Run-time Spatial Locality Detection and Optimization*

Teresa L. Johnson Matthew C. Merten Wen-mei W. Hwu
Center for Reliable and High-Performance Computing
University of Illinois
Urbana-Champaign, IL 61801
{tjohnson,merten,hwu}@crhc.uiuc.edu

Technical Report IMPACT-97-02
September 1997

Abstract

As the disparity between processor and main memory performance grows, the number of execution cycles spent waiting for memory accesses to complete also increases. As a result, latency hiding techniques are critical for improved application performance on future processors. In this paper we examine the spatial locality characteristics of several applications, and show that spatial locality varies substantially across and within applications. We then present a microarchitecture scheme which detects and adapts to this varying spatial locality, dynamically adjusting the amount of data fetched on a cache miss. The *Spatial Locality Detection Table*, introduced in this paper, facilitates the detection of spatial locality across adjacent small cached blocks. Results from detailed simulations of several integer programs show significant speedups. The improvements are due to the reduction of conflict and capacity misses by utilizing small blocks and small fetch sizes when spatial locality is absent, and the prefetching effect of large fetch sizes when spatial locality exists.

1 Introduction

This paper introduces an approach to solving the growing memory latency problem [2] by intelligently exploiting *spatial locality*. Spatial locality refers to the tendency for neighboring memory locations to be referenced close together in time. Traditionally there have been two main approaches used to exploit spatial locality. The first approach is to use larger cache blocks, which have a natural prefetching effect. However, large cache blocks can result in wasted bus bandwidth and poor cache utilization, due to fragmentation and underutilized cache blocks. Both negative effects occur when data with little spatial locality is cached. The second common approach is to prefech multiple blocks into the cache. However, prefetching is only beneficial when the prefetched data is accessed in cache, otherwise the prefetched data may displace more useful data from the cache, in addition to wasting bus bandwidth. Similar issues exist with write allocate caches, which, in effect, prefetch the data in the cache block containing the written address. Particularly when using large block sizes and write allocation, the amount of prefetching is fixed. However, the spatial locality, and hence the optimal prefetch amount, varies across and often within programs.

As the available chip area increases, it is meaningful to spend more resources to allow intelligent control over latency-hiding techniques, adapting to the variations in spatial locality. For numeric programs there are several known compiler techniques for optimizing data cache performance. In contrast, integer (non-numeric) programs often have irregular access patterns that the compiler cannot detect and optimize. For example, the temporal and spatial locality of linked list elements and hash table data are often difficult to determine at compile time. This paper focuses on cache performance optimization for integer programs. While we focus our attention on data caches, the techniques presented here are applicable to instruction caches.

In order to increase data cache effectiveness for integer programs we are investigating methods of *adaptive cache hierarchy management*, where we intelligently control caching decisions based on the usage characteristics of accessed data. In this paper we examine the problem of detecting spatial locality in accessed data, and automatically control the fetch of multiple smaller cache blocks into all data caches and buffers. Not only are we able to reduce the conflict and capacity misses with smaller cache lines and fetch sizes when spatial locality is absent, but we also reduce cold start misses and prefetch useful data with larger fetch sizes when spatial locality is present.

We introduce a new hardware mechanism called the *Spatial Locality Detection Table* (*SLDT*). Each SLDT entry tracks the accesses to multiple adjacent cache blocks, facilitating detection of spatial locality across those blocks while they are cached. The resulting information is later recorded in the *Memory Address Table* [3] for long-term tracking of larger regions called *macroblocks*. We show that these extensions to the cache microarchitecture significantly improve the performance of integer applications, achieving up to 17% and 26% improvements for 100 and 200-cycle memory latencies, respectively. This scheme is fully compatible with existing Instruction Set Architectures (ISA).

The remainder of this paper is organized as follows: Section 2 discusses related work; Section 3 discusses general spatial locality issues, and a code example from a common

*This technical report is a longer version of [1].

application is used to illustrate the role of spatial locality and cache line sizes in determining application cache performance, as well as to motivate our spatial locality optimization techniques; Section 4 discusses hardware techniques; Section 5 presents simulation results; Section 6 performs a cost analysis of the added hardware; and Section 7 concludes with future directions.

2 Related Work

Several studies have examined the performance effects of cache block sizes [4][5]. One of the studies allowed multiple consecutive blocks to be fetched with one request [4], and found that for data caches the optimal statically-determined fetch size was generally twice the block size. In this work we also examine fetch sizes larger than the block size, however, we allow the fetch size to vary based on the detected spatial locality. Another method allows the number of blocks fetched on a miss to vary across program execution, but not across different data [6].

Hardware [7][8][9][10][11] and software [12][13][14] prefetching methods for uniprocessor machines have been proposed. However, many of these methods focus on prefetching regular array accesses within well-structured loops, which are access patterns primarily found in numeric codes. Other methods geared towards integer codes [15][16] focus on compiler-inserted prefetching of pointer targets, and could be used in conjunction with our techniques.

The dual data cache [17] attempts to intelligently exploit both spatial and temporal locality, however the temporal and spatial data must be placed in separate structures, and therefore the relative amounts of each type of data must be determined a priori. Also, the spatial locality detection method was tuned to numeric codes with constant stride vectors. In integer codes, the spatial locality patterns may not be as regular. The split temporal/spatial cache [18] is similar in structure to the dual data cache, however, the run-time locality detection mechanism is quite different than that of both the dual data cache and this paper.

3 Spatial Locality

Caches seek to exploit the principle of locality. By storing a referenced item, caches exploit *temporal locality* - the tendency for that item to be rereferenced soon. Additionally, by storing multiple items adjacent to the referenced item, they exploit *spatial locality* - the tendency for neighboring items to be referenced soon. While exploitation of temporal locality can result in cache hits for future accesses to a particular item, exploitation of spatial locality can result in cache hits for future accesses to multiple nearby items, thus avoiding the long memory latency for short-term accesses to these items as well. Traditionally, exploitation of spatial locality is achieved through either larger block sizes or prefetching of additional blocks. We define the following terms as they will be used throughout this paper:

element A data item of the maximum size allowed by the ISA, which in our system is 8 bytes.

spatial reuse A reference to a cached element other than the element which caused the referenced element to be fetched into the cache.

The spatial locality in an application's data set can predict the effectiveness of spatial locality optimizations. Unfortunately, no quantitative measure of spatial locality exists, and we are forced to adopt indirect measures. One indirect measure of the amount of spatial locality is via its inverse relatioship to the distance between references in both space and time. With this in view, we measured the spatial reuses in a 64K-byte fully-associative cache with 32-byte lines. This gives us an approximate time bound (the time taken for a block to be displaced), and a space bound (within 32-byte block boundaries). We chose this block size because past studies have found that 16 or 32-byte block sizes maximize data cache performance [4]. These measurement techniques differ from those in [19], which explicitely measure the reuse distance (in time). Our goal is to measure both the reused and unused portions of the cache blocks, for different cache organizations.

Figure 1(a) shows the spatial locality estimates for the fully-associative cache. The number of dynamic cache blocks is broken down by the number of 8-byte elements that were accessed during each block's cache lifetime. Blocks where only one element is accessed have no spatial locality within the measured context. This graph does not show the relative locations of the accessed elements within each 32-byte cache block. Figure 1(a) shows that between 13-83% of the cached blocks have no spatial reuse. Figure 1(b) shows how this distribution changes for a 16K-byte direct-mapped cache. In this case between 30-93% of the blocks have no spatial reuse.

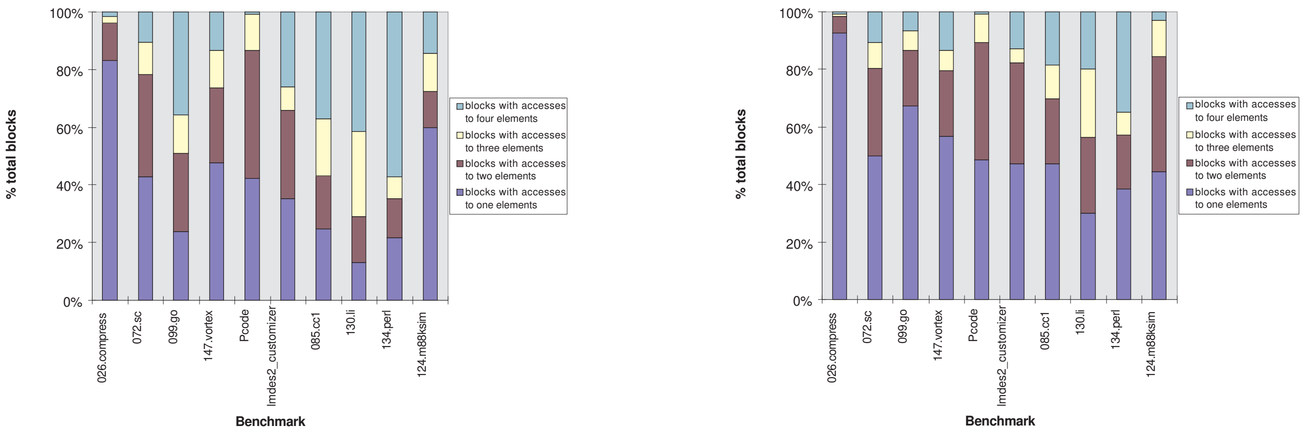
For a 32-byte cache block, over half the time the extra data fetched into the cache simply wastes bus bandwidth and cache space. Similar observations have been made for numeric codes [19]. Therefore, it would be beneficial to tune the amount of data fetched and cached on a miss to the spatial locality available in the data. This optimization is investigated in our work. We discuss several issues involved with varying fetch sizes, including cost efficient and accurate spatial locality detection, fetch size choice, and cache support for varying fetch sizes.

3.1 Code Example

In this section we use a code example from SPEC92 *gcc* to illustrate the difficulties involved with static analysis and annotation of spatial locality information, motivating our dynamic approach.

One of the main data structures used in *gcc* is an RTL expression, or *rtx*, whose definition is shown in Figure 2. Each *rtx* structure contains a two-byte code field, a one-byte mode field, seven one-bit flags, and an array of operand fields. The operand array is defined to contain only one four-byte element, however, each *rtx* is dynamically allocated to contain as many array elements as there are operands, depending on the *rtx* code, or RTL expression type. Therefore, each *rtx* instance contains eight or more bytes.

In the frequently executed *_rtx_renumbered_equal_p* routine, which is used during jump optimization, two *rtx*



(a) 64K-byte fully-associative

(b) 16K-byte direct-mapped

Figure 1: Breakdown of blocks cached in L1 data cache by how many 8-byte elements were accessed while each block was cached. The results for two cache configurations are shown, each with 32-byte blocks.

```
/* RTL expression ("rtx"). */
typedef struct rtx_def
{
  /* The kind of expression this is. */
  enum rtx_code code : 16;
  /* The kind of value the expression has. */
  enum machine_mode mode : 8;
  /* Various bit flags */
  unsigned int jump : 1;
  unsigned int call : 1;
  unsigned int unchanging : 1;
  unsigned int volatil : 1;
  unsigned int in_struct : 1;
  unsigned int used : 1;
  unsigned integrated : 1;
  /* The first element of the operands of this rtx.
     The number of operands and their types are controlled
     by the 'code' field, according to rtl.def. */
  rtunion fld[1];
} *rtx;
```

```
/* Common union for an element of an rtx. */
typedef union rtunion_def
{
  int rtint;
  char *rtstr;
  struct rtx_def *rtx;
  struct rtvec_def *rtvec;
  enum machine_mode rttype;
} rtunion;
```

Figure 2: Gcc *rtx* Definition

structures are compared to determine if they are equivalent. Figure 3 shows a slightly abbreviated version of the *_rtx_renumbered_equal_p* routine. After checking if the code and mode fields of the two *rtx* structures are identical, the routine then compares the operands, to determine if they are also identical. Four branch targets in Figure 3 are annotated with their execution weights, derived from execution profiles using the *SPEC* reference input. Roughly 1% of the time only the code fields of the two *rtx* structures are compared before exiting. In this case, only the first two bytes in each *rtx* structure is accessed. About 46% of the time x and y are *CONST_INT* *rtx*, and only the first operand is accessed. Therefore, only the first eight bytes of each *rtx* structure is accessed, and there is spatial locality within those eight bytes.

For many other types of RTL expressions, the routine will use the *for* loop to iterate through the operands, from last to first, comparing them until a mismatch is found. In this case there will be spatial locality, but at a slightly larger distance (in space) than in the previous case. Most instruction types contain more than one operand. The most common operand type in this loop is an RTL expression, which results in a recursive call to *_rtx_renumbered_equal_p*.

This routine illustrates that the amount of spatial locality can vary for particular load references, depending on the function arguments. Therefore, if the original access into each *rtx* structure in this routine is a miss, the optimal amount of data to fetch into the cache will vary correspondingly. For example, if the access `GET_CODE(y)` on line 10 of Figure 3, which performs the access `y->code`, misses in the L1 cache, the spatial locality in that data depends on whether the program will later fall into a *case* body of the *switch* statement on line 11 or into the body of the *for* loop on line 24, and on the *rtx* type of x which determines the initial value of i in the *for* loop. However, at the time of the cache miss on line 10 this information is not available, as it is highly data-dependent. As such, neither static analysis (if even possible) nor profiling will result in definitive or accurate spatial locality information for the load instructions. Dynamic analysis of the spatial locality in the data offers greater promise. For this routine, dynamic analysis of each *rtx* instance accessed in the routine would obtain the most accurate spatial locality detection. Also, dynamic schemes do not require profiling, which many users are unwilling to perform, or ISA changes.

```
int rtx_renumbered_equal_p (rtx x, rtx y)
{
  register int i;
  register RTX_CODE code = GET_CODE (x);
  register char *fmt;
  if (x == y) return 1;
  if ((code == REG || (code == SUBREG && GET_CODE (SUBREG_REG (x)) == REG))
      && (GET_CODE (y) == REG || (GET_CODE (y) == SUBREG && GET_CODE (SUBREG_REG (y)) == REG)))
    {   ... /* Rarely entered */ ...   }
   if (code != GET_CODE (y)) return 0;                    Exits here
                                                          448 times
   switch (code) {
     case PC: case CC0: case ADDR_VEC: case ADDR_DIFF_VEC:
        return 0;
     case CONST_INT:                                      Exits here
       return x->fld[0].rtint == y->fld[0].rtint;         29096 times
     case LABEL_REF:
        return (next_real_insn (x->fld[0].rtx) == next_real_insn (y->fld[0].rtx));
     case SYMBOL_REF:
        return x->fld[0].rtstr == y->fld[0].rtstr;
   }
   if (GET_MODE (x) != GET_MODE (y)) return 0;
   /* Compare the elements. If any pair of corresponding elements fail to match, return 0 for the whole thing. */
   fmt = GET_RTX_FORMAT (code);
   for (i = GET_RTX_LENGTH (code) - 1; i >= 0; i--) {
       register int j;
       switch (fmt[i]) {
         case 'i':
             if (x->fld[i].rtint != y->fld[i].rtint) return 0;
             break;
         case 's':
             if (strcmp (x->fld[i].rtstr, y->fld[i].rtstrs)) return 0;
             break;
        case 'e':                                   Case matches
           if (! rtx_renumbered_equal_p (x->fld[i].rtx, y->fld[i].rtx))     33060 times
             return 0;                                    Exits here
             break;                                       30014 times
         case 'E':
             ... /* Accesses *({x,y}->fld[i].rtvec) */ ...
             break;
       }
   }
   return 1;
 }
```

Figure 3: Gcc *_rtx_renumbered_equal_p* routine, executed 63173 times.

3.2 Applications

Aside from varying the data cache load fetch sizes, our spatial locality optimizations could be used to control instruction cache fetch sizes, write allocate versus no-allocate policies, and bypass fetch sizes when bypassing is employed. The latter case is discussed briefly in [3], and is greatly expanded in this paper. In this paper we examine the application of these techniques to control the fetch sizes into the L1 and L2 data caches. We also study these optimizations in conjunction with cache bypassing, a complementary optimization that also aims to improve cache performance.

4 Techniques

4.1 Overview of Prior Work

In this section we briefly overview the concept of a *macroblock*, as well as the *Memory Address Table* (*MAT*), introduced in an earlier paper [3] and utilized in this work.

We showed that cache bypassing decisions could be effectively made at run-time, based on the previous usage of the memory address being accessed. Other bypassing schemes include [20][21][17][22]. In particular, our scheme dynamically kept track of the accessing frequencies of memory regions called macroblocks. The macroblocks are statically-defined blocks of memory with uniform size, larger than the cache block size. The macroblock size should be large enough so that the total number of accessed macroblocks is not excessively large, but small enough so that the access patterns of the cache blocks contained within each macroblock are relatively uniform. It was determined that 1K-byte macroblocks provide a good cost-performance tradeoff.

In order to keep track of the macroblocks at run time we use an MAT, which ideally contains an entry for each macroblock, and is accessed with a macroblock address. To support dynamic bypassing decisions, each entry in the table contains a saturating counter, where the counter value represents the frequency of accesses to the corresponding macroblock. For details on the MAT bypassing scheme see [3]. Also introduced in that paper was an optimization geared towards improving the efficiency of L1 bypasses, by tracking the spatial locality of bypassed data using the MAT, and using that information to determine how much data to fetch on an L1 bypass. In this paper we introduce a more robust spatial locality detection and optimization scheme using the SLDT, which enables much more efficient detection of spatial locality. Our new scheme also supports fetching varying amounts of data into both levels of the data cache, both with and without bypassing. In practice this spatial locality optimization should be performed in combination with bypassing, in order to achieve the best possible performance, as well as to amortize the cost of the MAT hardware. The cost of the combined hardware is addressed in Section 6,

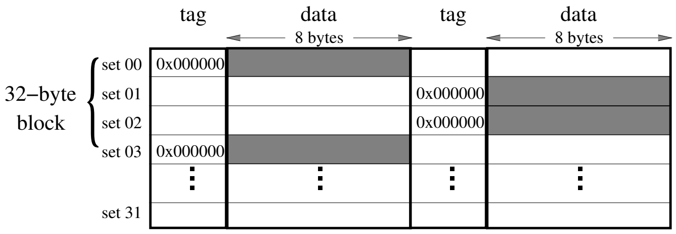


Figure 4: Layout of 8-byte subblocks from the 32-byte block starting at address 0x00000000 in a 512-byte 2-way set-associative cache with 8-byte lines. The shaded blocks correspond to the locations of the four 8-byte subblocks.

following the presentation of experimental results.

4.2 Support for Varying Fetch Sizes

The varying fetch size optimization could be supported using subblocks. In that case the block size is the largest fetch size and the subblock size is $gcd(fetch\_size_0, ..., fetch\_size_n)$, where n is the number of fetch sizes supported. Currently, we only support two power-of-two fetch sizes for each level of cache, so the subblock size is simply the smaller fetch size. However, the cache lines will be underutilized when only the smaller size is fetched.

Instead, we use a cache with small lines, equal to the smaller fetch size, and optionally fill in multiple, consecutive blocks when the larger fetch size is chosen. This approach is similar to that used in some prefetching strategies [23]. As a result, the cache can be fully utilized, even when the smaller sizes are fetched. It also eliminates conflict misses resulting from accesses to different subblocks. However, this approach makes detection of spatial reuses much more difficult, as will be described in Section 4.3. Also, smaller block sizes increase the tag array cost, which is addressed in Section 6. In our scheme, the *max_fetch_size* data is always aligned to *max_fetch_size* boundaries. As a result, our techniques will fetch data on either side of the accessed element, depending on the location of the element within the *max_fetch_size* block. In our experience, spatial locality in the data cache can be in either direction (spatially) from the referenced element.

4.3 Spatial Locality Detection Table

To facilitate spatial locality tracking, a *spatial counter*, or *sctr*, is included in each MAT entry. The role of the *sctr* is to track the medium to long-term spatial locality of the corresponding macroblock, and to make fetch size decisions, as will be explained in Section 4.4. This counter will be incremented whenever a *spatial miss* is detected, which occurs when portions of the same larger fetch size block of data reside in the cache, but not the element currently being accessed. Therefore, a hit might have occurred if the larger fetch size was fetched, rather than the smaller fetch size. In our implementation, where multiple cache blocks are filled when the larger fetch size is chosen, a spatial miss is not trivial to detect. If the cache is not fully-associative, the tags for different blocks residing in the same larger fetch size block will lie in consecutive sets, as shown in Figure 4, where the data in one 32-byte block is highlighted. Searching for other cache blocks in the same larger fetch size block of data will require access to the tags in these consecutive sets, and thus either additional cycles to access, or additional hardware support. One possibility is a restructured tag array design allowing efficient access to multiple consecutive sets of tags. Alternatively, a separate structure can be used to detect this information, which is the approach investigated in this work.

This structure is called the *Spatial Locality Detection Table* (*SLDT*), and is designed for efficient detection of spatial reuses with low hardware overhead. The role of the SLDT is to detect spatial locality of data while it is in the cache, for recording in the MAT when the data is displaced. The SLDT is basically a tag array for blocks of the larger fetch size, allowing single-cycle access to the necessary information. Figure 5 shows an overview of how the SLDT interacts with the MAT and L1 data cache, where the double-arrow line shows the correspondence of four L1 data cache entries with a single SLDT entry. In order to track all cache blocks, the SLDT would need N entries, where N is the number of blocks in the cache. This represents the worst case of having fetched only smaller (line) size blocks into the cache, all from different larger size blocks. However, in order to reduce the hardware overhead of the SLDT, we use a much smaller number of entries, which will allow us to capture only the shorter-term spatial reuses. The same SLDT could be used to track the spatial locality aspects of all structures at the same level in the memory hierarchy, such as the data cache, the instruction cache, and, when we perform bypassing, the bypass buffer.

The SLDT tags correspond to maximum fetch size blocks. The *sz* field is one bit indicating if either the larger size block was fetched into the cache, or if only smaller blocks were fetched. The *vc* (*valid count*) field is $log(max\_fetch\_size/min\_fetch\_size)$ bits in length, and indicates how many of the smaller blocks in the larger size block are currently valid in the data cache. The actual number of valid smaller blocks is $vc+1$. An SLDT entry will only be valid for a larger size block when some of its constituent blocks are currently valid in the data cache. A bit mask could be used to implement the *vc*, rather than the counter design, to reduce the operational complexity. However, for large maximum to minimum fetch size ratios, a bit mask will result in larger entries. Finally, the *sr* (*spatial reuse*) bit will be set if spatial reuse is detected, as will be discussed later.

When a larger size block of data is fetched into the cache, an SLDT entry is allocated (possibly causing the replacement of an existing entry) and the values of *sz* and *vc* are set to 1 and $max\_fetch\_size/min\_fetch\_size - 1$, respectively. If a smaller size block is fetched and no SLDT entry currently exists for the corresponding larger size block, then an entry is allocated and *sz* and *vc* are both initialized to 0. If an entry already exists, *vc* is incremented to indicate that there is now an additional valid constituent block in the data cache. For both fetch sizes the *sr* bit is initialized to 0. When a cache block is replaced from the data cache, the corresponding SLDT entry is accessed and its *vc* value is decremented if it is greater than 0. If *vc* is already 0, then this was the only valid block, so the SLDT entry is invalidated. When

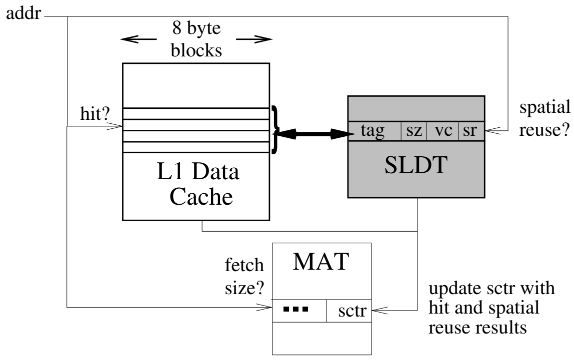


Figure 5: SLDT and MAT Hardware

an SLDT entry is invalidated its *sr* bit is checked to see if there was any spatial reuse while the data was cached. If not, the corresponding entry in the MAT is accessed and its *sctr* is decremented, effectively depositing the information in the MAT for longer-term tracking. Because the SLDT is managed as a cache, entries can be replaced, in which case the same actions are taken.

An *fi* (*fetch initiator*) bit is added to each data cache tag to help detect spatial hits. The *fi* bit is set to 1 during the cache refill for the cache block containing the referenced element (i.e. the cache block causing the fetch), otherwise it is reset to 0. Therefore, a hit to any block with a 0 *fi* bit is a spatial hit, as this data was fetched into the cache by a miss to some other element.

Table 1 summarizes the actions taken by the SLDT for memory accesses. The *sr* bit, which was initialized to zero, is set for all types of both spatial misses and spatial hits. Two types of spatial misses are detected. The first type of spatial miss occurs when other portions of the same larger fetch size block were fetched independently, indicated by a valid SLDT entry with a *sz* of 0. Therefore, there might have been a cache hit if the larger size block was fetched, so the corresponding entry in the MAT is accessed and its *sctr* is incremented. The second type can occur when the larger size block was fetched, but one of its constituent blocks was displaced from the cache, as indicated by a cache miss and a valid SLDT entry with a *sz* of 1. It is not trivial to detect if this miss is to the element which caused the original fetch, or to some other element in the larger fetch size block. The *sr* bit is conservatively set, but the *sctr* in the corresponding MAT entry is not incremented.

A spatial hit can occur in two situations. If the larger size block was fetched, then the *fi* bit will only be set for one of the loaded cache blocks. A hit to any of the loaded cache blocks without the *fi* bit set is a spatial hit, as described earlier. We do not increment the *sctr* on spatial hits, because our fetch size was correct. We only update the *sctr* when the fetch size should be changed in the future. When multiple smaller blocks were fetched, a hit to one of these is also characterized as a spatial hit. This case is detected by checking if *vc* is larger than 0 when *sz* is 0. However, we do not increment the *sctr* in this case either because a spatial miss would have been detected earlier when a second element in the larger fetch size block was first accessed (and missed).

Cache Access	SLDT Access	*fi*	*sz*	*vc*	Action
miss	hit	-	0		$sr = 1$; $sctr{+}{+}$
		-	1		$sr = 1$
hit	hit	0			$sr = 1$
			0	>0	$sr = 1$
hit	miss	0	-	-	alloc SLDT entry; $sz = 1$; $sr = 1$
		1	-	-	alloc SLDT entry
Cache entry replaced			$vc > 0$		$vc - -$
			$vc == 0$		invalidate SLDT entry
SLDT entry replaced or invalidated			$sr == 0$		$sctr - -$
			$sr == 1$		no action

Table 1: SLDT Actions. A dash indicates that there is no corresponding value, and a blank indicates that the value does not matter.

4.4 Fetch Size Decisions

On a memory access, a lookup in the MAT of the corresponding macroblock entry is performed in parallel with the data cache access. If an entry is found, the *sctr* value is compared to some threshold value. The larger size is fetched if the *sctr* is larger than the threshold, otherwise the smaller size is fetched. If no entry is found, a new entry is allocated and the *sctr* value is initialized to the threshold value, and the larger fetch size is chosen. In this paper the threshold is 50% of the maximum *sctr* value.

5 Experimental Evaluation

5.1 Experimental Environment

We simulate ten benchmarks, including *026.compress*, *072.sc* and *085.cc1* from the *SPEC92* benchmark suite using the *SPEC* reference inputs, and *099.go*, *147.vortex*, *130.li*, *134.perl*, and *124.m88ksim* from the *SPEC95* benchmark suite using the training inputs. The last two benchmarks consist of modules from the *IMPACT* compiler [24] that we felt were representative of many real-world integer applications. *Pcode*, the front end of *IMPACT*, is run performing dependence analysis with the internal representation of the *combine.c* file from GNU CC as input. *lmdes2_customizer*, a machine description optimizer, is run optimizing the SuperSPARC machine description. These optimizations operate over linked list and complex data structures, and utilize hash tables for efficient access to the information.

In order to provide a realistic evaluation of our technique for future high-performance, high-issue rate systems, we first optimized the code using the *IMPACT* compiler [24]. Classical optimizations were applied, then optimizations were performed which increase instruction level parallelism. The code was scheduled, register allocated and optimized for an eight-issue, scoreboarded, superscalar processor with register renaming. The ISA is an extension of the HP PA-RISC instruction set to support compile-time speculation.

We perform cycle-by-cycle emulation-driven simulation on a Hewlett-Packard *PA-RISC 7100* workstation, modelling the processor and the memory hierarchy (including all related busses). The instruction latencies used are those of a Hewlett-Packard *PA-RISC 7100*, as given in Table 2. The base machine configuration is described in Table 3.

Since simulating the entire applications at this level of detail would be impractical, uniform sampling is used to reduce simulation time [25], however emulation is still performed

Function	Latency	Function	Latency
Int ALU	1	FP ALU	2
memory load	2	FP multiply	2
memory store	1	FP divide (single prec.)	8
branch	1 + 1 slot	FP divide (double prec.)	15

Table 2: Instruction latencies for simulation experiments.

L1 Icache	32K-byte split-block, direct mapped, 64-byte block
L1 Dcache	16K-byte non-blocking (50 max), direct mapped, 32-byte block, multiported, writeback, no write alloc
L1-L2 Bus	8-byte bandwidth, split-transaction, 4-cycle latency, returns critical word first
L2 Dcache	same as L1 Dcache except: 256K-byte, 64-byte block
System Bus	same as L1-L2 Bus except: 100-cycle latency
Issue	8-issue uniform, except 4 memory ops/cycle max
Registers	64 integer, 64 double precision floating-point

Table 3: Base Configuration.

between samples. The simulated samples are 200,000 instructions in length and are spaced evenly every 20,000,000 instructions, yielding a 1% sampling ratio. For smaller applications, the time between samples is reduced to maintain at least 50 samples (10,000,000 instructions). To evaluate the accuracy of this technique, we simulated several configurations both with and without sampling, and found that the improvements reported in this paper are very close to those obtained by simulating the entire application.

5.2 Macroblock Spatial Locality Variations

Before presenting the performance improvements achieved by our optimizations, we first examine the accuracy of the macroblock granularity for tracking spatial locality. It is important to have accurate spatial locality information in the MAT for our scheme to be successful. This means that all data elements in a macroblock should have similar amounts of spatial locality at each phase of program execution.

After dividing main memory into macroblocks, as described in Section 4.1, the macroblocks can be further subdivided into smaller sections, each the size of a 32-byte cache block. We will simply call these smaller sections *blocks*. In order to determine the dynamic cache block spatial locality behavior, we examined the accesses to each of these blocks, gathering information twice per simulation sample, or every 100,000 instructions. At the end of each 100,000-instruction phase, we determined the fraction of times that each block in memory had at least one spatial reuse each time it was cached during that phase. We call this the *spatial reuse fraction* for that block. Figure 6 shows a graphical representation of the resulting information for three programs. Each row in the graph represents a 1K-byte macroblock accessed in a particular phase. For every phase in which a particular macroblock was accessed, there will be a corresponding row. Each row contains one data point for every 32-byte block accessed during the corresponding phase that lies in that macroblock. For the purposes of clarity, the rows were sorted by the average of the block spatial reuse fractions per macroblock. The averages increase from the bottom to the top of the graphs. The cache blocks in each macroblock were also sorted so that their spatial reuse fractions increase from left to right. Some rows are not full, meaning that not all of their blocks were accessed during the corresponding phase. Finally, the cache blocks with spatial reuse fractions falling within the same range were plotted with the same marker.

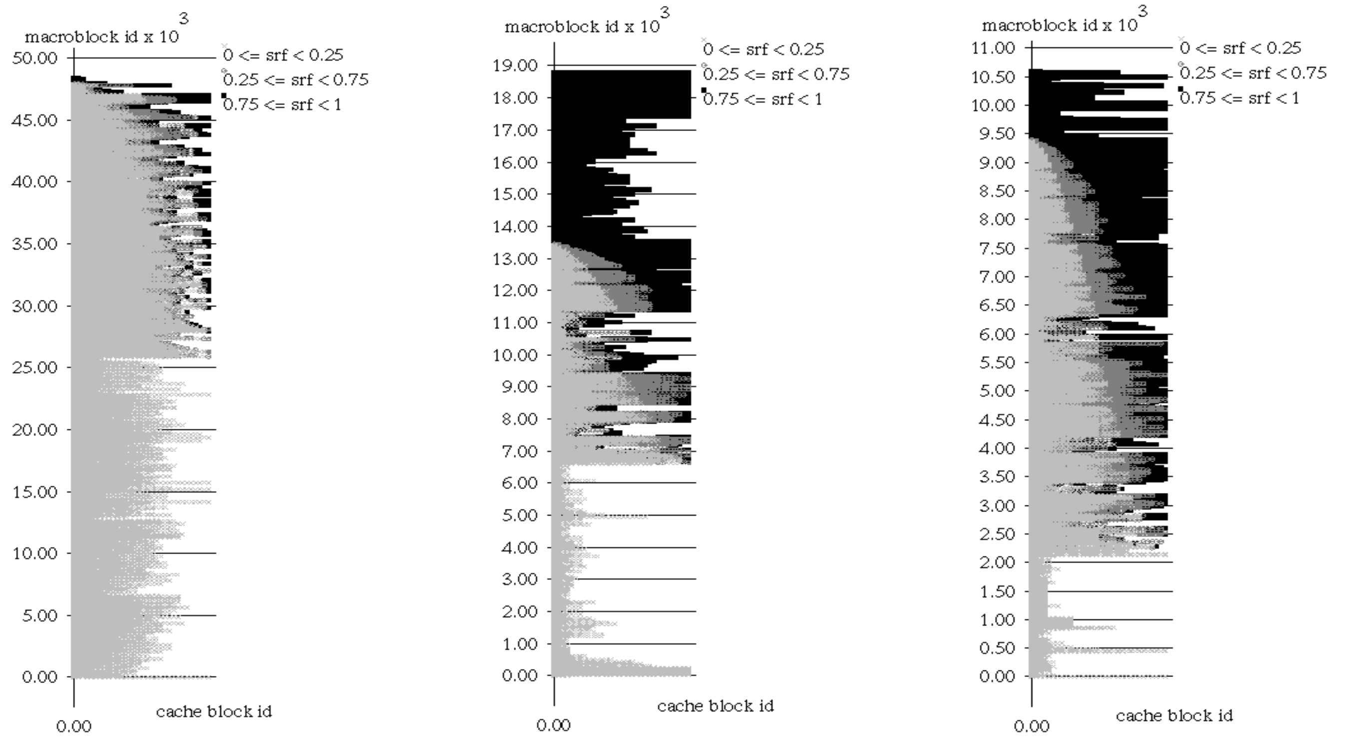
Figure 6(a) shows the spatial locality distribution for *026.compress*. Most of the blocks, corresponding to the lighter gray points, have spatial reuse fractions between 0 and 0.25, meaning that there was spatial reuse to those blocks less than 25% of the time they were cached. Very few of the blocks, corresponding to the black points, had spatial reuse more than 75% of the time they were cached. This represents a fairly optimal scenario, because most of the macroblocks contain blocks which have approximately the same amount of reuse. Figure 6(b) shows the distribution for *134.perl*. Around 34% of the macroblocks (IDs 0 to 6500) contain only blocks with little spatial reuse, their spatial reuse fractions all less than 0.25. About 29% of the macroblocks (IDs 13500 to 18900) contain only blocks with large fractions of spatial reuse, their spatial reuse fractions all over 0.75. About 37% of the macroblocks contain cache blocks with differing amounts of spatial reuse. The medium gray points in some of these rows correspond to blocks with spatial reuse fractions between 0.25 and 0.75. However, this information does not reveal the time intervals over which the spatial reuse in these blocks varies. It is possible that in certain small phases of program execution the spatial locality behavior is uniform, but that it changes drastically from one small phase of execution to another. This type of behavior is possible due to dynamically-allocated data, where a particular section of memory may be allocated as one type of data in one part of the program, then freed and reallocated as another type later. Finally, Figure 6(c) shows the distribution for *085.gcc*, which has similar characteristics to *134.perl*, but has more macroblocks with non-uniform spatial reuse fractions.

5.3 Performance Improvements

In this section we examine the performance improvement, or the execution cycles eliminated, over the base 8-issue configuration described in Section 5.1. To support varying fetch sizes, we use an SLDT and an MAT at each level of the cache hierarchy. The L1 and L2 SLDTs are direct-mapped with 32 entries. A large number of simulations showed that direct-mapped SLDTs perform as well as a fully-associative design, and that 32 entries perform almost as well as any larger power-of-two number of entries up to 1024 entries, which was the maximum size examined. The L1 and L2 MATs utilize 1K-byte macroblocks, and we examine both one and four-bit *sctr*s, We first present results for infinite-entry MATs, then study the effects of limiting the number of MAT entries.

5.3.1 Static versus Varying Fetch Sizes

The left bar for each benchmark in Figure 7(a) shows the performance improvement achieved by using 8-byte L1 data cache blocks with a static 8-byte fetch size, over the base 32-byte block and fetch sizes. These bars show that the better choice of block size is highly application-dependent. The right bars show the improvement achieved by our spatial locality optimization at the L1 level only, using an 8-byte L1 data cache block size, and fetching either 8 or 32-bytes on an L1 data cache miss, depending on the value of the



(a) 026.compress (b) 134.perl (b) 085.gcc

Figure 6: Spatial reuse fractions (*srf*) for cache-block-sized-data in the accessed macroblocks for three applications.

corresponding *sctr*. The results show that our scheme is able to obtain either almost all of the performance, or is able to outperform, the best static fetch size scheme. In most cases the 1 and 4-bit *sctr*s perform similarly, but in one case the 4-bit *sctr* achieves almost 2% greater performance improvement.

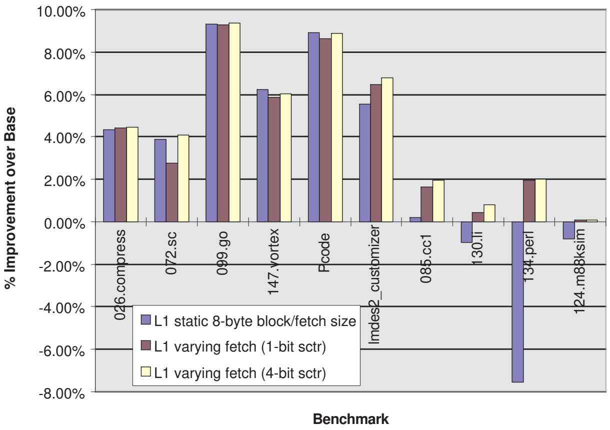
The four leftmost bars for each benchmark in Figure 7(b) show the performance improvement using different L2 data cache block and (static) fetch sizes, and our L1 spatial locality optimization with a 4-bit *sctr*. The base configuration is again the configuration described in Section 5.1, which has 64-byte L2 data cache block and fetch sizes. These bars show that, again, the better static block/fetch size is highly application-dependent. For example, *134.perl* achieves much better performance with a 256-byte fetch size, while *026.compress* achieves its best performance with a 32-byte fetch size, obtaining over 14% performance degradation with 256-byte fetches. The rightmost two bars in Figure 7(b) show the performance improvement achieved with our L2 spatial locality optimization, which uses a 32-byte L2 data cache block size and fetches either 32 or 256 bytes on an L2 data cache miss, depending on the value of the corresponding L2 MAT *sctr*. Again, our spatial locality optimizations are able to obtain almost the same or better performance than the best static fetch size scheme for all benchmarks.

Figure 8 shows the breakdown of processor stall cycles attributed to different types of data cache misses, as a percentage of the total base configuration execution cycles. The left and right bars for each benchmark are the stall cycle breakdown for the base configuration and our spatial locality optimization, respectively. The spatial locality optimizations were performed at both cache levels, using the same configuration as in Figure 7(b) with a 4-bit *sctr*. For the benchmarks that have large amounts of spatial locality, as indicated from the results of Figure 7, we obtain large reductions in L2 cold start stall cycles by fetching 256 bytes on L2 cache misses. The benchmarks with little spatial locality in the L1 data cache, such as *026.compress* and *Pcode*, obtained reductions in L1 capacity miss stall cycles from fetching fewer small cache blocks on L1 misses. In some cases the L1 cold start stall cycles increase, indicating that the L1 optimizations are less aggressive in terms of fetching more data, however these increases are generally more than compensated by reductions in other types of L1 stall cycles. The L1 conflict miss stall cycles increase for *lmdes2_customizer*, because it tends to fetch fewer blocks on an L1 miss, exposing some conflicts that were interpreted as capacity misses in the base configuration.

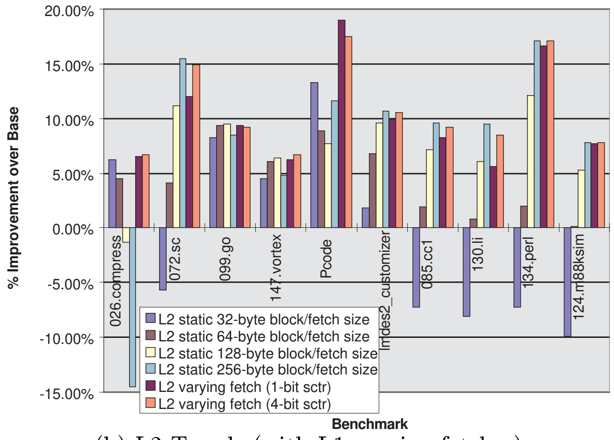
Revisiting the example of Section 3.1, we found that the access `y->code` on line 10 of Figure 3 missed 11,223 times, fetching 32 bytes for 47% of the misses, and 8 bytes for the remaining 53%. We also found that on average, 0.99 spatial hits and only 0.02 spatial misses to the resulting data occurred per miss, illustrating that our techniques are successfully choosing the appropriate amount of data to fetch on a miss.

5.3.2 Set-associative Data Caches

Increasing the set-associativity of the data caches can reduce the number of conflict misses, which may in turn reduce the advantage offered by our optimizations. However, the



(a) L1 Trends



(b) L2 Trends (with L1 varying fetches)

Figure 7: Performance for various statically-determined block/fetch sizes and for our spatial locality optimizations using both 1 and 4-bit *sctr*s.

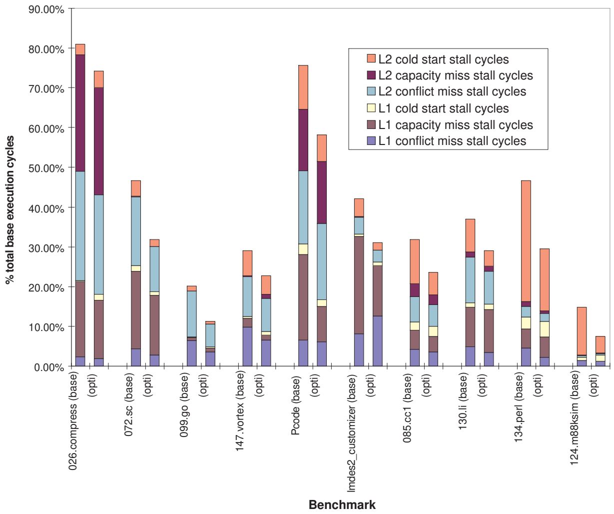


Figure 8: Stall cycle breakdown for base and the spatial locality optimizations.

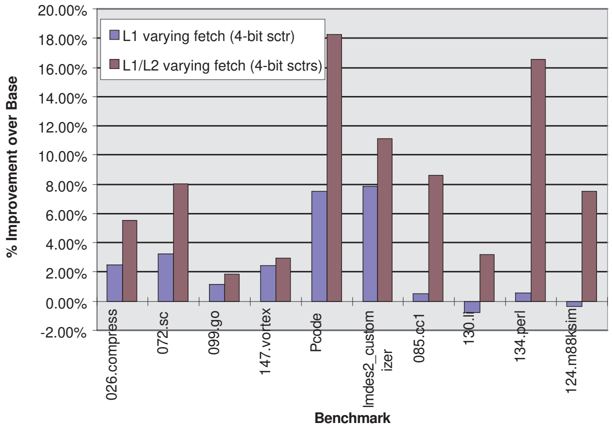


Figure 9: Performance for the spatial locality optimizations with 2-way and 4-way set-associative L1 and L2 data caches, respectively.

reductions in capacity and cold start stall cycles that our optimizations achieve should remain. To investigate these effects, the data cache configuration discussed in Section 5.1 was modified to have a 2-way set-associative L1 data cache and a 4-way set-associative L2 data cache.

Figure 9 shows the new performance improvements for our optimizations. The left bars show the result of applying our optimizations to the L1 data cache only, and the right bars show the result of applying our techniques to both the L1 and L2 data caches, using four-bit *sctr*s. The improvements have reduced significantly for some benchmarks over those shown in Figure 7. However, large improvements are still achieved for some benchmarks, particularly when applying the optimizations at the L2 data cache level, due to the reductions we achieve in L2 cold start stall cycles for data with spatial locality.

5.3.3 Growing Memory Latency Effects

As discussed in Section 1, memory latencies are increasing, and this trend is expected to continue. Figure 10 shows the improvements achieved by our optimizations when applied to direct-mapped caches for both 100 and 200-cycle latencies, each relative to a base configuration with the same memory latency. Most of the benchmarks see much larger improvements from our optimizations, with the exception of *026.compress*. Because *026.compress* has very little spatial locality to exploit, the longer latency cannot be hidden as effectively. Although the raw number of cycles we eliminate grows, as a percentage of the associated base execution cycle count it becomes smaller.

5.3.4 Comparison of Integrated Techniques to Doubled Data Caches

As the memory latencies increase, intelligent cache management techniques will become increasingly important. We examined the performance improvement achieved by integrat-

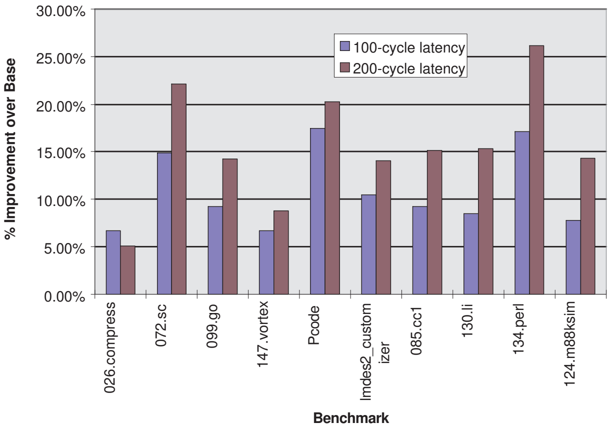


Figure 10: Performance for the spatial locality optimizations with growing memory latencies.

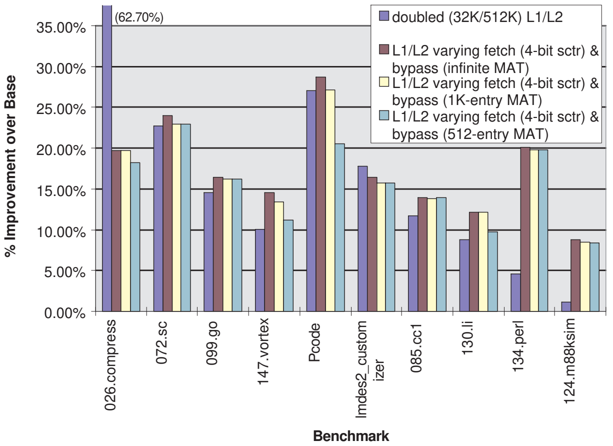


Figure 11: Comparison of doubled caches to integrated spatial locality and bypassing optimizations. Infinite, 1024-entry, and 512-entry direct-mapped MATs are examined.

ing our spatial locality optimizations with intelligent bypassing, using 8-bit access counters in each MAT entry [3]. The 4-way set-associative buffers used to hold the bypassed data at the L1 and L2 caches contain 128 8-byte entries and 512 32-byte entries, respectively. Then, the SLDT and MAT at each cache level are used to detect spatial locality and control the fetch sizes for both the data cache and the bypass buffer at that level.

Figure 11 shows the improvements achieved by combining these techniques at both cache levels for a 100-cycle memory latency. We show results for three direct-mapped MAT sizes: infinite, 1K-entry, and 512-entry. Also shown are the performance improvements achieved by doubling both the L1 and L2 data caches. Doubling the caches is a brute-force technique used to improve cache performance. Figure 11 shows that performing our integrated optimizations at both cache levels can outperform simply doubling both levels of cache. The only case where the doubled caches perform significantly better than our optimizations is for *026.compress*. This improvement mostly comes from doubling the L2 data cache, which results because its hash tables can fit into a

Cache Level	Data Cost (bytes)	Block Size (bytes)	Sets	Tag Size (bits)	Tag Cost (bytes)
L1	32K	32	1024	17	2176
L2	512K	256	2048	13	13312

Table 4: Hardware Cost of Doubled Data Caches.

512K-byte cache. *Pcode* is the only benchmark for which the performance degrades significantly when reducing the MAT size, however, 1K-entry MATs can still outperform the doubled caches. Comparing Figure 11 to the bypassing improvements in [3] shows that often significant improvements can be achieved by intelligently controlling the fetch sizes into the data caches and bypass buffers.

6 Design Considerations

In this section we examine the hardware cost of the spatial locality optimization scheme described in Section 4, and compare this to the cost of doubling the data caches at each level. As discussed in Section 4.1, the cost of the MAT hardware is amortized when performing both spatial locality and bypassing optimizations. For this reason, we compute the hardware cost of the hardware support for both of these optimizations, just as their combined performance was compared to the performance of doubling the caches in Section 5.3.4.

The additional hardware cost incurred by our spatial locality optimization scheme is small compared to doubling the cache sizes at each level, particularly for the L2 cache. For the 16K-byte direct-mapped L1 cache used to generate the results of Section 5.3, 18 bits of tag are used per entry (assuming 32-bit addresses). Doubling this cache will result in 17-bit tags. Because the line size is 32 bytes, the total additional cost of the increased tag array will be $17*2^{10} - 18*2^{9}$, which is 1K bytes[1] In addition, an extra 16K of data is needed. Similar computations will show that the cost of doubling the 256K-byte L2 cache is an extra 6144 bytes of tag and 256K bytes of data. The total tag and data costs of the doubled L1 and L2 caches is shown in Table 4.

For a direct-mapped MAT with 8-bit access counters and 4-bit spatial counters, Table 12 gives the hardware cost of the data and tags for the MAT sizes discussed in Section 5.3.4. Since all addresses within a macroblock map to the same MAT counter, a number of lower address bits are discarded when accessing the MAT. The size of the resulting *MAT address* for 1K-byte macroblocks is shown in column 3 of Table 12(a).

Table 12(b) shows the data and tag array costs for the direct-mapped data caches in our spatial locality optimization scheme. The data cost remains the same as the base configuration cost, but the tag array cost is increased due to the decreased line sizes and additional support for our scheme, which requires a 1-bit *fetch initiator bit* per tag entry.

The cost for the L1 buffer, which is a 4-way set-associative cache with 8-byte lines is shown in Table 12(c). As with the optimized data caches, the bypass buffers require a 1-bit *fetch initiator bit*, in addition to the address tag. The cost for the L2 bypass buffer is computed similarly in Table 12(c).

[1] We are ignoring the valid bit and other state.

MAT Entries	Data Cost (bytes)	Size of MAT Address (bits)	Tag Size (bits)	Tag Cost (bytes)
512	758	22	13	832
1K	1536	22	12	1536

(a) Hardware Cost of 512 and 1K entry MATs. Cost is same for both L1 and L2 cache levels.

Cache Level	Data Cost (bytes)	Fetch Size (bytes)	Block Size (bytes)	Sets	Tag Size (bits)	Tag Cost (bytes)
L1	16K	8/32	8	2048	18+1=19	4864
L2	256K	32/256	32	8192	14+1=15	15360

(b) Hardware Cost of Optimized Data Caches.

Cache Level	Entries	Fetch Size (bytes)	Block Size (bytes)	Data Cost (bytes)	Tag Size (bits)	Tag Cost (bytes)
L1	128	8/32	8	1K	24+1=25	400
L2	512	32/256	32	16K	20+1=21	1344

(c) Hardware Cost of Bypass Buffers.

Cache Level	SLDT Entries	Tag Size (bits)	Tag Cost (bytes)
L1	128	22+4=26	104
L2	128	19+5=24	96

(d) Hardware Cost of SLDTs.

Figure 12: Hardware Cost Breakdown of Spatial Locality Optimizations.

The final component of the spatial locality optimization scheme is the 32-entry SLDT, which can be organized as a direct-mapped tag array, with the *vc*, 1-bit *sz* and 1-bit *sr* fields included in each tag entry. The L1 SLDT requires a 2-bit *vc* because there are 4 8-byte lines per 32-byte maximum fetch, and the L2 SLDT requires a 3-bit *vc* due to the 8 32-byte lines per 256-byte maximum fetch. A bit mask could be used to implement the *vc*, rather than the counter design, to reduce the operational complexity. However, for large maximum to minimum fetch size ratios, such as the 8-to-1 ratio for the L2 cache, a bit mask will result in larger entries. Table 12(d) shows the total tag array costs of the L1 and L2 SLDTs.

Finally, combining the costs of the MAT, the optimized data cache, the bypass buffer, and the SLDT results in a total L1 cost of 24376 bytes with a 512-entry MAT, and 25848 bytes with a 1K-entry MAT. Therefore, the savings over doubling the L1 data cache is over 10K and 8K bytes for the 512 and 1K-entry MATs, respectively. Similar calculations show that our L2 optimizations save over 247K bytes and 245K bytes for the 512 and 1K-entry MATs, respectively, over doubling the L2 data cache. This translates into 26% and 44% less tags and data than doubling the data caches at the L1 and L2 levels, respectively, for the larger 1K-entry MAT. Comparing the performance of the spatial locality and bypassing optimizations to the performance obtained by doubling the data caches at both levels, as shown in Figure 11, illustrates that for much smaller hardware costs our optimizations usually outperform simply doubling the caches.

To reduce the hardware cost, we could potentially integrate the L1 MAT with the TLB and page tables. For a macroblock size larger than or equal to the page size, each TLB entry will need to hold only one 8-bit counter value. For a macroblock size less than the page size, each TLB entry needs to hold several counters, one for each of the macroblocks within the corresponding page. In this case a small amount of additional hardware is necessary to select between the counter values. However, further study is needed to determine the full effects of TLB integration.

7 Conclusion

In this paper, we examined the spatial locality characteristics of integer applications. We showed that the spatial locality varied not only between programs, but also varied vastly between data accessed by the same application. As a result of varying spatial locality within and across applications, spatial locality optimizations must be able to detect and adapt to the varying amount of spatial locality both within and across applications in order to be effective. We presented a scheme which meets these objectives by detecting the amount of spatial locality in different portions of memory, and making dynamic decisions on the appropriate number of blocks to fetch on a memory access. A Spatial Locality Detection Table (SLDT), introduced in this paper, facilitates spatial locality detection for data while it is cached. This information is later recorded in a Memory Address Table (MAT) for long-term tracking, and is then used to tune the fetch sizes for each missing access.

Detailed simulations of several applications showed that significant speedups can be achieved by our techniques. The improvements are due to the reduction of conflict and capacity misses by utilizing small blocks and small fetch sizes when spatial locality is absent, and utilizing the prefetching effect of large fetch sizes when spatial locality exists. In addition, we showed that the speedups achieved by this scheme increase as the memory latency increases.

As memory latencies increase, the importance of cache performance improvements at each level of the memory hierarchy will continue to grow. Also, as the available chip area grows, it makes sense to spend more resources to allow intelligent control over the cache management, in order to adapt the caching decisions to the dynamic accessing behavior. We believe that our schemes can be extended into a more general framework for intelligent runtime management of the cache hierarchy.

Acknowledgements

The authors would like to thank Mark Hill, Santosh Abraham and Wen-Hann Wang, as well as all the members of the IMPACT research group, for their comments and suggestions which helped improve the quality of this research. This research has been supported by the National Science Foundation (NSF) under grant CCR-9629948, Intel Corporation, Advanced Micro Devices Hewlett-Packard, SUN Microsystems, NCR, and the National Aeronautics and Space Administration (NASA) under Contract NASA NAG 1-613 in cooperation with the Illinois Computer laboratory for Aerospace Systems and Software (ICLASS).

References

[1] T. L. Johnson, M. C. Merten, and W. W. Hwu, "Run-time spatial locality detection and optimization," in *To appear in the Proceedings of the 30th International Symposium on Microarchitecture*, December 1997.

[2] K. Boland and A. Dollas, "Predicting and precluding problems with memory latency," *IEEE Micro*, pp. 59–66, August 1994.

[3] T. L. Johnson and W. W. Hwu, "Run-time adaptive cache hierarchy management via reference analysis," in *Proceedings of the 24th International Symposium on Computer Architecture*, pp. 315–326, June 1997.

[4] S. Przybylski, "The performance impact of block sizes and fetch strategies," in *Proceedings of the 18th International Symposium on Computer Architecture*, pp. 160–169, June 1990.

[5] A. J. Smith, "Line (block) size choice for cpu cache memories," *IEEE Transaction on Computers*, vol. C-36, pp. 1063–1075, 1987.

[6] F. Dahlgren, M. Dubois, and P. Strenstrom, "Fixed and adaptive sequential prefetching in shared memory multiprocessors," in *Proceedings of the 1993 International Conference on Parallel Processing*, pp. 56–63, August 1993.

[7] A. J. Smith, "Cache memories," *Computing Surveys*, vol. 14, no. 3, pp. 473–530, 1982.

[8] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *Proceedings of the 17th International Symposium on Computer Architecture*, pp. 364–373, June 1990.

[9] J.-L. Baer and T.-F. Chen, "An effective on-chip preloading scheme to reduce data access penalty," in *Proceeding of Supercomputing '91*, pp. 176–186, Nov. 1991.

[10] S. Mehrotra and L. Harrison, "Quantifying the performance potential of a data prefetch mechanism for pointer-intensive and numeric programs," Tech. Rep. 1458, CSRD, Univ. of Illinois, November 1995.

[11] J. W. C. Fu, J. H. Patel, and B. L. Janssens, "Stride directed prefetching in scalar processors," in *Proc. 25th Ann. Conference on Microprogramming and Microarchitectures*, Dec. 1992.

[12] A. K. Porterfield, *Software Methods for Improvement of Cache Performance on Supercomputer Applications.* PhD thesis, Department of Computer Science, Rice University, Houston, TX, 1989.

[13] T. C. Mowry, M. S. Lam, and A. Gupta, "Design and evaluation of a compiler algorithm for prefetching," in *Proc. Fifth Int'l Conf. on Architectural Support for Prog. Lang. and Operating Systems.*, pp. 62–73, Oct. 1992.

[14] W. Y. Chen, S. A. Mahlke, P. P. Chang, and W. W. Hwu, "Data access microarchitectures for superscalar processors with compiler-assisted data prefetching," in *Proceedings of the 24th Annual International Symposium on Microarchitecture*, pp. 69–73, November 1991.

[15] C.-K. Luk and T. C. Mowry, "Compiler-based prefetching for recursive data structures," in *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 222–233, September 1996.

[16] M. H. Lipasti, W. J. Schmidth, S. R. Kunkel, and R. R. Roediger, "SPAID: Software prefetching in pointer- and call-intensive environments," in *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pp. 231–236, December 1995.

[17] A. González, C. Aliagas, and M. Valero, "A data cache with multiple caching strategies tuned to different types of locality," in *Proc. International Conference on Supercomputing*, pp. 338–347, July 1995.

[18] V. Milutinovic, B. Markovic, M. Tomasevic, and M. Tremblay, "The split temporal/spatial cache: Initial performance analysis," in *Proceedings of the SCIzzL-5*, March 1996.

[19] K. S. McKinley and O. Temam, "A quantitative analysis of loop nest locality," in *Proceedings of the 7th International Conference on Architecture Support for Programming Languages and Operating Systems*, pp. 94–104, October 1996.

[20] R. A. Sugumar and S. G. Abraham, "Efficient simulation of caches under optimal replacement with applications to miss characterization," Tech. Rep. CSE-TR-143-92, Univ. of Michigan, 1992.

[21] G. Tyson, M. Farrens, J. Matthews, and A. R. Pleszkun, "A modified approach to data cache management," in *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pp. 93–103, December 1995.

[22] J. A. Rivers and E. S. Davidson, "Reducing conflicts in direct-mapped caches with a temporality-based design," in *Proceedings of the 1996 International Conference on Parallel Processing*, pp. 151–162, August 1996.

[23] J. W. C. Fu and J. H. Patel, "Data prefetching in multiprocessor vector cache memories," in *Proc. 18th Ann. Int'l Symp. Computer Architecture*, pp. 54–63, June 1991.

[24] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu, "IMPACT: An architectural framework for multiple-instruction-issue processors," in *Proceedings of the 18th International Symposium on Computer Architecture*, pp. 266–275, May 1991.

[25] J. W. C. Fu and J. H. Patel, "How to simulate 100 billion references cheaply," Tech. Rep. CRHC-91-30, Center for Reliable and High-Performance Computing, University of Illinois, Urbana, IL, 1991.