a region that requires all the tag bits for detecting conflicts. The general-purpose caches operate under the extreme and general case assumption that all the application datasets reside in the n -tag memory region, which corresponds to the entire memory space. Their inability to incorporate application knowledge regarding the refinement of the tag regions within which the application data resides is the fundamental reason for the significant amount of tag operation redundancy.

Based on the above observations, it is evident that if the D-cache architecture incorporates application knowledge of the loop dataset layout, the energy expensive tag reads can be optimized to read only the minimum required number of tag bitlines from the tag SRAM array. At the same time, the compiler can try to ensure that the loop data are placed in such a way that they span the minimal tag region that corresponds to their size.

3.2 Dataset with no D-cache conflicts

If the dataset for the particular application loop has size smaller than the D-cache, then it can be placed within a 0-tag region. This size analysis is performed at compile time and the loop data is placed within a 0-tag region in the memory. In this case, no conflicts in the D-cache for the loop data are possible and hence no tag operations are needed.

Figure 4: Loop dataset placement

Detection solely of the loop data cold cache misses is required for ensuring the correct D-cache operation. Detecting the cold cache misses can be achieved by invalidating the cache content in which the data resides; if the data spans the whole cache, then the cache needs to be invalidated completely. As the cache reuse across loops is negligible, if any, no performance penalty is incurred, or at worst in the case of data reuse between different loops, the insignificant penalty of the cold misses of the second loop data, constitutes the only performance penalty.

Once the cache part that will accommodate the loop data is invalidated, the cold misses are detected naturally through the usage of the valid bits of the cache lines. Noteworthy is that cold misses can occur not only in the first loop iteration but also in subsequent iterations, due to variably traversed control paths in loop iterations. Consequently, the valid bit usage is a natural solution for detecting cold misses within the loop dataset.

3.3 Minimal tag usage for conflicting datasets

If the loop dataset cannot fit within a 0-tag region, then there will be D-cache conflicts and some tag bits will be needed for conflict identification. Since we want to minimize the number of tag bits needed for comparison, a compile time data placement algorithm is suggested in order to place the dataset within the boundaries of a k -tag region with minimal k .

An even further refinement of placing the dataset within the tag regions can be defined by observing that when placing a dataset within the boundaries of a $k + 1$ -tag region, for some of the data arrays only k tag bits might be enough to check for conflicts within the D-cache. Figure 4 depicts an example in which the dataset of the example from the previous section is placed into a 1-tag region and its size does not allow the dataset to fit within a 0-tag region. Since the 1-tag region comprises two 0-tag regions, part of the loop dataset (matrices *A* and *B*) is placed in the first 0-tag region, and another part (the matrix *C*) in the second 0-tag region. As the size of these matrices is identical, one can observe that matrix *C* overlaps in the D-cache with matrix *A*, while matrix *B* from the first 0-tag region does not overlap with any data in the second 0-tag region. Consequently, when accessing matrix *B* no tag bits would be needed for conflict checking, while in accessing *A* and *C*, one tag bit is required for detecting conflicts. This principle can be generalized to larger tag regions. If part of the loop dataset is placed into a k -tag region and the rest of it is placed into the next k -tag region, then for the components in the first k -tag region that do not conflict with the components in the second k -tag region, only k tag bits would be sufficient for D-cache conflict identification.

We can see that there are two different scenarios for the number of tag bits needed for the D-cache operations within a loop. If the entire dataset resides in a single k -tag region, with overlap possibilities for all data components, then only k tag bits are needed for this loop. If part of the data is placed in the next k -tag region as described in

the previous paragraph, then for parts of the data components k tag bits would be enough, while for the remainder, $k + 1$ tag bits will be required.

The proposed methodology for energy reduction in the D-cache subsystem consists of two basic steps. The first step is the compile-time support that was presented in this section. This support includes placing the loop dataset within a tag region, or spanning more than one tag region by overlapping only some of the data arrays. During compile time, the minimal number of tag bits needed for each loop is determined and provided to the D-cache microarchitecture through the means outlined in the next section, wherein we describe the required hardware support.

4. IMPLEMENTATION

The proposed methodology requires a hardware support that would be able to dynamically enable only the minimum required number of bits from the tag array for the program loops. The hardware requires exact information as to how many tag bits are needed for a particular loop or functions within this loop.

First we present an efficient hardware for manipulating the tag memory array so that only the required minimal number of tag bits are used per application loop. The tag array in the cache subsystem is typically implemented as an SRAM array, possibly divided into multiple banks. The SRAM data array contains horizontal *wordlines* for each tag word and a vertical *bitline* for each bit within the tag word.

A read operation from the SRAM array is performed in the following way. The address decoder selects the wordline to be read from the array. All the bitlines are precharged and if the selected memory cell by the wordline contains logic zero, then the bitlines start to get discharged. Since the discharge is a quite slow process, a sense amplifier is utilized at the end of each bitline. If a small drop of the voltage level is detected, a logic zero is registered. The precharge and discharge of bitlines are the most energy consuming operations with SRAM data arrays [9].

By eliminating most of the bitline precharge and discharge operations, our approach greatly reduces the energy dissipation in the tag SRAM array. This is achieved by gating the bitlines according to the minimal number of tag bits required to check for D-cache conflicts. Only the needed bitlines, if any, are precharged and discharged, thus effectively eliminating the redundant reads. The sense amplifiers for the disabled bitlines are gated as well. Furthermore, the tag comparator cells are gated in order to perform the comparison only on the required tag bits.

The number of tag bits for each loop needs to be determined before entering the loop, so that the appropriate number of bitlines is enabled. Since this number is fixed for the loop, it can be stored in a special control register before entering the loop. Each bit in this special control register directly corresponds to an enable signal of bitline and sense amplifier. The default value of this register specifies that all tag bitlines are enabled. The current value of this register is used to determine the number of bitlines to enable. The only delay imposed by this implementation is the insignificant delay of the gating logic, which roughly corresponds to the delay of a simple *and* gate.

If there are data arrays placed in the next k -tag region with partial overlap with data arrays from the previous k -tag region, then the accesses to these arrays require only k tag bits. An identification mechanism is needed to distinguish the load instructions to these arrays from the others. One possible approach is the incorporation of an additional bit to the load instructions that will indicate whether an additional tag bit is needed. While highly cost effective in terms of hardware support, nonetheless, because of the optimized opcode size in most embedded processors, this approach might be infeasible. An alternative implementation consists of a small table, namely, a *Load Identifica-*

tion Table (LIT). When a load instruction is encountered, the LIT is indexed by the PC of that load instruction and a bit in it specifies whether an additional tag bit is needed. Note that the size of these tables must be limited to a very small number of entries in order to keep the benefits of optimizing the additional tag bit. A reasonable solution would contain at most 8 entries, which compared to the bitline saved from the much larger tag array is negligible.

The proposed implementation is highly cost effective, while inherently reprogrammable. It does not impose a timing constraint to the D-cache organization and can be reprogrammed in a post-manufacturing fashion. The application information is provided by the LIT and by writing values to the special control register for fixing the minimal number of tag bits. This part is performed during compile time, by inserting an instruction for writing the correct value into the register before entering the loop, and an instruction for writing the default value after exiting the loop. The content of the LIT is loaded at the same time as loading the application code into the program memory. Since there might be multiple program loops, multiple LITs might be needed. This can be achieved in a practical way by implementing the LIT as a bigger table and parts of it treated as separate smaller tables. When switching program loops, a switch between these smaller tables needs to be effected. This can be done easily in software by writing a control information to a register that selects the LIT.

Another issue that needs to be addressed is the invalidation of the cache lines before entering a program loop so as to use the valid bits for detecting the cold cache misses for the loop dataset. Prior to entering the loop that is targeted for tag optimization, the data cache content needs to be invalidated. This can be effectively achieved by software in terms of setting a control bit just before entering the loop.

Setting the value for enabling the tag bitlines, switching the LITs, and invalidating the corresponding cache part are the operations that are performed in software. Noteworthy is that all of them are performed outside the loop, thus contributing no additional instructions (and no consequent delay) inside the loop. The lookup in the LIT is performed by hardware only for the *load* instructions; therefore, again no delay is added in executing the loop or accessing the D-cache.

5. EXPERIMENTAL RESULTS

In our experimental study we evaluate the ability of our methodology to reduce energy consumption by minimizing the number of tag bits used in the D-cache. The results demonstrate the proposed approach on 16K and 32K direct-mapped D-caches. Both D-cache configurations contain 4 instructions per cache line. We utilize four numerical computation applications: *Matrix multiplication (mmul)* of matrices with size 64x64; *LU decomposition (lu)* [10] on a matrix with size 64x64; *Extrapolated Jacobi-iterative method (ej)* [10] on a 64x64 grid; and *successive over-relaxation (sor)* [11] on a matrix with size 64x64.

Figures 5 and 6 show execution and power data for the benchmarks on general-purpose 16K and 32K direct-mapped instruction caches. The first row gives the number of D-cache hits. The second row of the tables corresponds to the number of D-cache misses, while the third row presents the energy dissipated in the D-cache in mJ. In order to generate the statistics for the D-cache behavior, we utilize the SimpleScalar [12] simulator. We utilize for the cache configuration power models obtained by using the Cacti tool [13] assuming 0.35um process technology. The total energy dissipation is computed by using the execution statistics from SimpleScalar and the static power model produced by Cacti. The energy for the main memory is based on the data presented in [14] and assumes 4.95nJ per access.

To evaluate the proposed power optimization technique we have identified the application loops and computed the size of the loop dataset and consequently, the k -tag region with minimal k , in which

	mmul	sor	ej	lu
#hits	1,003,006	46,556	563,849	126,884
#misses	45,570	1072	615,151	7,937
energy	2.31	0.102	4.21	0.302

Figure 5: Access and energy (mJ) statistics for 16K DM D-cache

	mmul	sor	ej	lu
tag bits	2	0	3	2
energy	2.06	0.143	4.08	0.271
reduction	0.25	0.021	0.13	0.031
reduction(%)	10.82%	20.59%	3.08%	10.26%

Figure 7: Energy (mJ) statistics for tag optimized 16K D-cache

the dataset can be placed. The subsequent computation of the D-cache access statistics for all the references within the application loops was performed by applying manual assembly level instrumentation of the applications and effecting the corresponding modifications to the simulator. Finally, using power models for the optimized tag array we compute the energy consumed in the D-cache subsystem after optimizing the number of tag bits required for D-cache conflict identification. By utilizing Cacti, we compute the energy dissipated per tag bitline, sense amp, and comparator cell, which allows computation of the energy dissipation for a tag array with only k tag bits enabled. Figures 7 and 8 show the results achieved for the benchmarks.

The first row in these tables shows the minimal number of tag bits required for D-cache conflict identification. The *sor* benchmark requires no tag bits, since it works on a single array that fits within both 16K and 32K D-caches. The second row of these tables shows the absolute amount of energy dissipation (in mJ) for the tag optimized D-caches. The third row corresponds to the absolute energy reduction compared to general-purpose D-cache configurations from Figures 5 and 6, while the fourth row shows the percentage improvement. One can observe that the improvements vary from 3% to above 20%. Of course, the relative energy reduction is a function not only of the number of tag bits utilized, but also of the miss rate. When the cache misses are considerable, the relative improvement is diminished, since the energy dissipated in the main memory due to misses overwhelms the energy saved from removing some of the tag bits.

In case no tag bits are needed whatsoever, which is the case for *sor*, then the tag subsystem is completely turned off, including the tag decoder and wordline selection, thus further decreasing the energy dissipation. Together with the insignificant miss rate, this complete absence of tag bits underlies the significant energy improvement for the *sor* benchmark.

6. CONCLUSION

We have presented an optimization methodology for reducing the energy for the data cache subsystem of embedded processors. The proposed framework consists of compile time support for placing the application loop's dataset into the memory in such a way that the required number of tag bits for accessing the D-cache are minimized. The approach transfers certain application information to the D-cache microarchitecture and dynamically utilizes it to further eliminate the redundancy in the tag operations. An efficient reprogrammable implementation has been proposed for the presented application-specific, power optimization technique. It preserves the fundamental advantage of processor-based implementations of flexibility, design reuse, and high-volume productions.

Power consumption is a crucial quality factor in numerous modern applications. Our experimental results demonstrate the strength of

	mmul	sor	ej	lu
#hits	1,011,904	46,588	808,529	132,930
#misses	36,672	1040	370,471	1,891
energy	3.17	0.143	4.22	0.352

Figure 6: Access and energy (mJ) statistics for 32K DM D-cache

	mmul	sor	ej	lu
tag bits	1	0	2	1
energy	2.79	0.117	3.94	0.352
reduction	0.38	0.026	0.28	0.05
reduction(%)	11.99%	18.18%	6.64%	12.44%

Figure 8: Energy (mJ) statistics for tag optimized 32K D-cache

the proposed approach on a set of real-life applications and suggest the viability of the power minimization technique for a large range of important applications.

7. REFERENCES

- [1] M. B. Kamble and K. Ghose, "Analytical energy dissipation models for low-power caches", in *ISLPED*, pp. 143–148, August 1997.
- [2] K. Ghose and M. B. Kamble, "Reducing power in superscalar processor caches using subbanking, multiple line buffers and bitline segmentation", in *ISLPED*, pp. 70–75, August 1999.
- [3] S. Manne, A. Klauser and D. Grunwald, "Pipeline gating: speculation control for energy reduction", in *25th ISCA*, pp. 132–141, June 1998.
- [4] N. Bellas, I. Hajj and C. Polychronopoulos, "Using dynamic cache management techniques to reduce energy in a high-performance processor", in *ISLPED*, pp. 64–69, August 1999.
- [5] J. Kin, M. Gupta and W. H. Mangione-Smith, "The filter cache: an energy efficient memory structure", in *30th MICRO*, pp. 184–193, April 2001.
- [6] P. R. Panda and N. D. Dutt, "Low-power memory mapping through reducing address bus activity", *IEEE Transactions on VLSI Systems*, vol. 7, pp. 309–320, 1999.
- [7] N. Vijaykrishnan, M. Kandemir, M. J. Irwin, H. S. Kim and W. Ye, "Energy-driven integrated hardware-software optimizations using SimplePower", in *27th ISCA*, pp. 95–106, June 2000.
- [8] P. Petrov and A. Orailoglu, "Towards effective embedded processors in codesigns: customizable partitioned caches", in *9th CODES*, pp. 79–84, April 2001.
- [9] N. Bellas, I. Hajj and C. Polychronopoulos, "A detailed, transistor-level energy model for SRAM-based caches", in *ISCAS*, pp. 198–201, June 1999.
- [10] S. Nakamura, *Applied Numerical Methods with Software*, Prentice-Hall, Englewood Cliffs, N.J., 1991.
- [11] M. E. Wolf and M. S. Lam, "A Data Locality Optimizing Algorithm", in *PLDI*, pp. 30–44, June 1991.
- [12] D. Burger and T. M. Austin, "The SimpleScalar Tool Set, Version 2.0", Technical Report 1342, University of Wisconsin-Madison, CS Department, June 1997.
- [13] G. Reinman and N. Jouppi, "An Integrated Cache Timing and Power Model", Technical report, Western Research Lab, 1999.
- [14] W-T. Shiue and C. Chakrabarti, "Memory exploration for low power, embedded systems", in *36th DAC*, pp. 140–145, June 1999.