An Integrated Approach for Improving Cache Behavior

Gokhan Memik
EE Dept.
UCLA
memik@ee.ucla.edu

Mahmut Kandemir
CSE Dept.
Penn State
kandemir@cse.psu.edu

Alok Choudhary
ECE Dept.
Northwestern U.
choudhar@ece.nwu.edu

Ismail Kadayif
CSE Dept.
Penn State
kadayif@cse.psu.edu

Abstract

The widening gap between processor and memory speeds renders data locality optimization a very important issue in data-intensive embedded applications. Throughout the years hardware designers and compiler writers focused on optimizing data cache locality using intelligent cache management mechanisms and program-level transformations, respectively. Until now, there has not been significant research investigating the interaction between these optimizations. In this work, we investigate this interaction and propose a selective hardware/compiler strategy to optimize cache locality for integer, numerical (array-intensive), and mixed codes. In our framework, the role of the compiler is to identify program regions that can be optimized at compile time using loop and data transformations and to mark (at compile-time) the unoptimizable regions with special instructions that activate/deactivate a hardware optimization mechanism selectively at run-time. Our results show that our technique can improve program performance by as much as 60% *with respect to the base configuration and* 17% *with respect to a non-selective hardware/compiler approach.*

1 Introduction and Motivation

To improve performance of data caches, several hardware and software techniques have been proposed. Hardware approaches try to anticipate future accesses by the processor and try to keep the data close to the processor. Software techniques such as compiler optimizations [6] attempt to reorder data access patterns (e.g., using loop transformations such as tiling) so that data reuse is maximized to enhance locality. Each approach has its strengths and works well for the patterns it is designed for. So far, each of these approaches has primarily existed independently of one another. For example, a compiler-based loop restructuring scheme may not really consider the existence of a victim cache or its interaction with the transformations performed. Similarly, a locality-enhancing hardware technique does not normally consider what software optimizations have already been incorporated into the code. Note that the hardware techniques see the addresses generated by the processor, which already takes the impact of the software restructuring into account. Also, there is a promising aspect of combining the hardware and software approaches. Usually compilers have a global view of the program which is hard to obtain at run-time. If the information about this global view can be conveyed to the hardware, the performance of the system can be increased significantly. This is particularly true if hardware can integrate this information with the runtime information it gathers during execution.

But, can we have the best of both worlds? Can we combine hardware and software techniques in a logical and selective manner so that we can obtain even better performance than either applying only one or applying each independently?

To answer these questions, we use hardware and software locality optimization techniques in concert and study the interaction between these optimizations. We propose an integrated scheme which selectively applies one of the optimizations and turns off the other. Our goal is to combine existing hardware and software schemes intelligently and take full advantage of both the approaches. To achieve this goal, we also propose a region detection algorithm, which is based on the compiler analysis and that determines which regions of a given program are suitable for hardware optimization and subsequently, which regions are suitable for software optimization. Based on this analysis, our approach turns the hardware optimizations on and off. In the following, we first describe some of the current hardware and software locality optimization techniques briefly, give an overview of our integrated strategy, and present the organization of this paper.

1.1 Hardware Techniques

Hardware solutions typically involve several levels of memory hierarchy and further enhancements on each level. Research groups have proposed smart cache control mechanisms and novel cache architectures that can detect program access patterns at run-time and can fine-tune some cache policies so that the overall cache utilization and data locality are maximized. Among the techniques proposed are victim caches [10], column-associative caches [1], hardware prefetching mechanisms, cache bypassing using memory address table (MAT) [8, 9], dual/split caches [7], and multi-port caches.

1.2 Software Techniques

In the software area, there is considerable work on compiler-directed data locality optimizations. In particular, loop restructuring techniques are widely used in optimizing compilers [6]. Within this context, transformation techniques such as loop interchange [13], iteration space tiling [13], and loop unrolling have already found their ways into commercial compilers. More recently, alternative compiler optimization methods, called data transformations, which change the memory layout of data structures, have been introduced [12]. Most of the compiler-directed approaches have a common limitation: they are effective mostly for applications whose data access patterns are analyzable at compile time, for example, array-intensive codes with regular access patterns and regular strides.

1.3 Proposed Hardware/Software Approach

Many large applications, however, in general exhibit a mix of regular and irregular patterns. While software optimizations are generally oriented toward eliminating capacity misses coming from the regular portions of the codes, hardware optimizations can, if successful, reduce the number of conflict misses significantly.

These observations suggest that a combined hardware/compiler approach to optimizing data locality may yield better results than a pure hardware-based or a pure software-based approach, in particular for codes whose access patterns change dynamically during execution. In this paper, we investigate this possibility. In particular, we make the following major contributions:

• We present a compiler method that analyzes a given large program and divides it into regions, each of which can be optimized either using a software-based approach or using a hardware-based approach.

1530-1591/03 $17.00 © 2003 IEEE

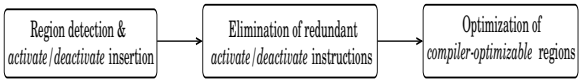

Figure 1. Overview of the compiler-related part of our framework.

• We propose a selective hardware/compiler optimization technique based on the analysis mentioned above.

• We present simulation results indicating that such a selective optimization approach outperforms the pure hardware-based or pure compiler-based approaches in various codes (taken from different benchmarks such as SPEC, TPC, and others). It also outperforms a straightforward combination of hardware and compiler approaches (this corresponds to the case when one uses the most aggressive compiler optimizations provided and the program runs with hardware optimizations turned on all the times).

We believe that the proposed approach fills an important gap in data locality optimization arena and demonstrates how two inherently different approaches can be reconciled and made to work together.

1.4 Paper Organization

The rest of this paper is organized as follows. Section 2 explains our approach for turning the hardware on/off. Section 3 explains the hardware and software optimizations used in this study. In Section 4, we explain the benchmarks used in our simulations. Section 5 reports performance results obtained using a simulator. Finally, in Section 6, we present our conclusions.

2 Program Analysis

2.1 Overview

Our approach combines both compiler and hardware techniques in a single framework. The compiler-related part of the approach is depicted in Figure 1. It starts with a region detection algorithm (Section 2.2) that divides an input program into uniform regions. This algorithm marks each region with special *activate/deactivate* (ON/OFF) instructions that activate/deactivate a hardware optimization scheme *selectively* at run-time. Then, we use an algorithm which detects and eliminates redundant activate/deactivate instructions. Subsequently, the software optimizable regions are handled by the compiler-based locality optimization techniques. The remaining regions, on the other hand, are handled by the hardware optimization scheme at run-time. These steps are detailed in the following sections.

2.2 Region Detection

In this section, we present a compiler algorithm that divides a program into disjoint regions, preparing them for subsequent analysis. The idea is to detect *uniform regions*, where 'uniform' in this context means that the memory accesses in a given region can be classified as either regular (i.e., compile-time analyzable/optimizable), as in array-intensive embedded image/video codes or irregular (i.e., not analyzable/optimizable at compile time), as in non-numerical codes (and also in numerical codes where pattern cannot be statically determined, e.g., subscripted array references). Our algorithm works its way through loops in the nests from the innermost to the outermost, determining whether a given region should be optimized by hardware or compiler. This processing order is important as the innermost loops are in general the dominant factor in deciding the type of the access patterns exhibited by the nests.

The smallest region in our framework is a single loop. The idea behind region detection can be best illustrated using an example. Consider Figure 2(a). This figure shows a schematic representation of a nested-loop hierarchy in which the head of each loop is marked with its depth (level) number, where the outermost loop has a depth of 1 and the innermost loop has a depth of 4. It is clear that the outermost loop is imperfectly-nested as it contains three inner nests at level 2. Figure 2(b) illustrates how our approach proceeds.

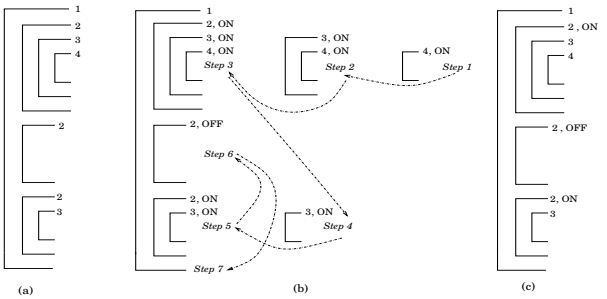

Figure 2. An example that illustrates the region detection algorithm. (a) Schematic representation of a nested-loop hierarchy. (b) Steps taken by our approach to insert ON/OFF instructions. (c) The resulting structure after the elimination of redundant ON/OFF instructions.

We start with the innermost loops and work our way out to the outermost loops. First, we analyze the loop at level 4 (as it is the innermost), and considering the references it contains, we decide whether a hardware approach or a compiler approach is more suitable (Step 1). Assume for now, without loss of generality, that a hardware approach is more suitable for this loop (how to decide this will be explained later). After placing this information on its loop header (in the form of an activate (ON) instruction), we move (Step 2) to the loop at level 3 which encloses the loop at level 4. Since this loop (at level 3) contains only the loop at level 4, we propagate the preferred optimization method of the loop at level 4 to this loop. That is, if there are memory references, inside the loop at level 3 but outside the loop at level 4, they will also be optimized using hardware. In a similar vein, we also decide to optimize the enclosing loop at level 2 using hardware (Step 3). Subsequently, we move to loop at level 1. Since this loop contains the loops other than the last one being analyzed, we do not decide at this point whether a hardware or compiler approach should be preferred for this loop at level 1.

We now proceed with the loop at level 3 in the bottom (Step 4). Suppose that this and its enclosing loop at level 2 (Step 5) are also to be optimized using a hardware approach. We move to the loop at level 2 in the middle (Step 6), and assume that after analyzing its access pattern we decide that it can be optimized by compiler. We mark its loop header with this information (using a deactivate (OFF) instruction). The leftmost part of Figure 2(b) shows the situation after all ON/OFF instructions have been placed.

Since we have now processed all the enclosed loops, we can analyze the loop at level 1 (Step 7). Since this loop contains loops with different preferred optimization strategies (hardware and compiler), we cannot select a unique optimization strategy for it. Instead, we need to switch from one technique to another as we process its constituent loops: suppose that initially we start with a compiler approach (i.e., assuming that as if the entire program is to be optimized in software), when we encounter with loop at level 2 at the top position, we activate (using a special instruction) the hardware locality optimization mechanism (explained later on). When we reach the middle loop at level 2, we deactivate the said mechanism, only to reactivate it just above the loop at level 2 at the bottom of the figure. This step corresponds to the elimination of redundant activate/deactivate instructions in Figure 1. We do not present the details and the formal algorithm due to lack of space. In this way, our algorithm partitions the program into regions, each with its own preferred method of locality optimization, and each is delimited by activate/deactivate instructions which will activate/deactivate a hardware data locality optimization mechanism at run-time. The resulting code structure for our example is depicted in Figure 2(c).

The regions that are to be optimized by compiler are transformed statically at compile-time using a locality optimization scheme (Section 3.2). The remaining regions are left unmodified as their locality behavior will be improved by the hardware during run-time (Section 3.1). Later in the paper, we discuss why it is not a good idea to keep the hardware mechanism on for the entire duration of the program (i.e., irrespective of whether the compiler optimization is used or not).

An important question now is what happens to the portions of the code that reside within a large loop but are sandwiched between two nested-loops with different optimization schemes (hardware/compiler)? For example, in Figure 2(a), if there are statements between the second and third loops at level 2 then we need to decide how to optimize them. Currently, we assign an optimization method to them considering their references. In a sense they are treated as if they are within an imaginary loop that iterates only once. If they are amenable to compiler-approach, we optimize them statically at compile-time, otherwise we let the hardware deal with them at run-time.

2.3 Selecting an Optimization Method for a Loop

We select an optimization method (hardware or compiler) for a given loop by considering the references it contains. We divide the references in the loop nest into two disjoint groups, analyzable (optimizable) references and non-analyzable (not optimizable) references. If the ratio of the number of analyzable references inside the loop and the total number of references inside the loop exceeds a pre-defined threshold value, we optimize the loop in question using the compiler approach; otherwise, we use the hardware approach.

Suppose `i`, `j`, and `k` are loop indices or induction variables for a given (possibly nested) loop. The analyzable references are the ones that fall into one of the following categories:

- scalar references, e.g., `A`
- affine array references, e.g., `B[i]`, `C[i+j][k-1]`

Examples of non-analyzable references, on the other hand, are as follows:

- non-affine array references, e.g., `D[i`^{2}`][j]`, `E[i/j]`, `F[3][i*j]`
- indexed (subscripted) array references, e.g., `G[IP[j]+2]`
- pointer references, e.g., `*H[i]`,`*I`
- struct constructs, e.g., `J.field`, `K->field`

Our approach checks at compile-time the references in the loop and calculates the ratio mentioned above, and decides whether compiler should attempt to optimize the loop. After an optimization strategy (hardware or compiler) for the innermost loop in a given nested-loop hierarchy is determined, the rest of the approach proceeds as explained in the previous subsection (i.e., it propagates this selection to the outer loops).

3 Optimization Techniques

In this section, we explain the hardware and software optimizations used in our simulations.

3.1 Hardware Optimization

The approach to locality optimization by hardware concentrates on reducing conflict misses and their effects. Data accesses together with low set-associativity in caches may exhibit substantial conflict misses and performance degradation. To eliminate the costly conflict misses, we use the strategy proposed by Johnson and Hwu [8, 9]. This is a selective variable size caching strategy based on the characteristics of accesses to memory locations. The principle idea behind the technique presented in [8] is to avoid such misses by not caching the memory regions with low access frequency, thereby keeping the highly accessed regions of the memory in cache. And in the case where spatial locality is expected for the fetched data, fetch larger size blocks. The overall approach is illustrated in Figure 3.

The scheme has two crucial parts – **(1)** a mechanism to track the access frequency of different memory locations and detect

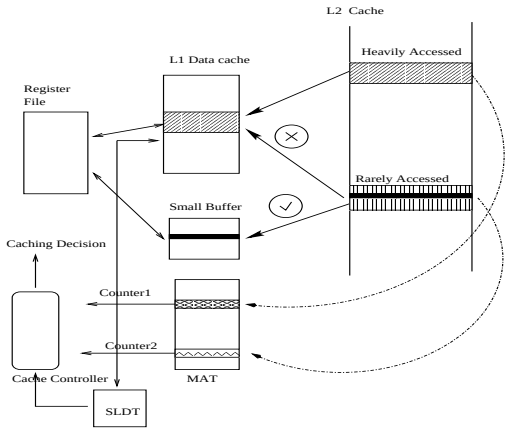

Figure 3. Hardware locality optimization scheme.

spatial locality, and **(2)** a decision logic to assist the cache controller to make caching decisions based on the access frequencies and spatial locality detection. To track the frequency of access to memory locations, the memory is divided into groups of adjacent cache blocks, called macro-blocks [8]. A Memory Access Table (MAT) captures the access frequencies of the macro-blocks. An additional table called Spatial Locality Detection Table (SLDT) is used to detect spatial locality. Each entry in the SLDT tracks spatial hits and misses for the block and stores this information by incrementing or decrementing the Spatial Counter. Exact mechanism of detecting spatial hits can be found in [9]. It is also possible to use victim caches to reduce the number of conflict misses. In this approach, a small fully-associative cache is used for the blocks to be replaced from the cache. If a block is to be replaced, then it is put in the victim cache. If a request comes to the block while it is still residing in the victim cache, the request is completed without going to the next level in memory hierarchy. Detailed information about victim caches can be found in [10].

3.2 Compiler Optimization

Compiler techniques for optimizing cache locality use loop and data transformations. In this section, we revise a technique that optimizes regular nested loops to take advantage of a cache hierarchy. The compiler optimization methodology used in this work is as follows:

- Using affine loop and data transformations, we first optimize temporal and spatial locality aggressively.
- We then optimize register usage through unroll-and-jam and scalar replacement.

For the first step, we use an extended form of the approach presented in [5]. We have chosen this method for two reasons. First, it uses both loop and data transformations, and is more powerful than pure loop ([13]) and pure data ([12]) transformation techniques. Secondly, this transformation framework was readily available to us. It should be noted, however, that other locality optimization approaches such as [12] would result in similar output codes for the regular, array-based programs in our experimental suite. The second step is fairly standard and its details can be found in [4]. A brief summary of this compiler optimization strategy follows. Consider the following loop nest:

```
for(i=1;i<=N;i++)
  for(j=1;j<=N;j++)
    U[j] = V[j][i] + W[i][j];
```

The approach detects that the loop `j` (and consequently the loop `i`) can be optimized by the compiler. Informally, the approach optimizes the nest (containing the loops `i` and `j`) as follows. It first determines the intrinsic temporal reuse in the nest. In this example, there is temporal reuse for only the ref-

Table 1. Base processor configuration.

Issue width	4
L1 (data) size	32K, 4-way set-associative, 32-byte blocks
L1 (instruction) size	32K, 4-way set-associative, 32-byte blocks
L2 size	512K, 4-way set-associative, 128-byte blocks
L1 access time	2 cycle
L2 access time	10 cycles
Memory access time	100 cycles
Memory bus width	8 bytes
Number of memory ports	2
Number of RUU entries	64
Number of LSQ entries	32
Branch prediction	bi-modal with 2048 entries
TLB (data) size	512K, 4-way associative
TLB (instruction) size	256K, 4-way associative

erence `U[j]`;[1] the other references exhibit only spatial reuse [13]. Therefore, in order to exploit the temporal reuse in the innermost position, the loops are interchanged, making the loop `i` innermost. Now this loop accesses the array `V` along the rows, and the array `W` along the columns. These access patterns result in selection of a row-major memory layout for the array `V` and a column-major memory layout for the array `W`. Once the memory layouts have been determined, implementing them in a compiler that uses a fixed default layout for all arrays (e.g., row-major in C) is quite mechanical [12]. After this step, depending on the architectural model (e.g., number of registers, pipeline structure, etc.), the compiler applies scalar placement and unroll-and-jam.

4 Methodology

4.1 Setup

The SimpleScalar [3] processor simulator was modified to carry out the performance evaluations. SimpleScalar is an execution-driven simulator, which can simulate both in-order and out-of-order processors. The baseline processor configuration for our experiments is described in Table 1. The simulator was modified to model a system with the hardware optimization schemes described in Section 3.1. The bypass buffer is a fully–associative cache with 64 double words and uses LRU replacement policy. The MAT has $4,096$ entries and the macro-block sizes are set to 1 KB (as in [8]). When simulating victim caches, we used a victim cache of 64 and 512 entries for level 1 and level 2 caches, respectively [10]. In addition, a flag indicated whether to apply the hardware optimization or not. The instruction set was extended to include activate/deactivate instructions to turn this optimization flag ON/OFF. When the hardware optimization is turned off, we simply ignore the mechanism.

After extensive experimentation with different threshold values, a threshold value (Section 2.3) of 0.5 was selected to determine whether a hardware or a compiler scheme needs to be used for a given inner loop. In the benchmarks we simulated, however, this threshold was not so critical, because in all the benchmarks, if a code region contains irregular (regular) access, it consists mainly of irregular (regular) accesses (between 90% and 100%). *In all the results presented in the next section, the performance overhead of ON/OFF instructions have also been taken into account.*

4.2 Benchmarks

Our benchmark suite represents programs with a very wide range of characteristics. We have chosen three codes from SpecInt95 benchmark suite (*Perl*, *Compress*, *Li*), three codes from SpecFP95 benchmark suite (*Swim*, *Applu*, *Mgrid*), one code from SpecFP92 (*Vpenta*), and six other codes from several benchmarks: *Adi* from Livermore kernels, *Chaos*, *TPC-C*, and three queries from *TPC-D* (*Q1*, *Q3*, *Q6*) benchmark suite. For the TPC benchmarks, we implemented a code segment performing the necessary operations (to execute query).

[1] *This reuse is carried by the outer loop* `i`. *The locality optimizations in general try to put as much of the available reuse as possible into the innermost loop positions.*

An important categorization for our benchmarks can be made according to their access patterns. By choosing these benchmarks, we had a set that contains applications with regular access patterns (*Swim*, *Mgrid*, *Vpenta*, and *Adi*), a set with irregular access patterns (*Perl*, *Li*, *Compress*, and *Applu*), and a set with mixed (regular + irregular) access patterns (remaining applications). Note that there is a high correlation between the nature of the benchmark (floating-point versus integer) and its access pattern. In most cases, numeric codes have regular accesses and integer benchmarks have irregular accesses.

Table 2 summarizes the salient characteristics of the benchmarks used in this study, including the inputs used, the total number of instructions executed, and the miss rates $(\%)$. The numbers in this table were obtained by simulating the base configuration in Table 1. For all the programs, the simulations were run to completion. It should also be mentioned, that in all of these codes, the conflict misses constitute a large percentage of total cache misses (approximately between 53% and 72% even after aggressive array padding).

4.3 Simulated Versions

For each benchmark, we experimented with four different versions — **(1)** *Pure Hardware* – the version that uses only hardware to optimize locality (Section 3.1); **(2)** *Pure Software* – the version that uses only compiler optimizations for locality (Section 3.2); **(3)** *Combined* – the version that uses both hardware and compiler techniques for the *entire duration* of the program; and **(4)** *Selective* (*Hardware/Compiler*) – the version that uses hardware and software approaches *selectively* in an integrated manner as explained in this paper (our approach).

4.4 Software Development

For all the simulations performed in this study, we used two versions of the software, namely, the base code and the optimized code. Base code was used in simulating the pure hardware approach. To obtain the base code, the benchmark codes were transformed using an optimizing compiler that uses aggressive data locality optimizations. During this transformation, the highest level of optimization was performed (`O3`). The code for the pure hardware approach is generating by turning of the data locality (loop nest optimization) using a compiler flag.

To obtain the optimized code, we first applied the data layout transformation explained in Section 2.1. Then, the resulting code was transformed using the compiler, which performs several locality-oriented optimizations including tiling and loop-level transformations. The output of the compiler (transformed code) is simulated using SimpleScalar. It should be emphasized that the pure software approach, the combined approach, and the selective approach all use the same optimized code. The only addition for the selective approach was the (ON/OFF) instructions to turn on and off the hardware. To add these instructions, we first applied the algorithm explained in Section 2 to mark the locations where (ON/OFF) instructions to be inserted. Then, the data layout algorithm was applied. The resulting code was then transformed using the compiler. After that, the output code of the compiler was fed into SimpleScalar, where the instructions were actually inserted in the assembly code.

5 Performance Results

All results reported in this section are obtained using the cache locality optimizer in [8, 9] as our hardware optimization mechanism. The detailed results obtained using a victim cache strategy as our hardware mechanism are omitted due to lack of space, but they are summarized later in Table 3 and in Section 5.2.

5.1 Results for Cache Bypassing

Figure 4 shows the improvement in terms of execution cycles for all the benchmarks in our base configuration. The improvements reported here are relative to the base architecture, where

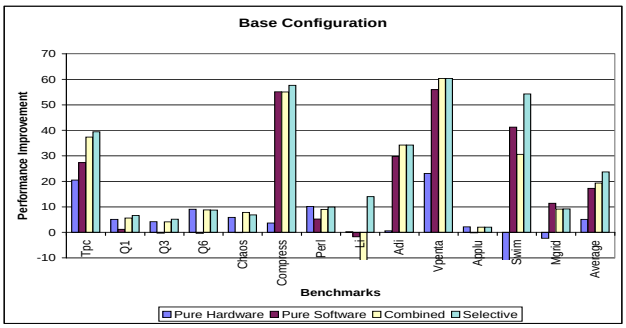

Figure 4. Base configuration.

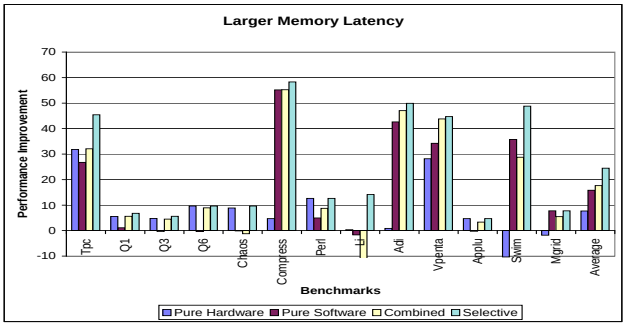

Figure 5. Larger memory latency (200 cycles).

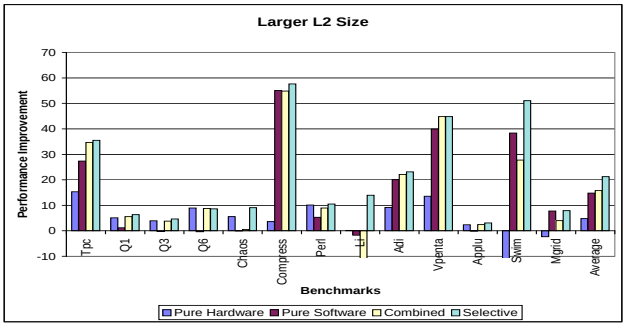

Figure 6. Larger L2 cache size (1 MB).

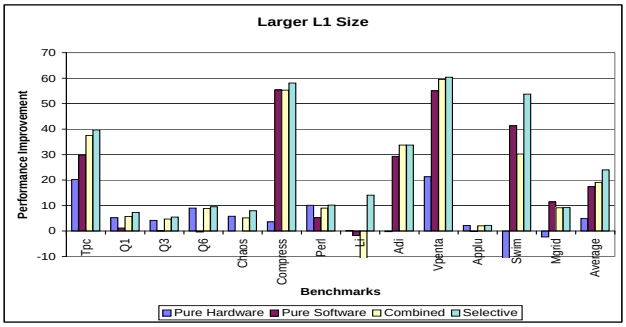

Figure 7. Larger L1 cache size (64 KB).

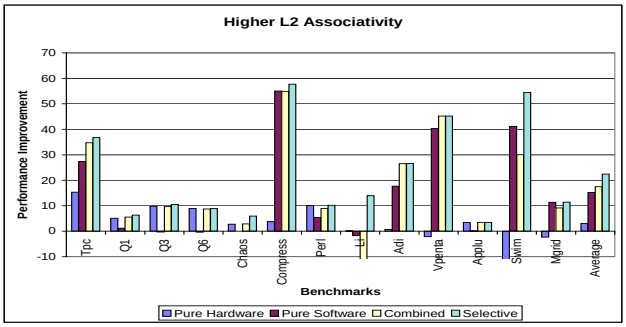

Figure 8. Higher L2 associativity (8).

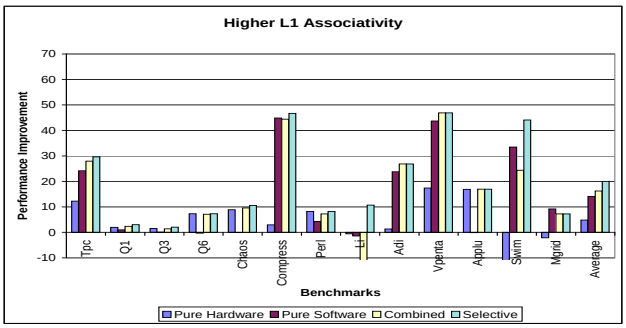

Figure 9. Higher L1 associativity (8).

Table 2. Benchmark characteristics. 'M' denotes millions.

Benchmark	*Input*	*Number of Instructions Executed*	*L1 Miss Rate [%]*	*L2 Miss Rate [%]*
Perl	primes.in	11.2M	2.82	1.6
Compress	training	58.2M	3.64	10.07
Li	train.lsp	186.8M	1.95	3.73
Swim	train	877.5M	3.91	14.42
Applu	train	526.0M	5.05	13.22
Mgrid	mgrid.in	78.7M	4.51	3.34
Chaos	mesh.2k	248.4M	7.33	1.82
Vpenta	Large enough to fill L2	126.7M	52.17	39.79
Adi	Large enough to fill L2	126.9M	25.02	53.49
TPC-C	Generated using TPC tools	16.5M	6.15	12.57
TPC-D,Q1	Generated using TPC tools	38.9M	9.85	4.74
TPC-D,Q3	Generated using TPC tools	67.7M	13.62	5.44
TPC-D,Q6	Generated using TPC tools	32.4M	4.20	10.98

the base code was simulated using the base processor configuration (Table 1).

As expected, the pure hardware approach yields its best performance for codes with irregular access. The average improvement of the pure hardware is 5.07%, and the average improvement for codes with regular access is only 2.19%. The average improvement of pure software approach, on the other hand, is 16.12%. The pure software approach does best for codes with regular access (averaging on a 26.63% percent improvement). The improvement due to the pure software approach for the rest of the programs is 9.5%. The combined approach improves the performance by 17.37% on the average. The average improvement it brings for codes with irregular access is 13.62%. The codes with regular access have an average improvement of 24.36% when the combined approach is employed.

Although, the naively combined approach performs well for several applications, it does not always result in a better performance. These results can be attributed to the fact that the hardware optimization technique used is particularly suited for irregular access patterns. A hardware mechanism designed for a set of applications with specific characteristics can adversely affect the performance of the codes with dissimilar locality behavior. In our case, the codes that have locality despite short term irregular access patterns are likely to benefit from the hardware scheme. The core data structures for the integer benchmarks have such access patterns. However, codes with uniform access patterns, like numerical codes are not likely to gain much out of the scheme. This is because these codes exhibit high spatial reuse which can be converted into locality by an optimizing compiler, or can be captured by employing a larger cache. The pure software approach has, on the other hand, its own limitations. While it is quite successful in optimizing locality in codes with regular access such as *Adi*, *Vpenta*, and *Swim*, (averaging 33.3%) average improvement it brings in codes with high percentage of irregular accesses is only 0.8%.

Our selective approach has an average improvement of 24.98%, which is larger than the sum of the improvements by the pure-hardware and pure-software approaches. Note that, although the combined approach performs well, our selective

approach has better or (at least) the same performance for all the benchmarks. On the average, the selective strategy brings a 7.61% more improvement than the combined strategy.

Why does the selective approach increase the performance? The main reason is that many programs have a *phase-by-phase nature.* History information is useful as long as the program is within the same phase. But, when the program switches to another phase, the information (used by the hardware mechanism employed) about the previous phase slows down the program until this information is replaced (i.e., until new information is collected). If this phase is not long enough, the hardware optimization actually increases the execution cycles for the current phase. Intelligently turning off the hardware eliminates this problem and brings significant improvements.

To evaluate the robustness of our selective approach, we also experimented with different memory latencies, cache sizes, and associativities. The average improvements with these variations are summarized in Table 3. In the following, we discuss the sensitivity of our approach and the other versions to each hardware configuration in detail.

Figure 5 shows the effect of increased memory latency. It gives the percentage improvement in execution cycles where the cost of accessing the main memory is increased to 200 cycles. The rest of the configuration is similar to that given in Table 1. As expected, the improvements brought by our scheme have increased. On the average, the selective scheme improved the performance by 28.52%, 22.27%, and 19.94% for integer, numerical and mixed codes, respectively.

Figure 6 gives the results for larger L2 size. In the experiments, the L2 size is increased to 1 MB. The rest of the parameters remains as in Table 1. Our selective strategy brings a 22.25% improvement over the base configuration. Although, it may look like the improvement has dropped, this is not the case. When we look at the relative improvement versus the pure hardware, pure software and combined approaches, we see that the relative performance remains the same. Figure 7 shows the percentage improvement in execution cycles when the size of the L1 data cache of the machine described in Table 1 is increased to 64K. On the average, the selective optimization strategy brings a 24.17% improvement.

Figure 8 shows the percentage improvement in execution cycles when the associativity of L2 data cache of the machine described in Table 1 is increased to 8, keeping its size constant at 512K. We note that although the overall impact of our approach decreases with the increased associativity, it still performs the best. It improves the performance by average 21.22%, which is 3.5% better than the combined approach. Similarly, Figure 9 gives the results for improved L1 cache associativity. Increasing L1 associativity has an effect similar to increasing L2 associativity. The second, third, fourth, and fifth columns in Table 3 give the average improvements due to the pure software, pure hardware, combined, and selective approaches, respectively, for different hardware configurations.

5.2 Results for Victim Cache

All the simulations performed for cache bypassing were also performed with the victim caches. Although victim caches performed better than the cache bypassing method for some benchmarks, the average improvement is less with victim caches. Victim caches, on the other hand, performed always better than the base configuration (see the sixth column in Table 3), whereas the cache bypassing decreased the performance up to a 12% for some ill cases. The difference between the naively combined approach and our selective approach was also usually less in victim caches. This is due to the low overhead of victim caches. The average improvements due to the combined and selective strategies when they are used in conjunction with victim cache are given in columns seven and eight of Table 3.

The results show that, we can improve the performance of even a passive method like victim caches. So, we do not only decrease the penalty of the hardware method for the applications that it is not suitable for, but we also increase the benefits of the hardware method for many applications. This can be explained by the following scenario. Assume that there is a nest that contains two "for loops", one of them being larger then the other. When we run the hardware for both of the loops, the smaller for loop will be able to evict the elements in the victim cache from the larger for loop. Since the loop is small, the execution of the large loop is going to start before we can take advantage of the data in victim cache. But, if we turn the victim cache off for the small loop, the elements of the large loop will remain in the victim cache, reducing the amount of conflict misses in the victim cache. This, in turn, increases the performance of the victim cache.

Table 3. Average improvements.

Experiment	Pure Software	Cache Bypass	Combined (bypass+ software)	Selective (bypass+ software)	Victim Caches	Combined (victim+ software)	Selective (victim+ software)
Base Confg.	16.12	5.07	17.37	24.98	1.38	16.45	23.82
Higher Mem. Lat.	15.82	7.69	17.66	26.07	4.52	16.24	24.88
Larger L2 Size	14.81	4.75	15.79	22.25	0.80	14.05	20.10
Larger L1 Size	17.42	4.94	17.04	24.17	1.16	16.45	22.55
Higher L2 Asc.	14.05	4.82	15.00	21.22	0.92	13.12	19.39
Higher L1 Asc.	13.96	3.96	14.51	20.93	2.14	12.06	19.21

6 Conclusions

In this paper, we presented a selective cache locality optimization scheme that utilizes both an optimizing compiler technique and a hardware-based locality optimization scheme. This scheme combines the advantages of both the schemes. Our simulation results also confirm this. Also, under different architectural parameters, our scheme consistently gave the best performance.

References

[1] A. Agarwal and S. D. Pudar. Column–associative caches: a technique for reducing the miss rate of direct–mapped caches. In *Proc. 20th Annual International Symposium on Computer Architecture*, San Diego, CA, May 1993.

[2] N. An et al. Analyzing energy behavior of spatial access methods for memory-resident data. In *Proc. 27th International Conference on Very Large Data Bases*, pp. 411–420, Roma, Italy, September, 2001.

[3] D. C. Burger and T. M. Austin. The SimpleScalar tool set, version2.0. *Technical Report CS-TR-97-1342,University of Wisconsin,Madison*, June 1998.

[4] D. Callahan, S. Carr, and K. Kennedy. Improving register allocation for subscripted variables. In *Proc. SIGPLAN Conference on Programming Language Design and Implementation*, ACM, New York.

[5] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee. Improving locality using loop and data transformations in an integrated framework. In *Proc. 31st International Symposium on Micro-Architecture (MICRO'98)*, Dallas, TX, Dec. 1998.

[6] F. Catthoor, S. Wuytack, E. D. Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle. Custom memory management methodology – exploration of memory organization for embedded multimedia system design. *Kluwer Academic Publishers*, June, 1998.

[7] A. Gonzalez, C. Aliagas, and M. Valero. A data cache with multiple caching strategies tuned to different types of locality. In *Proc. ACM International Conference on Supercomputing*, Barcelona, Spain, pages 338–247, July 1995.

[8] T. L. Johnson and W. W. Hwu. Run-time adaptive cache hierarchy management via reference analysis. In *Proc. the 24th International Symposium on Computer Architecture*, June 2–4, 1997.

[9] T. L. Johnson, M. C. Merten, and W. W. Hwu Run-time spatial locality detection and optimization. In *Proc. the 30th International Symposium on Microarchitecture*, December 1–3, 1997.

[10] N. P. Jouppi. Improving direct–mapped cache performance by the addition of a small fully–associative cache and prefetch buffers. In *Proc. 17th Annual International Symposium on Computer Architecture*, Seattle, WA, May 1990.

[11] W. Li. *Compiling for NUMA Parallel Machines*. Ph.D. Thesis, Department of Computer Science, Cornell University, Ithaca, New York, 1993.

[12] M. O'Boyle and P. Knijnenburg. Non-singular data transformations: Definition, validity, applications. In *Proc. 6th Workshop on Compilers for Parallel Computers*, pages 287–297, Aachen, Germany, 1996.

[13] M. Wolf and M. Lam. A data locality optimizing algorithm. In *Proc. ACM Symp. on Programming Language Design and Implementation*, pages 30–44, June 1991.