A Selective Hardware/Compiler Approach for Improving Cache Locality

Gokhan Memik, Mahmut Kandemir*, Malay Haldar, and Alok Choudhary
Center for Parallel and Distributed Computing
Northwestern University
Evanston, IL 60208-3118

Abstract

The widening gap between processor and memory speeds renders data locality optimization a very important issue in compilers. Throughout the years hardware designers and compiler writers focused on optimizing data cache locality using intelligent cache management mechanisms and program-level transformations, respectively. While pure compiler-oriented techniques are quite successful in optimizing codes with regular memory access patterns, they are less successful at improving the locality behavior of non-numerical (integer) programs. On the contrary, the hardware techniques such as cache bypassing and dual/split caches are effective in codes with irregular access patterns. Until now, there has not been significant amount of research investigating the interaction between these optimizations. With this work, we try to fill this gap. We also propose a selective hardware/compiler strategy to optimize cache locality for integer, numerical, 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 the unoptimizable regions with special instructions that activate a state-of-the-art hardware 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 non-selective hardware/compiler approach.

1 Introduction

Memory performance is crucial for high performance processors. As processor speeds increase, low latency and high-bandwidth access to memory becomes more critical. Memory hierarchies with multiple levels of caches are used to improve locality of access so that data can be supplied from the level closest to the processor. This basic architecture provides good performance to a certain extent. In order to improve the performance further, 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 attempt to reorder data access patterns (e.g., loop transformations, tiling, etc.) so that data reuse is maximized to enhance locality. Each approach has its strengths and work well for the patterns they are designed for.

*Dept. of Computer Science and Engineering, The Pennsylvania State University, University Park, PA 16802.
So far each of these approaches has primarily existed independently of one another. For example, a compiler loop restructuring optimization may not really consider the existence of a victim cache or consider its interaction with the optimizations. Similarly, a locality enhancing hardware technique does not normally consider what software optimizations have already been incorporated into the code, because they are based on the addresses generated by the processor, which already takes 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 gain at run-time. If information about this global view can be given to the hardware, the performance of the system can be increased significantly.

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? Are there simple and cost-effective (to implement) ways to combine them? Can one scheme interfere with another, if applied independently; that is, does it result in performance degradation as compared to using either one scheme only or no scheme at all?

To answer these questions, we use state-of-the-art hardware and software locality optimization techniques and study the interaction between these optimizations. Then, we propose a scheme which selectively applies one of the optimizations and turns the other off. Our goal is to combine existing state-of-the-art hardware and software schemes intelligently and take full advantage of both approaches. To achieve this goal, we also propose a region detection algorithm, which is based on the compiler analysis and determines which regions of the programs are suitable for hardware optimization scheme and subsequently, which regions are suitable for software scheme. Based on this analysis it turns the hardware optimizations on and off. In the following we first describe some of the hardware and software locality optimization techniques briefly. Then, we give a summary of the paper.

1.1 Hardware Techniques

Hardware solutions typically involve several levels of memory hierarchy and further enhancements on each level [14]. 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 locality are maximized. Among the techniques proposed are victim caches [18], column-associative caches [1], hardware prefetching [3, 32, 22], cache bypassing using memory address table (MAT) [16, 17], dual/split caches [26, 13], skewed-associative caches [37, 4], multi-port caches [27, 28] and careful page placement techniques [38].

1.2 Software Techniques

In the software area, there is a considerable work on compiler-directed data locality optimizations. In particular, loop restructuring techniques are widely used in optimizing compilers [47]. Within this context, loop-level transformations such as loop interchange [45], tiling [21, 10, 45, 20, 46], and loop unrolling [8] have already found their ways in commercial compilers [46]. More recently, alternative compiler optimization methods, called data transformations, which change the memory layout of data structures, have been introduced [31, 9, 15, 6, 42].
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, codes including regular access patterns, regular strides, and others.

### 1.3 Proposed Hardware/Software Technique

Many applications, however, in general exhibit a mix of regular and irregular patterns (for example, [30, 36]). While software optimizations are generally oriented towards eliminating capacity misses coming from regular portions of the codes, hardware optimizations can reduce the number of conflict misses.

*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 attempt to investigate this possibility. In particular, we make the following contributions:

- We present a compiler method that analyzes a given program and divides it into regions, each of which can be optimized either using a compiler approach (at compile-time using state-of-the-art compiler optimizations) or using a hardware approach (at run-time).
- 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 (taken from different benchmarks such as SPEC, TPC, and others) codes. It also outperforms the straightforward combination of hardware and compiler approaches (this corresponds to the case when one uses most 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. In that respect, we believe that the results presented in this paper provide a direction to start with to consider hardware/software co-optimizations.

*Our goal is to demonstrate that, by having a slight change in the hardware (on/off switch to enable or disable hardware optimizations) and a slightly more intelligent software, a selective hardware/compiler approach yields the best results for a variety of codes. We devise an approach to determine when to selectively turn hardware optimization on/off and show how to accomplish this. Using simulations on various codes, we demonstrate that the selective scheme provides better performance in most cases. We also show the results for a straightforward combination of the pure-hardware and pure-software approaches.*

The remainder of this paper is organized as follows. Section 2 explains our approach for turning the hardware on/off. Section 3 explains in detail the hardware and software optimizations used in this study. In Section 4, we explain the benchmarks used in our simulations. Section 5 presents performance results obtained using a
simulator (modified SimpleScalar Simulator [5]). In Section 6, we discuss related hardware and software work in the area of data locality optimization. Finally, in Section 7, we present our conclusions and briefly discuss the future work on this topic.

2 Program Analysis

In this section, we first give an overview of our approach and review data reuse and locality concepts. Then, we present our compiler analysis that decomposes a given program into disjoint regions and identifies which region is suitable for which kind of optimization.

2.1 Overview of Our Approach

In this paper, we present an approach that combines both compiler and hardware techniques in a single framework. The compiler-related part of the approach is depicted in Figure 1. Our approach 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 here means that the memory accesses in a given region can be classified as either regular (analyzable), as in numerical codes) or irregular (not analyzable), 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 (and the locality characteristics) exhibited by the enclosing loops and the program itself.

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 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.

We start with the innermost loops and work our way out to the outermost loops. First, we analyze the loop at level 4, 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 choose 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 shown 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 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.

Let us now assume that the middle loop at level 2 was instead to be optimized by hardware whereas the bottom loop at level 3 was to be optimized by software. In this case, the hardware-optimizable region would be larger, containing the first two loops at level 2 (and, of course, the loops enclosed by them), as these loops now would require the same preferred optimization method. That is, our algorithm tends to minimize the number of switches (activate/deactivate instructions) between regions, rendering their run-time overhead almost negligible.

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 at compile-time, otherwise we let the hardware deal with them at run-time. Of course, this strategy may not be optimal for every case, but we have found that it performs well in practice.

### 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 references and non-analyzable references. If the ratio of number of analyzable references inside the loop and 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. 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., $G[I/P][j]*2$
- 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.

### 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 associativity caches may exhibit substantial conflict misses and performance degradation [33]. To eliminate the costly conflict misses, we use the approach proposed by Johnson and Hwu [16]. This is a selective variable size caching strategy based on the access characteristics to the memory locations. Study of the access characteristics of Spec95 integer benchmarks such as li and compress shows that particular regions of the memory tend to get accessed heavily, while a large part of the memory is relatively sparsely accessed. Also spatial locality in data fetched by different instructions vary across the execution of the program. Memory regions with different access frequencies may map to the same cache block resulting in conflict misses. The principle idea behind the technique presented in [16] 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 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 [16]. The 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. The SLDT has an entry for each of the cache blocks. Each entry tracks spatial hits and misses for the block and stores this information by incrementing or decrementing the spatial counter. When the block is displaced the value of the counter is deposited in the MAT. Exact mechanism of detecting spatial hits can be found in [17].

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 to 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 [18].

3.2 Compiler Optimization

Compiler techniques for optimizing for 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:

1. Using affine loop and data transformations, we first optimize temporal and spatial locality aggressively.

2. 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 [19]. We have chosen this method for two reasons. First, it uses both loop and data transformations, and is more powerful than a pure loop (e.g., [45]) and a pure data (e.g., [31]) transformation technique. Secondly, this transformation framework was readily available to us. It should be noted, however, that other locality optimization approaches such as [9] would result in similar output codes for the regular programs in our experimental suite. As mentioned earlier, our objective is not to evaluate a specific compiler optimization framework. The second step is fairly standard and details can be found in [7] and [8]. A brief summary of this method follows. Consider the following loop nest.
Table 1: Base processor configuration.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Issue width</td>
<td>4</td>
</tr>
<tr>
<td>L1 (data) size</td>
<td>32K, 4-way associative, 32-byte block</td>
</tr>
<tr>
<td>L1 (instruction) size</td>
<td>32K, 4-way associative, 32-byte block</td>
</tr>
<tr>
<td>L2 size</td>
<td>512K, 4-way associative</td>
</tr>
<tr>
<td>L1 access time</td>
<td>2 cycle</td>
</tr>
<tr>
<td>L2 access time</td>
<td>10 cycles</td>
</tr>
<tr>
<td>Memory access time</td>
<td>100 cycles</td>
</tr>
<tr>
<td>Memory bus width</td>
<td>8 bytes</td>
</tr>
<tr>
<td>Number of memory ports</td>
<td>2</td>
</tr>
<tr>
<td>Number of RUU entries</td>
<td>64</td>
</tr>
<tr>
<td>Number of LSQ entries</td>
<td>32</td>
</tr>
<tr>
<td>Branch prediction</td>
<td>bi-modal with 2048 entries</td>
</tr>
<tr>
<td>TLB (data) size</td>
<td>512K, 4-way associative</td>
</tr>
<tr>
<td>TLB (instruction) size</td>
<td>256K, 4-way associative</td>
</tr>
</tbody>
</table>

\[
\text{for}(i=1; i<=N; i++) \\
\quad \text{for}(j=1; j<=N; j++) \\
\quad U[j] = V[j][i] + W[i][j];
\]

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 case there is temporal reuse for only the reference \( U[j] \); the other references exhibit only spatial reuse [45]. 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 [31]. After this step, depending on the architectural model (e.g., number of registers, pipeline structure, etc.), the compiler can apply scalar placement and unroll-and-jam.

Finally, some comments on implementation-related issues follow. To implement the technique presented in this paper, we need two additional instructions to activate/deactivate (ON/OFF) the hardware mechanism. While the instruction set architecture (ISA) can be extended, it is also possible to overload two existent instructions. Another question is whether the entries in the MAT (or in the victim cache) should be removed when the hardware is turned off. If the entries are to be removed, we can use status-bits that are present (but not used) in many of the current architectures. The decision of removing or not removing such entries is not a question of correctness, it is a problem of performance and it merits further research.

\footnote{This reuse is \textit{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.}
4 Methodology

4.1 Setup

The SimpleScalar [5] processor simulator was modified to carry out the performance evaluations. SimpleScalar is an execution-driven simulator, which can simulate aggressive out-of-order processors. The baseline processor configuration for our experiments is described in Table 1. The values for the architectural parameters are similar to those of existing state-of-the-art processors. The simulator was modified to model a system with the hardware optimization schemes described in Section 3.1. The bypass buffer was a fully-associative cache with 64 double words and uses LRU replacement policy. The MAT had 4,096 entries and macro-block sizes are set to 1 KB (as in [16]). When simulating victim caches, we used a victim cache of 64 and 512 entries for level 1 and level 2 caches, respectively [18]. 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 is selected to determine whether a hardware or a compiler scheme needs to be used for a given inner loop. In the benchmarks we have simulated, however, this threshold was not so critical, because in all the benchmarks, if a section contains irregular access, it consists mainly of irregular accesses. Similarly, sections having some regular accesses consist of mostly regular accesses.

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 [36], TPC-C [40], and three queries from TPC-D (Q1, Q3, Q6) benchmark suite [41]. For the TPC benchmarks, we implemented a code segment performing the necessary operations. Note that TPC benchmarks do not force a specific implementation. They dictate the SQL codes and the relations in the database [40, 41]. Although, we did not simulate a specific database management system, the code we have implemented is based on the Minibase Database Management System [44].

The important categorization for the benchmarks is made according to their access patterns. By choosing these benchmarks, we had a set that contains applications with regular access patterns, a set with irregular access patterns, and a set with mixed access patterns. While Swim, Mgrid, Vpenta, and Adi contain mostly regular accesses, Perl, Li, Compress, and Applu contain mostly irregular accesses. The rest of the codes consists of a mix of regular and irregular accesses. Note that there is a high correlation between the nature of the benchmark (floating point or integer) and its access pattern. In most cases, numeric codes have regular accesses and integer benchmarks have irregular accesses. Most floating point benchmarks manipulate big arrays in a sequential or analyzable manner. In Applu, however, the arrays are usually accessed using secondary variables (induction
variables [29]) and functions, which makes it non-analyzable for compilers.

Table 2 on page 14 summarizes the salient characteristics of the benchmarks used in this study, including the inputs used, total number of instructions executed (Instruction Count), memory accesses (Access Count) and 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.

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 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, base code and optimized code. Base code was used in simulating the pure hardware approach. To obtain the base code, the benchmark codes were transformed using SGI Compiler (MIPSpro compiler) [46]. During this transformation, the highest level of optimization was performed (03). For pure hardware approach we want the software approach to have no effect on the performance. Therefore during the translation process we turned off the optimizations related to data locality (loop nest optimizations) using -LNO:opt=0 option.

To obtain the optimized code, we first applied the data layout transformation explained in Section 2.1 (see [19] for details of the compilation framework and implementation). Then, the resulting code was transformed using the SGI Compiler, which performs several optimizations including tiling and loop-level transformations [21, 10]. We have used the highest optimization level (03). The output of the compiler (transformed code) is simulated using SimpleScalar. Pure software approach, combined approach and selective approach use the same optimized code. The only addition for 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 resultant code was then transformed using the SGI compiler. After that, the output code of the SGI compiler was fed into the SimpleScalar, where the instructions were actually inserted in the assembly code. The increase in the compilation time to obtain the optimized codes was always below 10%.

5 Performance Results

All the figures in this in this section correspond to the cache bypassing method. The figures for the victim caches are omitted due to the space limitations. Their behavior is similar to those of the cache bypassing method. The
Figure 7. Larger L1 cache size (The cache size is increased from 16 KB to 32 KB).

Figure 8. Higher L2 Associativity (The associativity of the L2 cache increased from 4 to 8).

Figure 9. Higher L1 Associativity (The associativity of the L1 cache increased from 4 to 8).
results of victim caches are later summarized in Table 3 and in Section 5.2.

When conducting these experiments, we had two ideas. First, we wanted to see how the naively combined approach performs with respect to the pure hardware and pure software approaches, in other words, we would like to see whether the combinations of different optimizations interact constructively or destructively. Secondly, we wanted to compare our scheme with pure hardware and pure compiler approach as well as to compare it with a straightforward combination of hardware and software optimizations. Figures for pure hardware and software optimizations show how amenable the particular benchmark is to respective scheme.

5.1 Results for Cache Bypassing

Figure 4 shows the improvement in terms of execution cycles for all the benchmarks in base configuration. The improvement is relative to the base architecture, where the base code was simulated using the base processor (Table 1). Note that, these codes are equivalent to the codes used by the pure hardware approach.

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 2.19%. The average improvement of pure software approach, on the other hand, is 16.12%. Pure software approach does best for codes with regular access (averaging 26.63% percent improvement). The improvement of the pure software approach for the rest of the programs is 9.5%. The combined approach improves the performance by 17.37% in average. The average improvement for codes with irregular access is 13.7%. The codes with regular access have an average improvement of 25.39% in the combined approach.

Although, the naively combined approach performs good 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

---

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

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Input</th>
<th>Number of instructions executed</th>
<th>Access Count</th>
<th>Access Rate [%]</th>
<th>Miss Count</th>
<th>Miss Rate [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>perl</td>
<td>primes.in</td>
<td>11M</td>
<td>4.8M</td>
<td>0.6</td>
<td>0.21M</td>
<td>1.6</td>
</tr>
<tr>
<td>compress</td>
<td>training</td>
<td>57M</td>
<td>17.5M</td>
<td>3.6</td>
<td>1.0M</td>
<td>10.07</td>
</tr>
<tr>
<td>it</td>
<td>train.lisp</td>
<td>188M</td>
<td>78.6M</td>
<td>1.95</td>
<td>2.7M</td>
<td>3.7</td>
</tr>
<tr>
<td>swim</td>
<td>train</td>
<td>873M</td>
<td>269.7M</td>
<td>3.91</td>
<td>15.6M</td>
<td>14.42</td>
</tr>
<tr>
<td>applu</td>
<td>train</td>
<td>525M</td>
<td>122.1M</td>
<td>5.05</td>
<td>9.0M</td>
<td>13.22</td>
</tr>
<tr>
<td>mgrid</td>
<td>mgrid.in</td>
<td>78M</td>
<td>27.2M</td>
<td>2.3</td>
<td>1.3M</td>
<td>0.2</td>
</tr>
<tr>
<td>chaos</td>
<td>mesh.2k</td>
<td>248M</td>
<td>80.5M</td>
<td>7.3</td>
<td>8.3M</td>
<td>1.8</td>
</tr>
<tr>
<td>tpenta</td>
<td>Large enough to fill L2</td>
<td>128M</td>
<td>33.2M</td>
<td>52.17</td>
<td>26.7M</td>
<td>39.75</td>
</tr>
<tr>
<td>adi</td>
<td>Large enough to fill L2</td>
<td>125M</td>
<td>32.9M</td>
<td>25.02</td>
<td>12.0M</td>
<td>53.49</td>
</tr>
<tr>
<td>TPC-C</td>
<td>Generated using tools provided by TPC</td>
<td>6M</td>
<td>1.4M</td>
<td>6.15</td>
<td>0.2M</td>
<td>12.57</td>
</tr>
<tr>
<td>TPC-D, Q1</td>
<td>Generated using tools provided by TPC</td>
<td>38M</td>
<td>13.5M</td>
<td>3.62</td>
<td>0.7M</td>
<td>0.7</td>
</tr>
<tr>
<td>TPC-D, Q3</td>
<td>Generated using tools provided by TPC</td>
<td>66M</td>
<td>26.8M</td>
<td>9.4</td>
<td>3.2M</td>
<td>2.4</td>
</tr>
<tr>
<td>TPC-D, Q6</td>
<td>Generated using tools provided by TPC</td>
<td>29M</td>
<td>8.6M</td>
<td>2.01</td>
<td>0.5M</td>
<td>10.98</td>
</tr>
</tbody>
</table>

---

2Although not presented here, we should mention that the reductions in cache misses follow a similar trend to those of execution cycles.
used is particularly suited for irregular computation. In general, 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. For example, compress has hash-tables, which are accessed frequently. 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 compiler.

The pure software approach, on the other hand, has 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 0.8%.

The selective approach has an average improvement of 22.94%, which is larger than the sum of the improvements by the pure-hardware and pure-software approaches. It performs 4.74% better than the combined approach. It also performs 5.87% better than the pure software and 17% better than the pure hardware approach. Note that, although the combined approach performs good, our selective approach performs either better than that or has the same performance for all the benchmarks. In 9 of 13 benchmarks, the selective approach gives the best performance. For the other benchmarks, its performance is matched by other approaches. In all the mixed codes, the selective approach has the best performance.

Why does the selective approach increases the performance? 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 actually increases the execution cycles for the phase. Intelligently turning off the hardware eliminates this problem and brings significant improvements.

The results in this section show that we can get the best of the two worlds by simple additions to the software and to the hardware.

### 5.1.1 Sensitivity Analysis

To evaluate the performance of our scheme, we also experimented with different memory latencies, cache sizes, and associativeities. The average improvements are given in Table 3. The relative performance of the selective approach with respect to other simulated approaches is given in Table 4.

The gap between processor clock rates and memory access time is increasing and the trend is expected to continue. Figure 5 shows the effect of increased memory latency. It gives the percentage improvement in execution cycle times where the cost of accessing to 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 average, the selective scheme improved the performance by 28.52%, 21.17%, and 18.76%, for integer, numerical and mixed codes, respectively. The efficiency of our scheme increases with increasing memory latencies. With
Table 3: Average improvements of the benchmarks

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Pure Software</th>
<th>Cache Bypassing</th>
<th>Combined (bypassing+software)</th>
<th>Selective (bypassing+software)</th>
<th>Victim Caches</th>
<th>Combined (victim+software)</th>
<th>Selective (victim+software)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base Configuration</td>
<td>16.12</td>
<td>5.07</td>
<td>17.37</td>
<td>22.94</td>
<td>1.38</td>
<td>16.45</td>
<td>18.82</td>
</tr>
<tr>
<td>Higher Memory Latency</td>
<td>15.82</td>
<td>7.69</td>
<td>17.66</td>
<td>24.42</td>
<td>4.52</td>
<td>18.58</td>
<td>20.57</td>
</tr>
<tr>
<td>Larger L1 Size</td>
<td>14.81</td>
<td>4.75</td>
<td>15.79</td>
<td>21.25</td>
<td>0.80</td>
<td>15.05</td>
<td>16.10</td>
</tr>
<tr>
<td>Larger L2 Size</td>
<td>17.42</td>
<td>4.94</td>
<td>19.13</td>
<td>24.08</td>
<td>1.36</td>
<td>16.45</td>
<td>18.82</td>
</tr>
<tr>
<td>Higher L2 Associativity</td>
<td>14.05</td>
<td>4.82</td>
<td>16.19</td>
<td>20.22</td>
<td>0.92</td>
<td>14.24</td>
<td>15.96</td>
</tr>
<tr>
<td>Higher L1 Associativity</td>
<td>13.96</td>
<td>3.96</td>
<td>15.83</td>
<td>19.93</td>
<td>2.14</td>
<td>14.09</td>
<td>15.75</td>
</tr>
</tbody>
</table>

Table 4: Relative improvement of selective approach with respect to other approaches simulated (for cache bypassing method).

<table>
<thead>
<tr>
<th>Hardware Compared</th>
<th>Base Configuration</th>
<th>Higher Memory Latency</th>
<th>Larger L2 Size</th>
<th>Larger L1 Size</th>
<th>Higher L2 Associativity</th>
<th>Higher L1 Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pure Hardware</td>
<td>17.0</td>
<td>15.5</td>
<td>15.8</td>
<td>18.2</td>
<td>14.69</td>
<td>15.1</td>
</tr>
<tr>
<td>Pure Software</td>
<td>5.8</td>
<td>7.4</td>
<td>5.6</td>
<td>5.6</td>
<td>5.4</td>
<td>5.2</td>
</tr>
<tr>
<td>Combined</td>
<td>4.7</td>
<td>5.7</td>
<td>4.7</td>
<td>4.1</td>
<td>3.5</td>
<td>3.5</td>
</tr>
</tbody>
</table>

With respect to the Combined version, the selective scheme brought a 5.7% improvement on average.

Figure 6 gives the results for larger L2 size. It may seem that, larger secondary caches may reduce the effect of optimizations that reduce misses in secondary caches. But the trends show that the penalty of a cache miss in clock cycles is increasing. Therefore, even if the miss rates will drop, there will be need to reduce them further. Also, decreased miss ratio in L2 caches seems to accentuate the importance of conflict misses in the first level cache. In the experiments, the size is increased to 1 MB. The rest of the parameters remains as in Table 1. Our selective strategy brings 21.25% improvement over the base configuration. Although, it may seem that 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 remain 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 24.08% improvement.

For many hardware optimization schemes, the performance improvement decreases as the associativity of the cache is increased, because of the decrease in the number of conflict misses. Figure 8 shows the percentage improvement in execution cycles when the associativity of L2 data cache of the machine described in Table 1

Table 5: Relative improvement of selective approach with respect to other approaches simulated (for victim cache method).

<table>
<thead>
<tr>
<th>Hardware Compared</th>
<th>Base Configuration</th>
<th>Higher Memory Latency</th>
<th>Larger L2 Size</th>
<th>Larger L1 Size</th>
<th>Higher L2 Associativity</th>
<th>Higher L1 Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pure Hardware</td>
<td>17.2</td>
<td>15.4</td>
<td>15.2</td>
<td>17.5</td>
<td>14.9</td>
<td>13.3</td>
</tr>
<tr>
<td>Pure Software</td>
<td>5.8</td>
<td>7.4</td>
<td>5.6</td>
<td>5.6</td>
<td>5.4</td>
<td>5.2</td>
</tr>
<tr>
<td>Combined</td>
<td>2.0</td>
<td>1.7</td>
<td>0.9</td>
<td>2.0</td>
<td>1.5</td>
<td>1.5</td>
</tr>
</tbody>
</table>
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 20.22%, which is 3.5% better than the combined approach. Similarly, Figure 9 gives the results for improved L1 cache associativity. Again, the other parameters of the cache remain as in Table 1. Increasing L1 associativity has an effect similar to increasing L2 associativity. Still, our scheme results in best performance in most benchmarks and performs at least as good as any other approach in the remaining benchmarks.

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, whereas the cache bypassing decreased the performance up to a 12% for some ill cases. Victim caches also performed good under many architectural parameters (as in the case of increased L1 associativity). 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 are given in Table 3. The improvements relative to other approaches are given in Table 5.

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 example. Assume that there is a nest that contains two for loops, one of them being larger than 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. And we will go into the large for loop, before we can take advantage of the new elements in the 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.

5.3 Summary of Experimental Results

Overall, our selective optimization strategy is quite successful in optimizing cache locality. It improves the performance for different L1 and L2 data cache sizes, latencies, and associativities. In all cases, we were able to achieve the best performance among the simulated systems. Note that the selective scheme is able to take advantage of both hardware and software optimizations and also eliminate the negative impact of the hardware technique on those applications for which it is not suitable.

6 Related Work

Much of the related compiler work on cache locality optimization is based on iteration space transformations. Wolf and Lam [45] define reuse vectors and reuse spaces and show how these concepts can be exploited by an
iteration space optimization technique. McKinley et al. [24] and Li [23] propose different loop transformation frameworks. The loop based locality enhancing techniques also include tiling [45, 21, 10, 20]. The compiler approach used in this work can improve the effectiveness of tiling by making it less sensitive to the tile size.

More recently new techniques based on memory layout transformations are proposed. These techniques focus directly on array layouts and try to modify the layouts such that unit-stride accesses will be obtained in the inner-most loops. O’Boyle and Knijnenburg [31] present different memory layout transformation techniques. Cierniak and Li [9] are among the first to offer an optimization scheme that combines loop and data transformations. The approach used in this work is more powerful than [9], as we consider a larger space for possible transformations. Anderson et al. [2] propose a compiler transformation technique which is more suitable for shared-memory parallel machines. A different trend of compiler work in optimizing locality focuses on eliminating conflict misses. Rivera and Tseng observed that eliminating conflict misses would enable better exploitation of spatial locality and proposed compile-time data transformations [33].

As stated earlier, the compiler-based locality optimization techniques are most useful for numerical codes whose access patterns can be detected statically at compile-time. Hardware locality optimization mechanism can be used to overcome this limitation. Very recently, compiler-techniques that target numerical codes with irregular access patterns have been proposed [25, 11]. Whether these techniques will be competitive with hardware-based approaches remains to be seen.

Locality optimizations by hardware use either characteristics of the load/store instructions making the memory reference or access patterns to the memory regions accessed, to make caching decisions. Rivers and Davidson [34] use extra bits in the cache to tag a particular cache block temporal or no-temporal based of the frequency of accesses. A non-temporal cache block is not placed in the cache the next time it is missed in the cache. Instead it is placed in a small associative buffer. Johnson and Hwu [16] make the identification of non-temporal blocks more adaptive to the computation by keeping reference count of memory regions and making comparisons of the reference counts of conflicting blocks to bypass less frequently accessed data. The scheme was further enhanced to detect spatial locality and support varying fetch sizes [17]. Tyson et al. [43] used the miss characteristics of a load/store to cache (or bypass) the data referenced by the particular load/store. A more elaborate scheme proposed by Gonzales et al. [13] uses the access pattern of loads/stores to invoke one of multiple caching strategies in a miss. Also, several hardware prefetching mechanisms were proposed [3, 22, 12, 35]. Sugumar and Abraham [39] also proposed a cache bypassing scheme where bypassing decisions are made at run-time.

7 Conclusions and Future Work

In this paper, we presented experimental results indicating the interaction between pure-hardware and pure-software approaches. We also presented a selective cache locality optimization scheme that utilizes both a state-of-the-art optimizing compiler technique and two state-of-the-art hardware optimization schemes. As compared with the pure compiler-based or pure hardware-based techniques, our approach brings an important advantage by using the optimization hardware selectively and by utilizing a powerful compiler technique in program regions
where it is most useful. Therefore, it combines the advantages of both. In other words, for the program parts for which the hardware optimization technique is not suitable, our selective approach eliminates its potentially negative impact. Our simulation results also confirm this. Specifically, our approach was able to increase the performance up to a 17.42% with respect to a naively combined approach in cache bypassing method. The average improvement was 4.7% in the base configuration. Also, under different architectural parameters, our scheme was able to give the best performance.

This work can be extended in several ways. First, we would like to make experiments with different hardware optimization schemes embedded in our framework. In fact, it might be possible to employ more than one hardware scheme in the same architecture and depending on the characteristics of the application at hand (that can be detected at compile time) we can activate/deactivate different schemes at run-time. This obviously requires a compiler analysis. Secondly, we can adopt a pure hardware approach but again selectively activate the main mechanism itself. In other words, we can use additional hardware to detect different program regions adaptively and to activate/deactivate some optimization policies at run-time. Another possible usage of the region detection algorithm may be to detect the phases of a program and to inform the hardware schemes of phase changes. This may reduce the phase transition overhead and increase the system performance. We also plan to extend our techniques to work with program region granularities larger than loop nests.

Overall, we believe that the possibility of combining novel cache management mechanisms with the state-of-the-art optimizing compiler technology will lead to interesting research problems.

References


