Compiler-Managed Partitioned Data Caches for Low Power

Rajiv Ravindran [*]
Java, Compilers, and Tools Laboratory
Hewlett-Packard Company, Cupertino, CA
ravindran@hp.com

Michael Chu and Scott Mahlke
Advanced Computer Architecture Laboratory
University of Michigan - Ann Arbor, MI
{mchu, mahlke}@umich.edu

Abstract

Set-associative caches are traditionally managed using hardware-based lookup and replacement schemes that have high energy overheads. Ideally, the caching strategy should be tailored to the application's memory needs, thus enabling optimal use of this on-chip storage to maximize performance while minimizing power consumption. However, doing this in hardware alone is difficult due to hardware complexity, high power dissipation, overheads of dynamic discovery of application characteristics, and increased likelihood of making locally optimal decisions. The compiler can instead determine the caching strategy by analyzing the application code and providing hints to the hardware. We propose a hardware/software co-managed partitioned cache architecture in which enhanced load/store instructions are used to control fine-grain data placement within a set of cache partitions. In comparison to traditional partitioning techniques, load and store instructions can individually specify the set of partitions for lookup and replacement. This fine grain control can avoid conflicts, thus providing the performance benefits of highly associative caches, while saving energy by eliminating redundant tag and data array accesses. Using four direct-mapped partitions, we eliminated 25% of the tag checks and recorded an average 15% reduction in the energy-delay product compared to a hardware-managed 4-way set-associative cache.

Categories and Subject Descriptors B.3.2 [*Memory Structures*]: [Cache Memories]; C.3 [*Special-purpose and Application-based Systems*]: [Real-time and Embedded Systems]; D.3.4 [*Programming Languages*]: [Code generation, Optimization]

General Terms Algorithms, Design, Experimentation, Performance

Keywords Partitioned cache, hardware/software co-managed cache, instruction-driven cache management, low-power, embedded processor

1. Introduction

Caches have been highly successful in bridging the processor-memory performance gap by providing fast access to frequently used data. They also save power by limiting expensive off-chip accesses. Data caches have proven to be effective as they help to dynamically capture both temporal and spatial locality without software intervention.

The use of caches in embedded domains, however, has been limited due to energy inefficient tag checking and comparison logic [4]. Set-associative caches can achieve a high hit-rate and good performance, but come at the expense of energy overhead. Direct-mapped caches remove much of the logic overhead and thus consume much less power per access, but they incur more misses. Studies show that the on-chip memory subsystem, specifically the instruction and data caches, is the highest contributor to the overall system power. For example, caches consume around 40% of the total processor power in the StrongARM 110 [20] and around 16% for the Alpha 21264 [6] processor. Thus, it is critical to use these on-chip memories efficiently to maintain high performance while operating at low power budgets.

Current techniques to attack this growing memory power problem can be broadly classified into hardware- or software-based solutions. Common hardware solutions include memory banking [10, 24], dynamic voltage/frequency scaling [16], and dynamic cache resizing [22]. Software-based solutions include the use of compiler-managed scratch-pads [4]. Although these techniques have merit, their use and effectiveness has been limited. Hardware-only techniques suffer from the disadvantage of adding complex structures that target a generalized class of applications, which can complicate the design and verification process. In addition, hardware techniques often resort to local program state information, such as the program execution history. This localized view can result in sub-optimal solutions. On the other hand, software-only techniques, in spite of being tuned for each application, tend to make conservative decisions in order to ensure correctness. Moreover, they can be limited to analyzing programs with highly constrained code and memory access behavior, such as array-only code that is indexed through affine functions [15].

Hardware/software co-management can potentially provide the combined benefits of both hardware and software techniques. Software management can help reduce hardware inefficiencies using global knowledge of the program behavior. Hardware assistance can capture dynamic program behavior, and thus aid in making software techniques more aggressive and effective.

In this paper, a hardware/software co-managed partitioned cache architecture is proposed that attempts to bridge the performance and energy gap between direct and set-associative caches. A partitioned cache consisting of multiple smaller direct-mapped partitions with the same combined size as a unified direct or set-associative cache is employed. Management of these partitions is controlled by the compiler using load/store instruction set extensions. Using a whole program knowledge of the data access patterns, the compiler controls cache lookup and data placement by assigning individual load/store instructions to these partitions.

[*] This work was done while the author was a graduate student at the University of Michigan - Ann Arbor

LCTES'07 June 13–16, 2007, San Diego, California, USA.
Copyright © 2007 ACM 978-1-59593-632-5/07/0006. . . $5.00

This software-guided partitioned cache architecture has many advantages. First, a smaller direct-mapped cache is more power efficient than either a unified direct-mapped or a set-associative cache, as the data and tag arrays are smaller. The software decides what partitions are activated, thus eliminating redundant tag/data array accesses to reduce power. Second, by managing the placement of data using memory reference instructions, the compiler can enforce a better replacement policy. For example, data items that are accessed with a high degree of temporal locality can be placed in different partitions to avoid conflicts. In addition, references that are separated in time or whose live-ranges do not intersect can share the same partition. Thus, this data orchestration can help reduce conflict misses. Region-based caches [18] have been proposed, where multiple caches are used to capture heap, global, and stack accesses. But, unlike their approach, partitioned caches provide much finer grain data management and control. Instruction-level management generalizes to all types of data for all classes of applications, including heap dominated ones, where distinct heap objects can be placed in different partitions. Thus, through compiler-controlled management, the partitioned cache architecture can achieve the performance of a set-associative cache, while having the energy signature of a direct-mapped cache.

2. Background

In this section, we motivate the need for hardware/software co-managed caches and discuss how software can exploit partitioned caches for finer grained cache management.

Hardware/Software Co-Managed Caches: In a traditional cache design, the hardware is used to determine replacement and allocation policies. A standard hardware cache controller has two main responsibilities: 1) checking if the referenced data is present in the cache, and 2) on a miss, deciding where in the cache to allocate the requested data. Performing checks in hardware allows for fast and efficient access of the referenced data. It provides the appearance of a uniform address space by hiding the details of the underlying cache architecture from the programmer. The tags help in dynamically locating the cached memory references that can be hard to analyze statically.

Making decisions in hardware usually forces a single, conservative allocation and replacement algorithm. Implementing a more flexible replacement policy entirely in hardware is usually expensive. Hardware-based schemes typically place data using the set-index field extracted from the address. This does not consider the access behavior pattern of the referenced data. For example, two data items referenced temporally adjacent to each other can be placed in the same set, thus causing conflicts. To reduce the effect of conflicts, set-associativity is employed such that conflicting data elements are placed in different ways. But here again, a simplistic replacement policy, such as pseudo-LRU, is used that does not guarantee the absence of conflicts. Moreover, for set-associative caches, on each reference, every way within the set has to be probed to check for the presence of the data. Different applications or separate phases within the same application can have widely varying associativity requirements depending on their locality and working set characteristics. Thus, many of these tag checks are redundant, wasting power at no added performance advantage.

Making replacement and allocation decisions in the compiler can offer several benefits, as it can employ more intelligent heuristics to make replacement decisions at considerably lower costs. Also, the compiler has the added advantage of analyzing the application's future behavior through profiling on a representative input set. Software control can thus customize the cache accesses based on the needs of the application. Software-only approaches like code/data reorganization [8, 28] can help avoid conflicts. However, the hardware, being transparent, is unaware of these transformations and thus performs wasted tag checks.

We employ a hardware/software co-managed caching policy where the hardware performs the critical cache lookup to reduce access time, while providing directives through software for efficient management. Exposing the operation of the cache to the software/compiler facilitates efficient use of this critical storage for both power and performance.

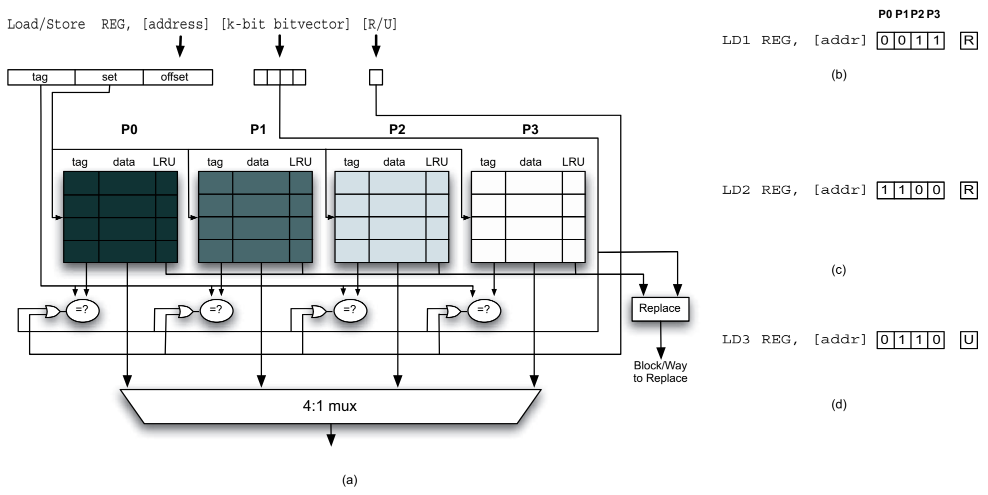
Partitioned Caches: Cache partitioning allows for coarser grained management by the software as compared to hardware-based replacement, while delegating critical finer granularity operations, such as tag checks, to the hardware. Traditionally, caches are logically organized into ways, where the tag corresponding to each of the ways are compared in parallel to reduce access cycle time. The data is read from the matching way. Prior to matching, the parallel access has to pre-charge and read all the tag and data arrays, but select only one of the ways[1], resulting in wasted dynamic energy [22]. Ideally, in an n-way set-associative cache, using an oracle predictor, only one of the n-ways must be read and the rest can be ignored.

In vertically partitioned [40] or way-partitioned caches [9], each way is treated as a separate partition. Prior work uses vertical partitions for either energy savings through hardware control [40], or as coarser grained units that are managed as scratch-pads [9]. We generalize this idea by allowing the compiler finer grain control over the individual ways. In particular, on a cache access, the compiler decides the partitions or ways to be probed for the referenced data. Similarly, cache replacement is restricted to certain ways so as to reduce conflict misses. Although multiple ways can reduce conflicts, pseudo-LRU replacement algorithms are non-optimal, and thus cannot guarantee correct decisions. By proactively placing temporally co-located data references in different partitions, the compiler can avoid conflicts. Similarly, references that have poor temporal locality can be restricted to a single partition or even be allowed to bypass the cache by not assigning them to any of the partitions, thus preventing cache pollution. Smaller L1 caches with low associativity, high degree of conflict misses, and requiring a single cycle access are ideal candidates for way-based partitioning. It should be noted that partitioning need not be restricted to individual ways, but rather can include multiple ways per partition.

More importantly, in addition to reducing conflict misses, way-partitioned software-managed caches can restrict cache lookups to only selected ways that are guaranteed to contain the data. This provides a two-fold advantage. First, restricting memory references, based on its individual memory needs, to only a subset of the available cache can help save energy as only the assigned set of tags and their corresponding data arrays are activated. Second, per-access cycle time can be improved since only a limited set of ways are read on a single access. Ideally, the compiler can restrict each access to just a single way, thus matching the energy savings of an oracle predictor.

Way partitioning provides an ideal platform for the compiler to exercise fine-grained management of data placement by reducing conflicts while lowering the energy consumed. This work focuses on L1 caches, although the technique can be generalized to other levels of the cache hierarchy. Single level caches are common in embedded domains, where energy is a primary design constraint. We focus on software-based way-partitioning of L1 caches, as they are typically small and capture majority of the memory references.

[1] L1 caches generally employ parallel access to ensure the fastest access time. Lower levels of the memory hierarchy, i.e., L2/L3 caches, often serialize tag and data access to reduce unnecessary energy consumption as access latency is less important.



Load/Store REG, [address] [k-bit bitvector] [R/U]
tag
set
offset
P0
P1
P2
P3
tag data LRU
=?
Replace
Block/Way to Replace
4:1 mux
(a)
P0 P1 P2 P3
LD1 REG, [addr] 0 0 1 1 R
(b)
LD2 REG, [addr] 1 1 0 0 R
(c)
LD3 REG, [addr] 0 1 1 0 U
(d)

Figure 1. Software-managed vertically partitioned cache: (a) Cache design, (b)-(d) Load/Store instructions with partition bitvector annotations.

3. Partitioned Cache Architecture

In this section, we look at the architecture and the ISA extensions required for a software-managed partitioned cache. The cache controller is altered to allow for a specified subset of the cache ways to be activated on a lookup. In addition, on a cache miss, the cache can be directed to only use a subset of the ways during replacement. Modified load/store instructions are used by the compiler to control the operations of the cache.

An important factor in determining the cache design is the granularity of partition assignment. Partition assignment should ideally be made at the cache block level, thereby matching replacement decisions. However, since memory blocks that a program accesses can vary for every run of the program, we instead assign partitions to the load or store instructions that access these blocks. By guiding the memory instructions, data placement in the cache is indirectly controlled. In order to guide load/store operations to their designated cache ways, we extend the instruction set architecture to include partition designations.

Figure 1 shows a 4-way set-associative cache. Each way is treated as a software-controlled cache partition. As shown on top of Figure 1, we conceptually extend the load/store instructions with additional operand fields. For a k-partitioned cache, a k-element bitvector immediate field is used to denote the partition(s) (i.e., way(s)) to which the instruction is assigned. For example, in Figure 1(b), *LD 1* is assigned to partitions 2 and 3. Similarly, *LD 2* in Figure 1(c) is assigned to partitions 0 and 1, and *LD 3* in Figure 1(d) is assigned to partitions 1 and 2.

Cache Replacement: On a miss, a block within the cache must be selected for replacement. Only those ways specified by the replacement bitvector are considered for eviction. This allows the compiler control over the replacement decisions among the ways in a set. If an instruction is assigned to multiple partitions, LRU or another replacement policy can be used to choose among the specified partitions. For regular load/store instructions that are unannotated with bitvectors, all the partitions are considered for replacement. Thus, the flexibility of the underlying hardware allows the compiler to treat individual loads as needing any number of ways of the cache, based on its access characteristics.

Cache Lookup: On a cache access, all ways that could possibly include the cache block must be probed. This is required to avoid any coherency and duplication of cache blocks. If two memory references sharing the same set of data objects are placed in different partitions, then all such partitions have to be checked for the presence of the data. The lookup could be accomplished by adding a second bitvector for lookup to each instruction, which corresponds to all the partitions that could hold the referenced data object. During lookup, only the specified ways have to be probed. Separating replacement from lookup provides the combined flexibility of improving performance by controlling data placement, while saving power by avoiding redundant tag checks.

While the lookup bitvector would accomplish the task, its task can easily be folded into the original replacement bitvector and a single extra bit-field. This bit-field, called the R/U-(**R**estricted/**U**nrestricted) bit, is added to each load/store instruction and shown in Figure 1. This field is used to restrict the tag checks to the partitions specified in the bitvector. *R* means that only the specified partitions in the replacement bitvector need be probed, while *U* forces all the partitions to be checked. Although a unified lookup/replacement bitvector is less general than having a separate lookup bitvector, it has been found to provide comparable benefits at reduced costs.

This lookup optimization requires the compiler to guarantee that two memory instructions assigned to different partitions with the R-bit will never access the same data. More details on how this is done is described in Section 4. In the example shown in Figure 1, *LD 1* and *LD 2* are assigned to two different sets of partitions with their R-bit set.

ISA Support: The ISA extensions as described above require adding more encoding bits, which might not be practical for embedded processors where code size is a primary design constraint. This overhead can be reduced by using a special purpose register, called the *cache access register* (CAR), to hold the partition bitvector. This allows extending the design to an arbitrary number of partitions. Further, we introduce two different types of load/store instructions. One that is partition cognizant, while the other that is not. The partition aware instruction implicitly sources the CAR. New move instructions are required to initialize the CAR with the appropriate partition bitvector and the R/U-bit values. Partition unaware instructions do not use the CAR and perform lookup and replacement on all the ways just as in traditional designs. As shown in our experiments, the resulting code size overhead is only 0.4%. Further, multiple such registers can be used to avoid scheduling constraints. Alternately, if the ISA permits, explicit general purpose register operands can be used to hold the bitvector.

Besides the modifications to the cache controller to honor the partitions specified by the current instruction, no additional hardware beyond a standard cache is necessary. The tag directory structure is retained. If partitioning is not supported, the assignment can be ignored and the default set-associative scheme can be used. For future generation processors that could be designed with higher or lower associativity, the compiler specified partitions can be virtualized by allowing multiple specified partitions to refer to a single way or a single specified partition to refer to multiple ways.

The objective of this work is to balance two opposing goals. On the one hand, cache access energy can be reduced by restricting memory references to as few ways as possible. Energy is reduced by limiting the number of tag and data arrays that are activated during each access. Moreover, controlled placement can help avoid conflict misses, thus eliminating off-chip accesses. However, this can potentially lead to capacity misses for references that have a moderate to large working sets. Large working sets are handled by allocating ways to the corresponding accesses. Therefore, the compiler must balance these trade-offs so as to reduce conflicts while minimizing energy.

4. Compiler Partitioning of Memory Instructions

Software-managed cache partitioning gives the programmer or the compiler explicit control to manage the different partitions based on the memory needs of the application. In this section, we describe a compiler heuristic that automatically analyzes an application and assigns the important load/store instructions within the program to different cache partitions. The compiler assigns these memory reference instructions to partitions (i.e., cache ways) with two major goals: (1) reducing conflicts among data objects that are simultaneously accessed and (2) restricting the instructions to the appropriate number of partitions to satisfy the working set of that reference. The first goal improves cache utilization by eliminating conflict misses that arise due to interferences among temporally co-located data reference streams. The second goal tries to reduce the number of redundant tag/data array checks that would otherwise have to be performed by the hardware. These goals provide the combined benefit of achieving the performance advantage of set-associative caches while reducing the energy consumed.

Compiler-managed partition assignment consists of two phases: cache estimation and assignment. The cache estimation phase analyzes the memory access and usage characteristics of the application to estimate the cache configuration requirement of individual loads and stores. In addition, it also computes the degree of temporal interference among each of the loads/stores. In the assignment phase, a greedy heuristic assigns memory instructions to partitions. We employ a combined static and profile-driven compiler analysis. Pointer-analysis is used to control cache lookups for correctness, while profiling is used to guide partition assignment. Profile-based analysis is more accurate in the presence of dynamically allocated data structures. Alternately, affine array index analysis [35] can be used to compute the memory reuse patterns and conflicts of data objects.

Figure 2(a) shows a sample C code segment with nested loops. This example is used throughout this section to illustrate the partitioning process. The load and store instructions that correspond to the accessed arrays are labeled in the source. Between the two innermost loops (with indices *j* and *k*), the arrays *y* and *x* are reused, while accesses to the *w* objects (*w1* and *w2*) are distinct. During execution of each innermost loop, the accesses to *y*, *x*, and the *w* objects are temporally co-located. So to prevent cache misses, each of these data objects have to be assigned such that they do not overlap with each other. Also, during the execution of the second loop, *w2* should not displace either *y* or *x*. But since *w1* and *w2* are distinct, they have no reuse and can overlap in the cache.

4.1 Phase 1: Cache Estimation

In order to assign partitions to the load/store instructions, the compiler first analyzes the application to identify three key attributes: (i) the data sharing pattern among different load/store instructions, (ii) the cache configuration required to satisfy the working set requirement of each load/store instruction, and (iii) the degree of conflict between every pair of load/store instructions.

Multiple instructions could share access to the same set of data structures. Grouping such references helps to restrict the number of partitions to which they are assigned. In addition, it guarantees that instructions that access the same set of data objects have the same partition assignment. The working set size estimates how much cache should be allotted to the instruction, while the degree of conflict guides placement of instructions to different partitions. These attributes are mined independent of the address and the organization/replacement-policy of the target cache architecture.

The compiler initially profiles the code by running the application on a train input set. The profile run generates a trace of the load/store instructions executed during the run along with the addresses referenced in units of cache blocks. The traces are processed on-line by analyzing a window of references that slides over the generated trace. The window size is bound to two times the size of the cache in units of cache blocks.

4.1.1 Computing Data Sharing Among Load/Stores

Capturing commonality between instructions that access the same data can help limit cache accesses to just the required set of partitions. This involves using points-to analysis to guarantee that instructions that could potentially access the same data set are assigned to same partitions, and proactively merging such instructions to prevent them from being assigned to different partitions. This, in turn, reduces the number of partitions that are to be activated at run-time.

Points-to Analysis: To restrict the load/store instructions to access a subset of cache partitions, the compiler has to guarantee that no two references assigned to two different partitions can access the same data item. Pointer analysis is used to avoid this problem. The pointer analysis phase within our compiler annotates every load/store instruction with a set of object identifiers that it potentially accesses [23]. The objects can be global/stack arrays/variables/structures or heap allocated objects.

Initially, each distinct object identifier and its associated load/store instructions are placed into a separate object set. If an instruction accesses multiple data objects, they could reside in different object sets and hence the corresponding sets are merged. This process continues until a set of completely disjoint object sets are obtained. All instructions within the same object set have to be assigned to the same set of partitions. More conservatively, if they are assigned different partitions, their U-bit is set so that they check all available partitions on each reference for correctness. Instructions within each object set are guaranteed not to access data objects of another set and hence can be assigned the R-bit. Each such fused set are now treated as a single new instruction for the remainder of the analysis.

Heuristic Fusing: The pointer analysis phase is usually conservative and can potentially fuse more instructions than necessary. To avoid this, we also perform heuristic fusing. Based on the profile information, all instructions that share more than a threshold (60% is used in our experiments) of the commonly accessed cache blocks are fused. This is a less aggressive, but more accurate form of fusing, which helps to group instructions that truly alias.

Heuristic fusing is only used for the subsequent assignment phase. But, the setting of the R/U-bit is done based on the conservative pointer analysis information to guarantee correctness (see Section 4.2). To ensure correctness, pointer analysis may poten-

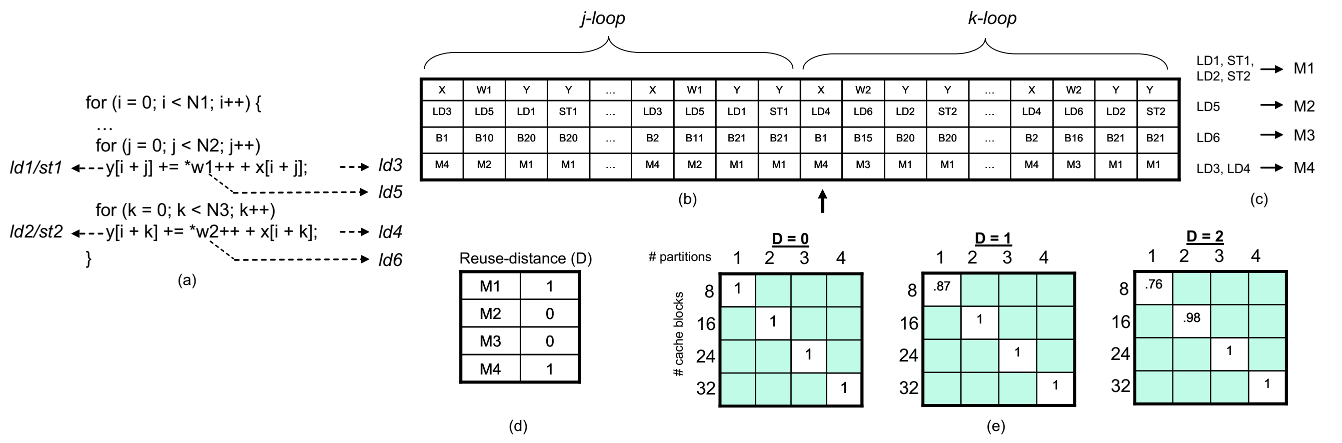


Figure 2. Compiler partitioning example: (a) Annotated code segment, (b) Trace consisting of array references, cache blocks, and load/stores from the example in Figure 2, (c) Fused load/store instructions, (d) Reuse distance (D) for each fused instruction, (e) Hit rate estimate for different cache configuration using Equation 1.

tially tag two references as 'may alias', although at run-time they have a low probability of aliasing. Conservatively grouping such references forces them to be assigned to the same set of partitions leading to conflicts. Heuristic fusing avoids this by separating these references such that they can be assigned to different partitions. But, based on pointer analysis, the U-bit for both these references have to be set because, in the unlikely event of an alias, all other partitions have to be probed for any previously cached data. Although this fails to reduce redundant tag checks, it can help in improving performance by proactively reducing conflicts. Thus, the hardware-supported decoupling of replacement from lookup allows the compiler to perform aggressive optimizations without being overly conservative.

An important point to note is that pointer analysis cannot detect the case when two loads access different objects that fall within the same cache block. Data padding is used to prevent such false sharing. Since most objects occupy multiples of cache lines, the overhead due to padding is minimal. Alternately, for objects that exhibit high locality, the corresponding load/store instructions can be grouped (based on static analysis [17]) to ensure that the same placement decisions are made for the group.

We use the trace shown in Figure 2(b) to illustrate the partitioning process. This is derived from the code in Figure 2(a). The B_is represent the cache blocks that are accessed by the load/store instructions labeled in Figure 2(a). Along with the instructions, the corresponding data objects referenced are also shown. Note, while executing each innermost loop, the accesses to individual arrays can span multiple blocks. For example, accesses to array *x* in the *j*-loop spans blocks B1 and B2. Instructions LD3 and LD4 reference the same array *x* and are grouped into a fused instruction. In this example, both points-to analysis and heuristic fusing leads to the same result, and hence for ease of explanation we assume conservative fusing based on pointer analysis. The grouping of instructions is shown in Figure 2(c). Each fused instruction is shown as M_i and is also listed at the bottom of the trace in Figure 2(b).

4.1.2 Estimating Cache Requirement For Load/Stores

The goal of the compiler is to allocate sufficient cache partitions to each load/store instruction to hold its expected working set. By assigning the right number of partitions, we eliminate capacity/conflict misses for all data accessed by that reference, while avoiding any redundant tag/data array accesses.

Working Set Size Estimation: To estimate the number of partitions, first the working set size of each load/store instruction, based on the concept of stack-reuse distance [19] (denoted as D), is computed. For a specific instruction L_k, its cache block reference in temporal order are used. The size is estimated by looking at the past references to unique cache blocks by that instruction. The trace is scanned in reverse order starting from the current reference B_k. All unique references to B_i ($\neq B_k$) until the last occurrence of B_k is the number of cache blocks required (reuse distance) such that the current occurrence of B_k hits in a fully associative cache. The last B_k seen is then removed from the trace, and the current B_k becomes the last occurrence for subsequent passes. The working set size of L_k is one more than the weighted average of such reuse distances.

Consider the example trace shown in Figure 2(b). The current reference is shown by the arrow pointing to the fused instruction *M4* that references the block *B1*. Prior to this, *M4* references *B2* after referencing *B1* at the beginning of the trace. Thus, for the current reference of *B1* to hit, *M4* requires at least two cache blocks. The reuse distance (D) for each reference M_i is shown in Figure 2(d).

Number of Partitions: The number of partitions assigned to a load/store should be large enough to hold its working set. This requires an accurate estimation of the hit rate of a given load/store instruction for various cache configurations based on its reuse distance.

Given a reuse distance D, cache size in terms of number of blocks B, and associativity A, the hit rate can be approximately computed using the formula [5]:

$$\sum_{a=0}^{A-1} \binom{D}{a} \left(\frac{A}{B}\right)^{a} \left(\frac{B-A}{B}\right)^{D-a} \tag{1}$$

The reuse distance D can be interpreted as: for a given reference B, there is, on average, D unique references to other blocks B_i between two unique references to B. The above formula assumes that the intervening references have an equal chance of being placed in any of the cache blocks within a set.

Using Equation 1, the hit probability for different cache configurations is shown in Figure 2(e). Each entry in the matrix represents the hit rate for a given value of D. The rows of the matrix are the number of blocks in the cache, while the columns represent the associativity (or the number of partitions) in the cache. Here, we are only concerned with the entries in the diagonal of

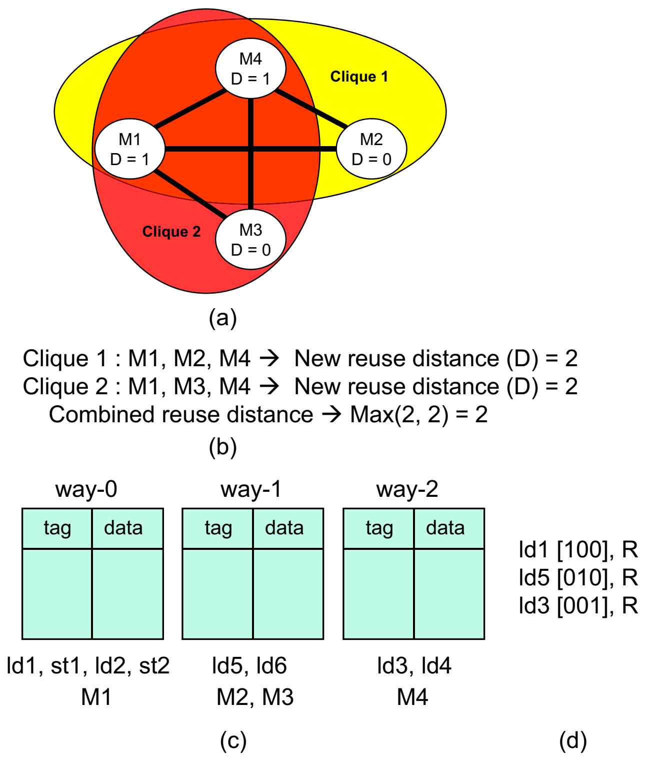


Figure 3. Continuation of compiler partitioning example from Figure 2: (a) Interference graph with cliques shaded, (b) The new reuse distance for each of the cliques, (c) Assignment of loads/stores to partitions/ways, (d) Annotating with partition bitvectors.

the matrix. The top left entry represents a single partition (or way) with the total number of blocks same as that within a single way. As we proceed along the diagonal, the cache size is increased by adding more ways. The off-diagonal entries correspond to cases where partitions/ways can be combined to produce larger direct-mapped caches.

The number of partitions required for each load/store instruction with a given reuse distance D, is computed from the above matrices by picking the entry along the diagonal with the highest hit rate. Although, we have shown a performance driven matrix, in reality, we use an energy matrix, where each entry corresponds to the energy consumed for that reference. The energy is computed using the formula $NumReferences * EnergyPerAccess(B, A) + (1 - HitRate) * NumReferences * EnergyPerMiss$, where $EnergyPerAccess$ is a computed from CACTI [27] for a cache configuration with B blocks and A ways, while $EnergyPerMiss$ is the energy required to fetch a cache line from L2 or off-chip memory on a miss. The $HitRate$ is obtained from the performance matrix. Using this new energy matrix, the least energy consuming configuration is selected for each load/store instruction.

4.1.3 Computing Interferences between Loads/Stores

Section 4.1.2 computed the number of partitions required for individual load/store instructions. Since the cache is shared, it is important to assign memory instructions to the cache so that they do not conflict with each other. References that overlap temporally are to be placed in different cache partitions, while references that are not simultaneously live can share the same set of partitions.

This temporal interference between load/store instructions is captured using a graph-based representation, aptly named the *interference graph* (IG). IG nodes are the fused load/store instructions. An edge exists between the nodes if the degree of overlap exceeds a threshold. To compute the interference, a single pass is made over the trace. Every load/store reference that occurs between two consecutive occurrences of another load/store is recorded. Using the trace in Figure 2(b), the IG for the continuing example is presented in Figure 3(a). It can be observed from the trace that during the execution of the inner *j*-loop, references M1, M2, and M4 occur together temporally, while in the *k*-loop, M1, M3, and M4 overlap temporally. But references M3 and M2 occur at two different points in time and do not interfere with each other and hence do not have an edge.

4.1.4 Spatial Locality-Based Optimizations

Up to this point in the discussion, the temporal locality properties of the application have been captured through trace-based analysis. The target architecture's default block size was used to capture the spatial locality for each reference. But, different applications have differing spatial locality characteristics. Thus, the ideal block size specific to each application has to be computed.

We saw earlier how the working set is computed in units of cache blocks for each memory instruction. The cache configuration, derived from Equation 1, assumes that each block in the working set has an equal probability of conflicting with every other block. This turns out to be overly pessimistic for references with high spatial locality and can force the reference to be assigned to multiple caches. Data references that exhibit a high degree of spatial locality can be assigned to a direct-mapped cache, as such a cache maps spatially adjacent references to adjacent cache blocks and avoids conflicts. Thus, accurate estimation of block sizes can help restrict memory references to fewer partitions.

To compute the ideal block size of the application, we vary the block size starting from the smallest granularity (the block size of the target cache architecture) to a maximum size (the size of a single partition) using a single pass of the trace. For each block size, the fraction of references that can be captured by a block of the new size is determined. The block size that maximizes this fraction is selected as the new block size. This new block size is computed prior to the earlier described phases and is used to estimate the reuse distance, the working set size, and the cache configuration as described above.

In Figure 2, using the default block size, M4 has a working set of two cache blocks. But the two blocks that M4 accesses, B1 and B2, are spatially adjacent as they are part of the array *x*. Hence, they will not conflict in a direct-mapped cache. By using twice the default block size, the new working set is just a single block.

4.2 Phase 2: Cache Assignment

The goal of the assignment phase is to bind load/store instructions to partitions/ways in the cache. Instructions that exhibit a high degree of temporal overlap are placed in different partitions so as to prevent their data sets from conflicting.

In Section 4.1.2, the least energy consuming cache configuration was selected for each load/store instruction based on reuse characteristics. If there were an infinite number of partitions, each load/store instruction could be assigned such that its individual working set requirements are satisfied. But, since the number of partitions are limited, some instructions must be assigned to the same set of partitions. Again, if these overlapped instructions do not interfere temporally, there would not be any conflicts. Ideally, when faced with a choice of placing a set of instructions in the same set of partitions, the trace can be used to compute the combined working set as described in Section 4.1.2. The least number of partitions that can retain the combined working set can then be estimated for this overlapped set of instructions. But, this would involve multiple passes of the trace and is clearly impractical. So, we need a quick approximation of the combined working set from the individual working set estimates.

To compute the combined working set size for a set of load/store instructions, we use the IG computed in Section 4.1.3. If an edge exists between two nodes, they are assumed to interfere. Hence, the total number of cache blocks required to hold the working set of both these references combined is the sum of the working set size of each reference. But, if they do not interfere, the combined working set size is the maximum of the working set size of both the references.

Computing Combined Working Sets: We use the graph-theoretic notion of a clique to compute a combined working set. Given a set of potentially overlapping instructions, $M_1, M_2, ..., M_n$, we first consider a subgraph within the interference graph consisting of just these nodes. All maximum cliques within this subgraph are then enumerated. Each clique represents a set of instructions that occur together in time and hence may conflict. All such instructions must be assigned sufficient partitions to prevent conflicts. For each clique, the sum of the working set size of each node in the clique is computed. The working set size of the combined set $M_1, M_2, ..., M_n$ is therefore the maximum of the computed sums over all cliques. From this combined reuse distance, Equation 1 can be used to find the most energy efficient cache configuration for this set of overlapped instructions.

For the interference graph composed of the memory reference instructions M1-M4 shown in Figure 3(a), all maximum cliques are enumerated in Figure 3(b). In this example, we assume that the block size is same as that of the target architecture and hence use the same reuse distance for each instruction as listed in Figure 2(d). The reuse distance of the combined graph is two and is shown in Figure 3(b). From equation 1, for the reuse distance (D) value of two, three partitions are required (see Figure 2(e)) to achieve a high hit-rate.

Partitioning Algorithm: We use a simple greedy heuristic to place the instructions in different partitions. Each memory instruction in decreasing frequency order is placed starting from the first partition to the total number of partitions. At each candidate placement point, the most energy efficient cache configuration for that instruction is selected (Section 4.1.2). If there are previously placed instructions, the new working set and the corresponding cache configuration is selected by enumerating all possible cliques for the set of overlapped instructions. Because of the overlaps, the number of partitions for the current instruction can be more than what would have been if only that instruction were to be considered in isolation. Among all such potential placement points, the position that results in the most energy efficient configuration is finally selected for placement for that instruction. This process is then repeated for all instructions.

In our example in Figure 2(a), M1 and M4 have a working set size of one, based on the new block-size computed in Section 4.1.4. For D equal to zero, the number of partitions selected is one (from the matrix in Figure 2(e)). Since M1 and M4 interfere (from the IG in Figure 3(a)), to minimize conflicts, the greedy heuristic places them in different partitions. Alternately, if they were to be overlapped, to fit the combined working set, both the references will have to be assigned to more than one partition, resulting in more tag checks than necessary. Similarly, M2 and M3 require a single partition and do not interfere. Their combined working set fits within a single partition. Hence, M2 and M3 are placed in the same partition but disjointedly from M1 and M4. The final placement of all M_is and their corresponding load/store instructions are shown in Figure 3(c).

R/U assignment: The final step of the assignment process is setting the R/U-bit of load/store instructions. This bit specifies whether the placed instruction needs to check only the assigned partitions during cache lookup for the referenced data. If the R-bit is set, only the assigned ways are probed. If the U-bit is set, all the ways are probed. This is orthogonal to the assignment to partitions. The assignment uses the aliased information computed in Section 4.1.1, where a set of potentially aliasing loads or stores are grouped into a single set. Two instructions from different sets are guaranteed not to interfere. If all instructions within the same points-to set have the same partition assignment, then it is safe to assign the R-bit to each of these instructions. If not, they are assigned the U-bit to avoid coherency and duplication issues. In Figure 3(d), since each of the M_is access distinct data objects, they are all assigned the R-bit. Thus, in this example, the compiler was able to restrict each reference to just a single way, thus providing the energy savings of an oracle way-predictor, while maintaining the hit rate of a 3-way hardware cache by proactively avoiding conflicts through careful placement.

5. Experimental Evaluation

5.1 Methodology

We use the Trimaran [36] compiler and simulator infrastructure for our experiments. The simulator was modified to support the trace analysis described in Section 4. A parametrized cache simulator was built to model way-partitioning based on the annotated load/store instructions generated by the compiler. We assume a RISC, single-issue processor, similar to ARM926EJ, to study the effects of partitioning. The compiler includes aggressive classical optimizations, function inlining, and pointer analysis for load/store optimizations.

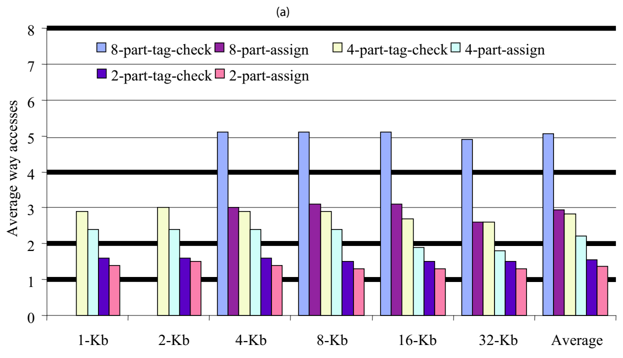
Benchmarks from the MediaBench benchmark suite are evaluated on varying L1 data cache sizes and configurations. The partitioning is performed using the train input set run to completion, while results are reported on a reference input set. The cache sizes are varied from 1 Kbyte to 32 Kbytes, all with a 32 byte block size.

We evaluate three different way-partition configurations - *2-part*, *4-part*, and *8-part*, where *n-part* denotes a partitioning of the original cache of *n-ways* into *n-partitions* such that each partition is a single way that is direct-mapped and software managed. We compare against traditional hardware-based 1-, 2-, 4-, 8-, and 16-way set-associative cache configurations of different sizes. Since partitioning can default to all ways, the appropriate comparison has to be made between *2-*, *4-*, and *8-part* to the respective 2-, 4-, and 8-way set-associative cache. We assume LRU replacement policy for the hardware-based cache configurations. For the partitioned cache, when an instruction is assigned to multiple partitions, LRU is used to select among the assigned partitions.

The basic motivation behind these configurations was to ideally restrict instructions to just one partition, while assigning conflicting instructions to different partitions. This allows only a single smaller direct-mapped way to be activated during each access while achieving the miss-rate equivalent to a *n-way* cache. The goal is to level or even out-perform a set-associative cache while being below the access energy envelope of a direct-mapped cache as individual ways are smaller than a unified direct-mapped cache.

5.2 Results

Tag-checks & Way Assignment: Figure 4(a) shows how effective the placement is in restricting the number of ways assigned. It plots two metrics: (i) the average number of dynamic data/tag-array accesses, and (ii) the average number of partitions that are assigned per instruction. The first metric is larger than the second as more ways must be probed to check for the presence of the referenced data. The lines shown in bold are the same metric for a hardware managed direct, 2-way, 4-way, and 8-way caches where both the number of tag-checks and replacements are same as the



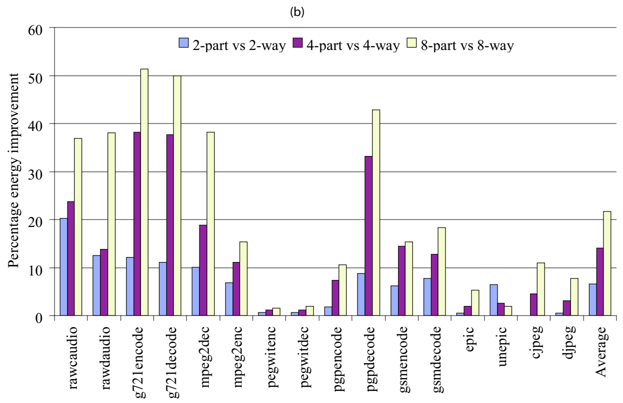
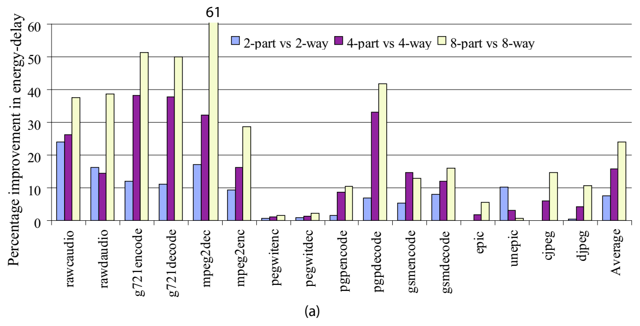


Figure 4. (a) Average data/tag-array accesses and partition assignment for different cache sizes and configurations and (b) Percentage reduction in energy for a 16 Kb cache.



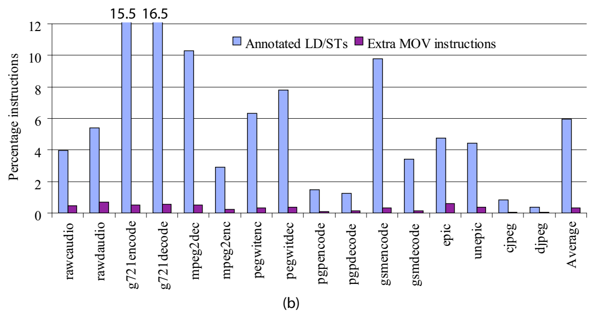


Figure 5. For a 16 Kb cache, (a) Percentage reduction in energy-delay and (b) Percentage annotated ld/st instructions and code size overhead.

associativity of the cache. The partitioned cache can independently control both the metrics and hence their values are different.

On average, for the *8-part* configuration, we observe a 36% reduction in cache accesses and a 63% reduction in assigned ways (Figure 4(a)). For the *8-part* configuration, the average number of assigned ways (second bar under *Average*) is around 2.9. This means that although there are 8-ways, on an average, it behaves like a 2.9-way associative cache. Other configurations show smaller reductions as they have lesser partitions.

As we scale the number of partitions and their sizes, the number of assigned ways decreases proportionally. This is due to the nature of our heuristics that adapt depending on the available cache partitions. Larger partitions satisfy the working set needs of each instruction and hence there is less need to "spread" them across more ways (Section 4.1.2). Despite assigning to less ways, we are able to achieve miss rates comparable to hardware-based set-associative caches that must activate all the available ways.

Improvement in Energy: We focus our energy results on the data cache sub-system as our optimizations target only the data cache. Figure 4(b) compares the percentage improvements in cache energy saved when compared to the corresponding hardware managed cache with the same number of ways on a 16 Kb cache.

The energy measurements are obtained using CACTI-3.2 [27], assuming the same physical configuration as shown in Figure 1. For all caches, we obtain the energy per access for a single way and scale it by the number of ways activated to compute the total energy [22]. The Am41PDS3228D SRAM [2] was assumed to be the off-chip memory with 3.024nJ per access (16-bits).

Since *8-part* eliminates all of the misses with a few number of ways, it achieves the highest relative energy-savings of around 20%. The *2-part* is not able to eliminate as many tag/data-array checks when compared to *4-* or *8-part* caches and hence we observe a comparatively smaller relative energy improvement. For *g721encode*, we observe almost 50% savings in relative energy for a *8-part* configuration. On average, when all configurations are compared relative to a direct-mapped cache (not shown), *2-part* partition is the most energy efficient. This is because, the 2-partitions are able to remove most of the misses with an average of 1.6 tag-checks.

Improvement in Energy-Delay: MediaBench applications have a small data memory footprint and hence do not require large caches. On a 16 Kb cache, the conflict misses were almost negligible. Thus, there was no performance advantage in using a partitioned data cache compared to a traditional hardware-managed set-associative cache. But, the partitioned cache has the advantage of achieving the same performance advantage of a corresponding set-associative cache without incurring the tag/data-array access overhead.

Figure 5(a) compares the percentage improvements in the energy-delay product when compared to the corresponding hardware managed cache with the same number of ways on a 16 Kb cache. Since we model a single-issue machine, we use the simple performance equation: $Hits + Misses * MissPenalty$, where miss-penalty is assumed to be the off-chip latency of 25 cycles [33]. Energy-delay shows similar trends as energy, where relatively, *8-part* shows maximum savings.

Benchmark	**2-part**	**4-part**	**8-part**	**1-way**	**2-way**	**4-way**	**8-way**	**16-way**
rawcaudio	1.3	1.2	1.3	3.8	1.4	1.4	1.4	1.4
rawdaudio	1.4	1.5	1.5	4.2	1.6	1.5	1.5	1.5
g721encode	0.0	0.0	0.0	1.9	0.0	0.0	0.0	0.0
g721decode	0.0	0.0	0.0	2.0	0.0	0.0	0.0	0.0
mpeg2dec	19.0	20.3	19.2	43.1	21.4	26.1	35.7	42.2
mpeg2enc	6.7	4.8	4.6	13.8	7.1	5.2	5.7	8.6
pegwitenc	71.0	70.6	70.2	77.0	71.1	70.6	70.3	70.5
pegwitdec	94.9	94.3	93.8	99.9	95.1	94.5	94.1	94.3
pgpencode	21.1	20.1	20.1	23.1	21.1	20.5	20.1	20.1
pgpdecode	2.0	1.6	1.8	5.4	1.8	1.6	1.6	1.5
gsmencode	1.2	1.0	1.2	1.9	1.1	1.0	1.0	1.0
gsmdecode	1.2	1.1	1.0	1.8	1.2	1.1	0.9	0.9
epic	28.2	27.7	28.2	31.3	28.0	27.6	28.3	28.5
unepic	19.4	13.7	13.8	24.8	20.5	13.8	13.6	13.7
cjpeg	39.3	25.8	21.3	43.0	38.4	26.3	22.5	22.1
djpeg	48.5	30.4	24.7	75.0	48.5	30.9	25.9	25.7
Average	**22.2**	**19.6**	**18.9**	**28.2**	**22.4**	**20.1**	**20.2**	**20.8**

Table 1. Misses/1000 instructions with various 1 Kb software and hardware partitioned cache configurations.

Impact on Cache Misses: Table 1 shows the misses/1000 instructions for different cache partitions on a 1 Kb cache. A smaller cache was used to highlight the ability of the partitioned cache architecture to reduce conflict misses.

On an average, both *4-* and *8-part* partitions perform better than even a 16-way hardware managed cache, while the *2-part* partition performs slightly better than the corresponding 2-way cache. For mpeg2dec, *4-* and *8-part* outperforms the 4-way and 8-way cache, where we observe an increase in misses for higher associativity. This is due to the non-optimality of LRU. Overall, partitioning is able to perform as good as the corresponding set-associative cache.

For the *4-* and *8-part* caches, partitioning is able to remove around 80% of the conflict misses compared to a direct-mapped cache, which is close to a 8-way hardware-managed cache (85%). We observed that capacity misses for the partitioned cache were lower than in the corresponding set-associative cache. An *8-part* cache achieved a 16% reduction in capacity misses compared to an 8-way cache. This is because capacity misses are defined based on a fully-associative cache with LRU, which is not optimal. The compiler-directed data placement is able to eliminate the misses such that it equals or even betters the hardware-based strategy by pro-actively placing the conflicting data elements in different ways.

In our study, we also varied the cache sizes to study the impact of partitioning on different sized caches with different numbers of partitions. In general, we found that for smaller caches the miss-rate improvement is more for partitioned caches because of higher conflict misses. Higher conflict misses for the base case provides opportunities for the compiler to reduce them by pro-actively avoiding conflicts among co-accessed data items using a whole program knowledge. Secondly, two and four partitions often perform as good as a 8-way set-associative cache. Since fewer partitions consume less energy while maintaining low miss-rates, they are recommended partition configurations.

Code Size: Figure 5(b) shows the percentage of static load/store instructions that are annotated by the compiler on a *4-part* 16 Kb cache. The compiler annotates only the most frequently executed and profitable instructions, while the rest are assigned to all partitions. On average, only 6% load/stores are annotated. As described in Section 3, these annotated instructions require an extra move instruction to initialize the CAR with the bit-vector corresponding to the assigned partitions. Since many instructions are assigned to the same set of partitions, common sub-expression elimination and loop-invariant code-motion is applied to remove such redundant moves. The graph also shows that the static move instructions (right bar of Figure 5(b)) inserted average around 0.4%. The dynamic cycle count increase observed is less than 1%. Thus, the code size increase and performance overhead of these moves is negligible.

6. Related Work

Multiple/Split Partitioned Caches: A variety of hardware cache organizations [11, 26, 29] consisting of multiple/split caches aimed at storing data based on spatial, temporal, or a combination of access pattern behavior have been explored in the past. All of these schemes employ hardware techniques to dynamically classify memory blocks into each of the special caches (partitions). The cache controller is modified to detect the access pattern and route the data to the appropriate cache partition. Recently, hardware-based programmable decoders have been suggested in [41] to reduce conflict misses in direct-mapped caches. Hardware-based dynamic partitioning of shared caches for multiple processes or threads [34] have also been proposed.

The use of compile time classification of memory reference instructions into spatial, temporal, and spatial-temporal has been explored in [30]. The classified data references are then cached into three separate organizations. At run-time, the cache controller places data in a given cache depending on the instruction. Spatial and temporal caches are very small and fully associative, while spatial-temporal caches are larger. Different block sizes are also used for each of the caches.

Although our partitioned cache approach is similar in spirit to earlier work on split caches, our scheme is more flexible in that we allow more generalized form of partitions. In addition, instead of a dedicated hardware controller deciding on what data needs to reside in which partition, the compiler, is used to make partitioning decisions. Most other partitioning schemes physically partition the caches with customized configurations which might not be applicable for all workloads. In comparison, our scheme can easily be defaulted to a traditional unified cache by using regular load/store instructions.

The partitioned cache techniques presented in [14, 25] differ from the scheme we propose in several aspects. Their hardware scheme does not handle coherency issues, whereas we selectively probe all cache partitions to detect and resolve coherency and data duplication. Our scheme also has the ability to specify multiple non-contiguous partitions with possibly a global replacement among the different partitions. In addition, we only need to partition a select set of load/store instructions and can easily default to a traditional cache based on the needs of the application. [14, 25] use hardware-based partition descriptor tables to record the size and offset of the partitions in the original cache. The PC of the load/store instruction is used to index into another table to identify the assigned partition. This can affect hit time. They limit their analysis to loops with affine accesses and do not handle multiple partitions for a single load/store.

Coarser-Grained Partitioning: While our method uses a fine-grained approach to partitioning, more coarser-grained techniques have been studied in the past. A hardware/compiler scheme is used in [37] to classify an instruction as cacheable or non-cacheable based on the miss rate. In region-based caching [18], caches are partitioned depending on whether an access is to the heap, stack, or the global address space. Minimax caches [38] partition scalars and non-scalars to different caches to reduce conflicts. Page-based partitioning has been proposed in [33], where a smaller direct-mapped cache is placed next to a larger main cache. To avoid conflict misses, page coloring schemes have been proposed in the past [32]. But, these require additional OS support and are dependent on whether the cache is virtually or physically indexed. Moreover, they are targeted towards reducing just the conflicts within a cache and not towards reducing the number of tag lookups. Our scheme can control data placement irrespective of the underlying addressing schemes. It allows more fine-grained partitioning and control and can emulate the above strategies by annotating the instructions based on their broader classification. Instruction-driven control can generalize to all kinds of data access patterns.

Hardware/Software Cache Management: FlexCache [21] uses software-based cache management by grouping references to hot pages to avoid redundant tag-checks. This can affect hit-case latency, which is reduced using ISA extensions and special registers to store the address translations. They do not target partitioned caches, which combine the benefit of both traditional caches and software management.

Way-partitioning [9] partitions the ways (columns) of a cache such that the replacement decisions are restricted to certain ways. A bit-vector is used to restrict the allowable ways and use the reserved partitions as scratch-pad memories. They do not make fine-grained replacement decisions on a per instruction basis, which allows us to tackle both energy and conflict misses. Their technique performs partitioning at the page-level by modifying the TLB. Moreover, they do not allow restricting lookups to the assigned partitions. Thus, our technique generalizes on their method.

Cache management through compiler specified hints has been proposed in [31] to decide what data is to be retained/evicted. But, their focus is on reducing conflicts and are not always applicable in general, especially in the presence of dynamically allocated data. The hardware still performs energy inefficient tag checks. Our work instead tries to efficiently use the available ways in a set-associative cache for energy improvements while maintaining performance through a mix of static and profile-driven analysis. The proposed solution can be applied over the existing code/data re-organization techniques [28, 8] to further improve performance.

Hardware/Software Techniques Towards Cache Energy Savings: To reduce the energy consumed in set-associative caches, recently, pseudo set-associative caches have been proposed [7, 12, 13, 42]. The basic idea is to probe each of the ways sequentially or use some form of hardware way prediction. For the common case, where the first access results in a hit, there can be substantial savings in energy and access time. Unlike our method, the probing is done in hardware with no compiler control. Our technique is more general, as it can selectively activate different sets of tag/data-arrays for different references in the application without incurring any cycle time overheads. In fact, for instructions that need to access multiple ways, we can allow techniques similar to theirs to further reduce power at the expense of increased cycle time. Dynamically reconfigurable caches have been proposed in [1, 3, 40] where selected portions of the cache can be disabled for energy savings and for dynamically tuning the memory configuration depending on the application's needs. Compiler-based techniques to reduce tag energy have been proposed in [39]. These techniques do not try to reduce conflict misses. Alternately, banking [10] can be deployed to reduce cache access energy, but this is orthogonal to partitioning and can be applied to individual partitions if desired.

7. Conclusion

In this paper, we presented a novel compiler-managed partitioned cache architecture, where individual ways within the cache are explicitly controlled by load/store instructions. These load/store instructions provide directives to the hardware to control placement within individual ways, as well as regulating the ways that need be probed during cache access. This primary benefit is avoiding redundant tag/data array checks, thus reducing energy. Performance improvement can also be achieved by reducing conflict misses through intelligent placement of data. In addition, a compiler algorithm that uses whole program knowledge and profile information is presented that assigns instructions to the partitions with negligible increase in code size. An average of 15% energy savings was achieved with four 4 Kb direct-mapped caches when compared to a traditional 4-way set-associative 16 Kb cache.

Acknowledgments

We would like to thank Dr. Krishnan Kailas and Dr. Zehra Sura of IBM TJ Watson Research Center for the initial discussions on partitioned caches. This work was supported by the Engineering Research Centers Program of the National Science Foundation under award number EEC-9986866, National Science Foundation grant CCF-0347411, and equipment donated by Hewlett-Packard and Intel Corporation.

References

[1] ALBONESI, D. Selective Cache Ways: On Demand Cache Resource Allocation. In *Proc. of the 32nd Annual International Symposium on Microarchitecture* (Nov. 1999), pp. 248–259.

[2] AMD. *Am41PDS3228D SRAM*, 2004.

[3] BALASUBRAMONIAN, R., ET AL. Memory Hierarchy Reconfi g-uration for Energy and Performance in General-Purpose Processor Architectures. In *Proc. of the 33rd Annual International Symposium on Microarchitecture* (Dec. 2000), pp. 245–257.

[4] BANAKAR, R., ET AL. Scratchpad Memory: A Design Alternative for Cache On-Chip Memory in Embedded Systems. In *Proc. of the Tenth International Conference on Hardware/Software Codesign* (May 2002), pp. 26–32.

[5] BREHOB, M., AND ENBODY, R. An Analytical Model of Locality and Caching. Tech. Rep. MSU-CSE-99-31, Michigan State University, Sept. 1999.

[6] BROOKS, D., TIWARI, V., AND MARTONOSI, M. A framework for architectural-level power analysis and optimizations. In *Proc. of the 27th Annual International Symposium on Computer Architecture* (June 2000), pp. 83–94.

[7] CALDER, B., GRUNWALD, D., AND EMER, J. Predictive Sequential Associative Caches. In *Proc. of the 2nd International Symposium on High-Performance Computer Architecture* (Feb. 1996), p. 244.

[8] CHILIMBI, T. M., HILL, M. D., AND LARUS, J. R. Cache-conscious structure layout. In *Proc. of the SIGPLAN '99 Conference on Programming Language Design and Implementation* (May 1999), pp. 1–12.

[9] CHIOU, D., JAIN, P., RUDOLPH, L., AND DEVDAS, S. Application Specifi c Memory Management in Embedded Systems Using Software-Controlled Caches. In *Proc. of the 37th Design Automation Conference* (2000), pp. 416–419.

[10] GHOSE, K., AND KAMBLE, M. B. Reducing Power in Superscalar Processor Caches Using Subbanking, Multiple Line Buffers and Bit-Line Segmentation. In *Proc. of the 1999 International Symposium on Low Power Electronics and Design* (Aug. 1999), pp. 70–75.

[11] GONZALEZ, A., ALIAGAS, C., AND VALERO, M. A data cache with multiple caching strategies tuned to different types of locality. In *Proc. of the 1995 International Conference on Supercomputing* (July 1995), pp. 338–347.

[12] HUANG, M., ET AL. L1 Data Cache Decomposition for Energy Effi ciency. In *Proc. of the 2001 International Symposium on Low Power Electronics and Design* (Aug. 2001), pp. 10–15.

[13] INOUE, K., ISHIHARA, T., AND MURAKAMI, K. Way Predicting Set-Associative Caches for High Performance and Low Energy. In *Proc. of the 1999 International Symposium on Low Power Electronics and Design* (Aug. 1999), pp. 273–275.

[14] IRWIN, J., ET AL. Predictable Instruction Caching for Media Processors. In *IEEE 13th International Conference on Application-specific Systems, Architectures and Processors* (July 2002), pp. 141–150.

[15] KANDEMIR, M., ET AL. A compiler-based approach for dynamically managing scratch-pad memories in embedded systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 23*, 2 (Feb. 2004), 243–260.

[16] KIM, N., ET AL. Circuit and Microarchitectural Techniques for Reducing Cache Leakage Power. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems 12*, 2 (Feb. 2004), 167–184.

[17] LARSEN, S., AND AMARASINGHE, S. Increasing and detecting memory address congruence. In *Proc. of the 11th International Conference on Parallel Architectures and Compilation Techniques* (Sept. 2002), pp. 18–29.

[18] LEE, H.-H. S., AND TYSON, G. S. Region-based Caching: an Energy Effi cient Memory Architecture for Embedded Processors. In *Proc. of the 2000 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems* (Nov. 2000), pp. 120–127.

[19] MATSSON, R. L., GECSEI, J., SLUTZ, D., AND TRAIGER, I. Evaluation techniques for storage hierarchies. *IBM Systems Journal 9*, 2 (1970), 78–117.

[20] MONTANARO, J., ET AL. A 160 MHz, 32-b, 0.5-W CMOS RISC Microprocessor. *Digital Technical Journal 9*, 1 (1997), 49–62.

[21] MORITZ, C., FRANK, M., AND AMARASINGHE, S. Flexcache: A framework for fexible compiler generated data caching. In *Proc. of the Workshop on Intelligent Memory Systems* (Nov. 2000), pp. 8–14.

[22] M.POWELL, S.YANG, B.FALSAFI, K.ROY, AND T.N.VIJAYKUMAR. Gated-VDD: A circuit technique to reduce leakage in deep-submicron cache memories. In *Proc. of the 2000 International Symposium on Low Power Electronics and Design* (July 2000), pp. 90–95.

[23] NYSTROM, E., KIM, H.-S., AND HWU, W. Bottom-up and top-down context-sensitive summary-based pointer analysis. In *Proc. of the 11th Static Analysis Symposium* (Aug. 2004), pp. 165–180.

[24] OZTURK, O., AND KANDEMIR, M. Nonuniform Banking for Reducing Memory Energy Consumption. In *Proc. of the 2005 Design, Automation and Test in Europe* (Mar. 2005), pp. 814–819.

[25] PETROV, P., AND ORAILOGLU, A. Performance and Power Effectiveness in Embedded Processors - Customizable Partitioned Caches. *IEEE Trans. Comput.-Aided Design Integrated Circuits 20*, 11 (Nov. 2001), 1309–1318.

[26] PRVULOVIC, M., ET AL. The Split Spatial/Non-Spatial Cache: A Performance and Complexity Evaluation. In *Proc. of the IEEE TCCA Newsletter* (July 1999), pp. 3–10.

[27] REINMAN, G., AND JOUPPI, N. P. Cacti 2.0: An integrated cache timing and power model. Tech. Rep. WRL-2000-7, Hewlett-Packard Laboratories, Feb. 2000.

[28] RIVERA, G., AND TSENG, C.-W. Data transformations for eliminating confict misses. In *Proc. of the SIGPLAN '98 Conference on Programming Language Design and Implementation* (June 1998), pp. 38–49.

[29] RIVERS, J. A., AND DAVIDSON, E. S. Reducing Conficts in Direct-Mapped Caches with a Temporality-Based Design. In *Proc. of the 1996 International Conference on Parallel Processing* (Aug. 1996), pp. 154–163.

[30] SANCHEZ, J., AND GONZALEZ, A. A Locality Sensitive Multi-Module Cache with Explicit Management. In *Proc. of the 1999 International Conference on Supercomputing* (June 1999), pp. 51–59.

[31] SARTOR, J., VENKITESWARAN, S., MCKINLEY, K., AND WANG, Z. Cooperative Caching with Keep-Me and Evict-Me. In *Proc. of the* 9^{th} *Annual Workshop on Interaction between Compilers and Computer* (Feb. 2005), pp. 46–57.

[32] SHERWOOD, T., CALDER, B., AND EMER, J. Reducing cache misses using hardware and software page placement. In *Proc. of the 1999 International Conference on Supercomputing* (June 1999), pp. 155–164.

[33] SHRIVASTAVA, A., ISSENIN, I., AND DUTT, N. Compilation techniques for energy reduction in horizontally partitioned cache architectures. In *Proc. of the 2005 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems* (Sept. 2005), pp. 90–96.

[34] SUH, G., RUDOLPH, L., AND DEVDAS, S. Dynamic Partitioning of Shared Cache Memory. *Journal of Supercomputing 28*, 1 (Apr. 2004), 7–26.

[35] TEMAM, O. An algorithm for optimally exploiting spatial and temporal locality in upper memory level. *IEEE Transactions on Computers 48*, 2 (1999), 150–158.

[36] TRIMARAN. An infrastructure for research in ILP, 2000. http://www.trimaran.org/.

[37] TYSON, G., FARRENS, M., MATTHEWS, J., AND PLESZKUN, A. R. A Modifi ed Approach to Data Cache Management. In *Proc. of the 28th Annual International Symposium on Microarchitecture* (Dec. 1995), pp. 93–103.

[38] UNSAL, O. S., ET AL. The Minimax Cache: An Energy-Effi cient Framework for Media Processors. In *Proc. of the 8th International Symposium on High-Performance Computer Architecture* (Feb. 2002), pp. 131–141.

[39] WITCHEL, E., LARSEN, S., ANANIAN, S., AND ASANOVIC, K. Direct Addressed Caches for Reduced Power Consumption. In *Proc. of the 34th Annual International Symposium on Microarchitecture* (Dec. 2001), pp. 124–133.

[40] YANG, S.-H., FALSAFI, B., POWELL, M. D., AND VIJAYKUMAR, T. Exploiting Choice in Resizable Cache Design to Optimize Deep-Submicron Processor Energy-Delay. In *Proc. of the 8th International Symposium on High-Performance Computer Architecture* (Feb. 2002), pp. 151–161.

[41] ZHANG, C. Balanced Cache: Reducing Confict Misses of Direct-Mapped Caches. In *Proc. of the 33rd Annual International Symposium on Computer Architecture* (June 2006), pp. 155–166.

[42] ZHANG, C., ZHANG, X., AND YAN, Y. Multi-column implementations for cache associativity. In *Proc. of the 1997 International Conference on Computer Design* (1997), pp. 504–509.