Using Cache as a Local Memory

Arvind Mandhani, Todd A. Cook — Department of Electrical and Computer Engineering, Rutgers University

Ulrich Kremer* — Department of Computer Science, Rutgers University

Abstract

Inability to reuse data, conflicting references, and underutilization of cache capacity are responsible for poor cache performance on various commonly used applications. Data prefetching, blocking, and data copying have been used to address these problems. These techniques, though effective, are directed towards solving one aspect of the overall problem. We propose a comprehensive solution to these problems by using the on-chip cache as an explicitly managed local memory. With minimal changes to existing cache hardware, the cache can be dynamically configured to act as a local memory or as a conventional cache; reconfiguration overhead is small. In conjunction with the compiler, with no changes to existing instruction set, and with the help of a few new instructions, we achieve an overall execution speedup ranging from 1.4 to 2.6. Cache interference is eliminated without copying data. Software is able to use conventional cache performance improvement techniques without the performance unpredictability associated with them.

1 Introduction

Cache memories are widely used to alleviate the memory latency bottleneck in modern microprocessors. With the increasing gap between processor and memory speed, high cache performance has become critical. However, poor reuse of data, conflicts between various references, and underutilization of cache capacity lead to poor cache performance for various commonly used applications [4, 23].

In this paper, we propose a new approach to improve the cache performance of programs with memory access patterns not well-suited for conventional cache architectures. We propose to use the cache as a reconfigurable local memory, where the allocation/deallocation of data in the local memory is performed by the compiler; we show that a conventional cache can be dynamically reconfigured as a local memory, with minimum overhead, to eliminate the cache misses caused by self-interference, cross-interference, and line misalignment.

We demonstrate the effectiveness of our scheme on a class of applications where array accesses are the dominant form of data access. While there are many techniques for improving the performance of these applications, they do not consistently realize the full potential of caches. The large data sets of typical scientific applications yield poor reuse of data due to the limited flexibility of most caches [10]. Array-based applications can also exhibit poor address sequentiality, which depends on the stride of the array references; in numerical codes, the regular strides that occur within loop nests cause cache interference that degrades performance considerably [3, 4, 19]. Constant but non-unit stride accesses do not in general improve cache performance; even after blocking, there is a relatively high variance in cache misses for different matrix sizes [13]. Even if the working set fits in the cache for blocked loop nests, cache utilization still remains poor due to non-sequentiality of data accesses; for example, a cache with a line size of four words would utilize only one fourth of its data if the stride of access were four. Data copying is one approach which has been proposed to reduce conflict misses [13, 20]; however, copying has its own overhead, and it can cause subtle problems with self interference [15].

To address these problems, we propose to use the on-chip cache as a local memory. With minimal changes to existing cache hardware, the cache can be configured to act as a local memory or as a conventional cache, with almost insignificant reconfiguration overhead. Block transfer instructions move data between off-chip memory and the local memory, and local memory access instructions refer to individual data elements in the local memory. Block transfer of data is done in hardware by a block transfer unit. To maintain compatibility, the processor can operate in either normal or local memory modes. In the normal mode, the cache is accessed using conventional load and store instruc-

*Corresponding Author; Dept. of Comp. Science, Rutgers University, New Brunswick, NJ 08903; email: uli@cs.rutgers.edu.

tions. If a compiler detects array references which can take advantage of the local memory scheme, it inserts instructions to switch the processor to the local memory mode. As is shown later, the switching overhead between the two modes is minimal.

Our approach has a number of strengths, which include:

- It can be implemented as a small extension to any instruction set architecture, without modifying existing instructions.
- Only five new instructions are needed to implement the scheme.
- No changes are required to the fundamental structure of the cache; changes only affect how addresses are computed before being applied to the cache.
- Hardware changes will not affect any critical path.
- Performance benefits are possible for non-numerical codes; for example, heap allocation with copying garbage collection [9]. [1]

We demonstrate the effectiveness of our scheme by showing speedups of 1.4 to 2.6 on a set of representative, array-intensive loop nests.

In the next section, we discuss previous work. In Section 3, we describe the local memory scheme in detail; we discuss the new instructions required, the changes in the memory system of the processor, and the design of the cache itself. In Section 4, we demonstrate, with the help of an example, how our scheme, in conjunction with the compiler, can be used to improve cache performance. Section 5 describes our simulation environment and results. Section 6 discusses future work. We conclude in Section 7.

2 Previous Work

Several hardware and software approaches have been proposed to improve cache performance of array-based applications. One such approach is to prefetch data into cache before it is needed [2, 8, 10, 11, 17]. Data prefetching is generally effective in hiding the long cache miss penalty by fetching data in advance of use and in parallel with computations. Several intelligent approaches have been proposed for data prefetching, where the decision to prefetch or not is made either in software [17] or in hardware [2]. Prefetching requires that the cache hold the working set and the future working set simultaneously, possibly increasing cache pollution and bus traffic[7]. Bus traffic can also increase in hardware prefetching due to unnecessary prefetches. Fu et al. [11] state that prefetching does reduce the cache miss ratio, but a high percentage of loads do not contribute to this reduction. Also, prefetching does not target interference misses, since it does not alter the way data is mapped into the cache.

Several schemes have been proposed where the prefetched data is brought into a separate buffer instead of cache. Chen et al. [7] use a *preload buffer* to store the prefetched data, and then that buffer is used instead of the cache. A similar buffer, called a *programmable cache*, was proposed by Sohi et al. [18] for use in the scalar unit of a vector processor. If caches are retained for general-purpose applications, implementation of these techniques results in duplication of on-chip memory space.

Data copying has been proposed to reduce conflict misses [13]. Temam et al. [20] present a compile time technique for determining when copying is advantageous. However, the overhead incurred in copying can be low only if the data reuse factor is high. Also, copying can cause subtle problems with self interference if the data to be copied and the destination are both cached [15].

Each of the techniques proposed above targets a specific issue. For example, blocking tries to reduce capacity misses; prefetching attempts to ameliorate long cache miss penalties; and copying targets conflict misses. Several other schemes even discard the cache altogether [7, 15, 18].

In contrast, we propose a comprehensive solution to the above problems, without using a separate buffer or copying data from one place in the cache to another, while still retaining all the benefits of previous solutions. We use conventional cache performance techniques without the performance unpredictability associated with them. For example, copying methods copy non-contiguous data into a contiguous region of memory; we achieve the same effect by a hardware gather/scatter operation with a lower overhead. We also use prefetching, but with our scheme, prefetching does not result in cache pollution or excessive increases in memory traffic. Similarly, we also use blocking, but cache performance is consistent irrespective of the access stride. Experimental results validating our claims are discussed in Section 5.

McKee et al. [15] discuss a dynamic access ordering system, where memory is connected to the processor through a controller that issues memory requests and determines the order of requests during streaming computations. The data fetched from memory is held in stream buffers, which are implemented as a set of FIFOs. McKee's argument against using caches is that cache-level access ordering suffers from lack of compile-

[1] This application will be explored in future work.

time information regarding data placement and alignment. Our local memory approach provides an efficient solution to these problems.

3 Cache as a Local Memory

We propose to use the cache as an explicitly managed local memory. To use the local memory, a block of data is first fetched into the local memory with help of a *block load* instruction. The individual elements of the data block are then accessed using *local load* and *local store* instructions. After computations are performed, the data is stored back to the lower level of the memory hierarchy using a *block store* instruction.

Block fetching of data is performed by a hardware block transfer unit; but determination of where data is stored in the local memory is explicitly done by the compiler. The compiler therefore knows the address of each block element in the local memory.

The data block fetched into the local memory is always stored sequentially, irrespective of the original stride between data elements; this implies that non-unit stride array references are mapped into a unit-stride address space. The local memory accesses can then exploit spatial locality irrespective of the stride of the array access. Using a cache as a local memory improves cache utilization, since unlike non-unit stride references where only one word per cache block may be useful, our scheme only brings useful data into the cache.

Figure 1 shows a simple example illustrating the potential performance benefits of a local memory. Four elements of array A are referenced with a stride of 4; Figure 1(a) shows the individual elements of the array which are accessed. Assuming each cache line can hold four elements of the array, Figure 1(b) shows the layout of the array data in a conventional cache. Figure 1(c) shows the data layout after the four elements of the array are fetched into the local memory using a block load instruction. For this example, the working set size in the local memory can be four times larger than the working set in a conventional cache.

To avoid confusion, we will refer to accesses into local memory as *local memory accesses* and into conventional cache as *cache accesses*. Similarly, data in the local memory is called *local memory data*, and data in the cache is called *cache data*.

3.1 Local Memory Data Addressing

This subsection shows how a conventional cache can be addressed as a local memory. In local memory mode, the individual data elements are accessed using the same mechanism as in a normal cache access. The difference between the two modes is in how addresses are constructed before being passed to the cache.

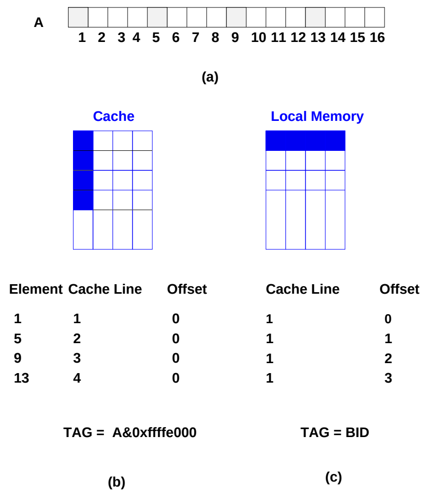

Figure 1: Cache as a Local Memory

To make local memory data allocation explicit, the compiler provides the tag, line number, and offset for the data, instead of using the original address. The need for a tag arises because access to elements of a data block can begin even before the whole block is fetched; to make sure that correct data is referenced, each data block is assigned a unique block ID, called BID. BID is equivalent to the tag of a conventional cache location, and a local load or store instruction that accesses a data element uses the BID to identify the data block. The line number where the data is stored is determined by the compiler assigned variable, LINE_NO. The N-bit address of the first data element of a data block in the local memory would be:

n … k+1	k … j+1	j … 0
BID	LINE_NO	0

The first element is always aligned in the cache line; hence, the offset is zero. The way a compiler would compute the address shown above would be:

```
ADDR = (BID << k) || (LINE_NO << j)
```

where `j` is the number of bits required for the offset into the cache-line, and `k` is the number of bits required to index the cache set plus `j`. For example, for a 32KB four-way set-associative cache with 32 byte line size, `j` will be 5 and `k` will be 13. The line number, LINE_NO, for the example in Figure 1 would be 1.

3.2 Memory System Organization

The on-chip memory system, with the addition of the block transfer unit, is shown in Figure 2. The block

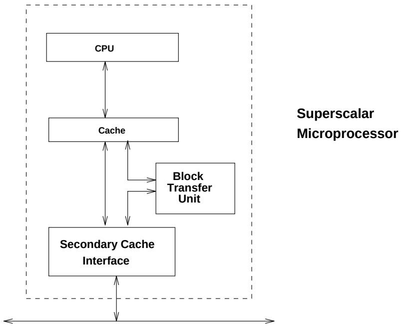

Figure 2: The Memory System

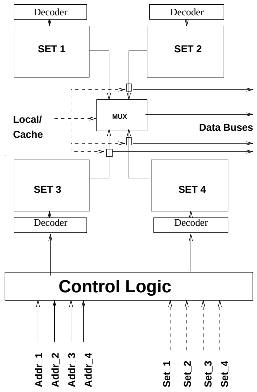

Figure 3: The Cache/Local Memory

transfer unit shares the cache with the processor, and it also shares the secondary cache interface with the on-chip cache. In the normal mode, the block transfer unit is disabled.

In a pipelined processor, two loads and one write operation every cycle would be required to support long vector operations. To provide that kind of memory bandwidth, the cache would have to have at least three ports. In the local memory scheme, it is important that the block transfer unit does not compete with the processor for cache access; thus, we need a dedicated cache port for the block transfer unit. This implies that to support high bandwidth to local memory, the on-chip cache has to have four ports, which is not a practical solution.

To get around the need for a large number of cache ports, we use a dual-ported, four-way set-associative cache. (Alternative organizations are possible.) In the normal mode, the processor has access to both ports. In the local memory mode, however, one of the cache ports is used by the processor, and the other port is used by the block transfer unit.

Use of a set-associative cache provides the possibility of independent access to each set. In the local mode, the compiler determines in which set data will be allocated. For a local memory access, since the set number is provided with the address, only that set has to be indexed to access the data. Therefore, in any cycle, up to four simultaneous data references can be made to the local memory, one to each set. We use three sets as local memory; the fourth set acts as a direct mapped, conventional cache with one quarter the size of the original cache. Therefore, if the cache-controller has one cache request and three local memory requests in a single cycle, and all of the requests are to different sets, they can all access the cache/local-memory simultaneously. This mixed approach makes it possible to cache scalar data with local memory data.

Figure 3 shows the cache configuration we use to illustrate our scheme. This configuration is not a requirement; other configurations are possible. Set numbers 2 through 4 are used as local memory sets, and set number 1 is used as a direct mapped cache. In local memory mode, each set is accessed independently; thus, each set has to have its own address decoder and an independent bus to the processor. The address decoders are duplicates of the original address decoder. The multiplexer which selects the successful set in the normal mode is disabled in local memory mode.

The cache is write-back, no-write-allocate, and supports sub-block placement; again, other organizations are possible. Each cache line has a valid status bit. Since sub-block placement is supported, each word in a line has a dirty bit. To support local memory operations, another status bit, *local*, is also provided for each word. The next subsection explains how *local* is used.

3.3 Switching Modes

Switching modes does not have any significant overhead, because no flushing of the cache is required; this implies that data brought into the cache or local memory will still be there when modes are switched. The *local* status bit is used to distinguish between normal cache data and local memory data.

When switching from normal mode to local memory mode, dirty cache data will be written back to memory if it must be displaced by block load instructions . Since the dirty data would have been eventually displaced anyway, this flushing is not an additional overhead. Also, local loads and stores will not reference data remaining from normal mode, since that data will have its *local* status bit reset.

When switching from local memory mode to normal mode, only unmodified data will remain in the local memory, since the compiler is responsible for writing local memory contents back to main memory. This writeback would eventually have to be done even if no mode switch were made. Therefore, since there is no modified local data, there will be no writeback of local data in normal mode. Also, the unmodified local data will have its *local* status bit set; thus, it will not be accessed in the normal mode.

3.4 Instruction Set Support

Implementation of our scheme requires augmenting the instruction set of the processor, but no changes in the existing instructions are required. The following are brief descriptions of the new instructions:

block load
: This instruction fetches data from lower levels of the memory hierarchy into the local memory. It has five operands: the address of the first element of the array, the new address where the array will be stored in the local memory, the stride of the access, the length of the array, and the destination local memory set number. This instruction is executed by the block transfer unit, which does not interfere with the processor-cache access path. In our implementation, the first four parameters are written to a group of four Local Memory Parameter Registers (LMPR); we provide three such groups, one each for the three local memory sets. The instruction itself has one parameter, the local memory set number. Three groups of LMPRs provide more flexibility in issuing block load instructions and saves some of the extra instruction overhead. For example, if two block load instructions with different length, stride, and set parameters are executed in a loop, each instruction would be assigned a group of LMPRs; the length and stride parameters would be written only once, instead of in each iteration.

block store
: This instruction stores array data from local memory back to lower levels of the memory hierarchy. It has the same arguments as the block load instruction.

switch
: This instruction toggles between processor modes. It acts as a barrier so that all previous instructions complete before execution proceeds.

local load
: In the local memory mode, the processor uses this instruction to load data from the local memory into a processor register. The syntax and format of the local load instruction are similar to a generic load/store instruction using a "register + offset" addressing mode. Since this instruction has to specify the local memory set number also, we include a two-bit set specifier. For example:

```
llm     r1,1,0(r2)
```

loads a double word into `r1` from the local memory set number 1 at an address given by register `r2`. If the offset were 8, the local load instruction now would be:

```
llm     r1, 1,8(r2)
```

local store
: This instruction stores the contents of a register into the local memory. The operation of this instruction is similar to that of the local load instruction.

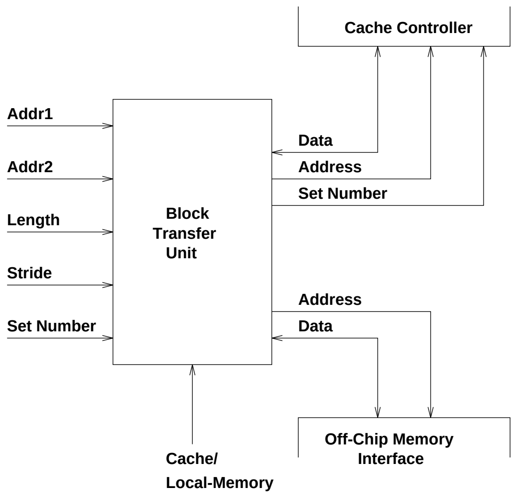

Figure 4: Cache as a Local Memory

3.5 Block Transfer Unit

The block transfer unit executes block load and store instructions; it is essentially an address generator. Figure 4 shows how it interfaces with the cache/local-memory and the off-chip memory interface. The inputs to the unit are the parameters of the block load and store instructions. `Addr1` is the address of data in main memory, and `Addr2` is the address of data in local memory.

In the case of a block load instruction, a read request to off-chip memory is made at address `Addr1`. After the data arrives, a write request to the local memory is made at address `Addr2`. Addresses `Addr1` and `Addr2` are then incremented by the size of the data element. The process is repeated until all data elements are fetched.

In a pipelined memory system, it is possible to have more than one data element request outstanding. In the case of unit stride accesses, the size of each request is equal to the bus width instead of the data element size. The block store instruction operates similarly.

Data is only fetched into local memory by the block transfer unit. If a local load instruction does not find the local memory data, it does not generate a request for the off-chip memory. Block load and store instructions are always executed in program-sequential order. This requirement simplifies the design of the block transfer unit. Our experience has shown that the complexity of an aggressive design far outweighs the performance gains.

Yamada [22] proposes a design for a *Data Relocation and Prefetching* (DRP) unit, which executes instructions similar to our block loads and stores. Our block fetch unit design is a simplified version of Yamada's.

4 Using the Local Memory

To modify an existing program to use the local memory require two changes: generating block load instructions for the array accesses in the inner-loop, and generating local loads and stores to access individual elements of the arrays. Techniques for generating block loads and stores are similar to vector code generation [24]. Allocating line numbers to different data blocks is essentially similar to vector register allocation; Allen et al. [1] discuss in detail the optimization techniques for vector register allocation. Standard vectorization techniques, such as sectioning vector statements, loop interchange, and loop fusion can be applied to generate code for our local memory scheme.

For the scheme to be advantageous, it is important that block loads and stores be executed in parallel with computations and that data be prefetched into local memory before it is needed. To achieve that, each local memory set is divided into two regions: the computation region, where the data is accessed for computations, and the prefetch region, where data is prefetched for the next set of computations.

Figure 4(a) shows a loop nest where `A` and `B` are double-precision square matrices of size `NxN`. The stride of access in the inner-loop is equal to N elements, or N*8 bytes. The inner-loop involves three memory accesses: two loads and one store; A(J, K-1) does not need to be fetched, since it is fetched in the previous iteration. N elements of `A` and N-1 elements of `B` are accessed in one iteration of the outer loop.

Figure 4(b) shows the loop nests after they have been transformed to use the local memory. For simplicity in presenting the example, we are assuming that N elements of the matrices require less than half of the memory space provided by a local memory set (larger data sets can be handled using appropriate sectioning). We are also assuming a 32K, four-way set-associative cache with a line size of 32 bytes. Each cache set has 256 lines, capable of storing 1024 double-precision elements.

```
(a) original loop nest.

     DO 20 J = 1, N
        DO 20 K = 2, N
           A(J, K) = A(J, K) + A(J, K-1)*B(J, K)
 20     CONTINUE


(b) local memory transformed loop nest.

     NEXT_ADDR = COMP_ADDR(0, 0)
     BLOCK_LOAD(A(1, *), NEXT_ADDR, N, N*8, 1)
     BLOCK_LOAD(B(1, *), NEXT_ADDR, N-1, N*8, 2)
     NEXT_LINE = 128
     CURRENT_LINE = 0

     DO 30 J = 1, N
        NEXT_ADDR = COMP_ADDR(NEXT_LINE, J)
        IF(J.LT.N) THEN
           BLOCK_LOAD(A(J+1, *), NEXT_ADDR, N, N*8, 1)
           BLOCK_LOAD(B(J+1, *), NEXT_ADDR, N-1, N*8, 2)
        ENDIF
        CURR_ADDR = COMP_ADDR(CURR_LINE, J-1)

        DO 20 K = 2, N
           A(J, K) = A(J, K) + A(J, K-1)*B(J, K)
 20     CONTINUE

        BLOCK_STORE(A(J,*), CURR_ADDR, N, N*8, 1)

        CURR_LINE = NEXT_LINE
        IF(CURR_LINE.EQ.0) THEN
           NEXT_LINE = 128
        ELSE
           NEXT_LINE = 0
 30  CONTINUE
```

Figure 5: Local Memory transformed loop nest

The first two block load instructions bring in the array elements accessed in the first iteration of the outer-loop. Inside the loop body, another set of block load instructions fetches the data for the next iteration; this data is thus fetched in parallel with the current iteration. Variable `CURR_LINE` denotes the line number were data access will begin for the current iteration of the `J` loop; similarly, variable `NEXT_LINE` denotes the first line number where the prefetched data block will be stored. `CURR_ADDR` and `NEXT_ADDR` are the corresponding local memory addresses for `CURRENT_LINE` and the `NEXT_LINE` respectively. The `COMP_ADDR` function calculates the local memory address, given the cache line number and the BID for the array data. The computation and the prefetch region alternate between the lower and upper half of a local memory set.

To determine values for BIDs, we use the value of the outer loop index, `J`, when the block load instruction is executed. When `CURR_ADDR` is calculated, J-1 is used as the BID, since the data for current iteration is fetched in the previous of iterations of the `J` loop. Using loop indices as BIDs simplifies the task of assigning BIDs because this coordinates the use of BIDs between iterations of a loop.

Figure 4(b) does not show how the individual array elements are accessed in the local memory. The following assembly language code for one iteration of the `K` loop shows how it can be done (the actual code would be unrolled and software pipelined):

```
move $2, curr_addr

llm     $f4, 1,0($2)     # $2 = &A(J, 1)
llm     $f6, 2,0($2)     # $3 = &B(K, 2)
mul.d   $f8, $f4, $f6
llm     $f10, 1,8($2)    # 33($2) = &A(J, 2)
add.d   $f16, $f10, $f8
slm     $f16, 1,8($2)
addi    $2, $2, 8
```

The same register is used to address both `A` and `B`. Since `A(J,K-1)` and `B(J,K)` have corresponding positions in the local memory sets and the same BID, their local memory addresses are the same. In a conventional microprocessor, the compiler would use two different registers to address `A` and `B`; hence, there will be two induction variable updates per iteration of the inner loop. Our scheme reduces the loop overhead by eliminating separate address registers for separate arrays.

The transformation shown in Figure 4(b) is valid when `N` is less than 512. When the value of `N` is not known at compile time, appropriate sectioning techniques can be used.

5 Experimental Evaluation

For a realistic evaluation of our approach, a set of benchmark loop nests was aggressively optimized using a source level compiler capable of performing loop-level transformations and tiling. The resulting optimized code forms our base code and all performance results are reported relative to the performance of this base code. The base code is then transformed to use the local memory. The simulation results presented here are obtained by simulating an aggressive superscalar processor, and they represent a realistic lower bound on the performance gain. We use the MIPS-II instruction set as the basis of our experiments, but any other existing instruction set architecture could have been used.

5.1 Simulation Environment

Since the local memory-scheme involves incorporation of new instructions, we modified SPIM [14] to implement the new instructions and to generate traces. We also added a cache to SPIM to help execute the block loads and stores.

The traces generated by SPIM are piped to the trace-driven, superscalar simulator Ssim [12]. The simulator supports out-of-order instruction issue with the help of a reservation station at the input of each functional unit. The simulator has 10 functional units, and we added a functional unit to simulate the operation of the block transfer unit. Ssim implements register renaming with the help of a reorder buffer at each functional unit.

Func Unit	Latency Result	Latency Issue	Func Unit	Latency Result	Latency Issue
Integer	1	1	Float Add	2	2
Shifter	1	1	Float Mul	3	2
Branch	1	1	Float Div	19	19
Load	1	2	Float Conv	2	1
Store	1	1	Local	1	1

Table 1: Functional Unit latencies

Table 1 lists the latencies we use for the functional units. Where appropriate, latencies shown are for double word operands. Four instructions are decoded in every cycle, and if there is no dependence and the functional units are not busy, as many instructions can be issued as there are functional units. The latencies we have chosen are close to the latencies of the MIPS R10000 [16].

We use a dual-ported, 32KB, four-way set-associative cache with 32-byte lines. The secondary cache has a latency of 8 cycles with an issue latency of 2 cycles. The secondary cache bus is pipelined and is 128 bits wide. Fetching of one cache line has a latency of 9 cycles. For block load instructions, if the access is non-unit stride, it takes 8 cycles to fetch one double word; if the access is unit stride, the normal cache line refill latency applies.

5.2 Benchmarks

We selected a number of kernels from standard benchmark suites to evaluate our scheme. The loop nests are named as "*benchmark-subroutine-outer_loop_number*". `ARC2D` is a general-purpose, implicit finite-difference code for analyzing two-dimensional fluid flow problems by solving the Euler equation. `LUD` solves the linear system $Ax = b$ given the LU factorization of A; we ran the benchmark using a 1000x1000 matrix. `NAS` is a kernel benchmark program; subroutine `VPENTA` inverts 3 pentadiagonals simultaneously. `VECOPS` and `VECSKIP` are part of the LANL benchmark suite; they consist

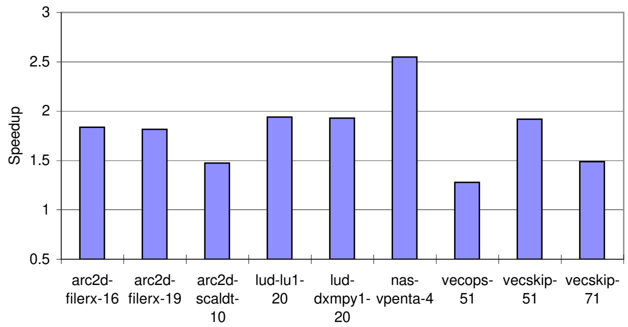

Figure 6: Speedups obtained using a local memory versus a Conventional cache

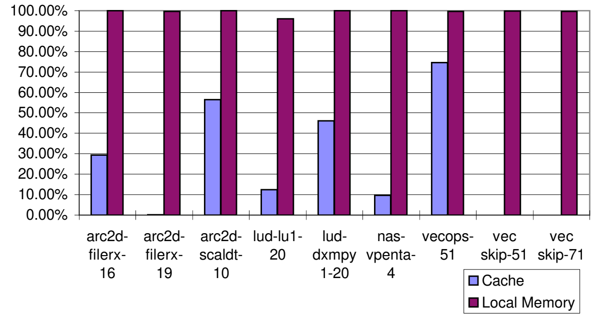

Figure 7: Read hit ratio and no-stall ratio for original and local memory loop nests, respectively.

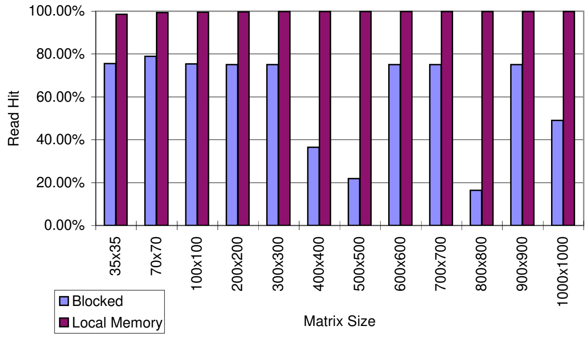

Figure 8: Read hit ratio and no-stall ratio for loop nest in Figure 5(a) with various matrix sizes

of primitive vector calculations as functions of length. `VECSKIP` performs the same operations as `VECOPS`, except that vectors are accessed in noncontiguous memory locations; the stride of access is 23. Loop nest 51 is a `DAXPY` loop, and loop nest 71 is the same loop with the scalar replaced a vector. All loop nests operate on array data and are representative of computations found in scientific applications.

We used SUIF [21] to perform general loop level transforms, including unimodular transforms and tiling transforms. SUIF is capable of determining the block size for a loop nest given the cache size and cache line size. Assembly code is generated using the SGI Fortran compiler, version 4.0.2. The Fortran compiler performs optimizations such as loop-unrolling, software pipelining, and code scheduling. The code thus generated forms our base code. The base code is transformed for local memory by hand. In cases where transformations performed by SUIF do not result in performance improvements for the base code, we use the original loop nest as the base code.

5.3 Simulation Results

Figure 6 shows the speedups obtained for the loop nests when run under the local memory versus using a normal cache.

The speedups obtained range from 1.4 to 2.6. Transformations performed by SUIF result in a high performance gain for the base code of `arc2d-scaldt-10`, resulting in a lower speedup under our scheme. `vecops-51` is a single loop, with no data reuse and a high cache hit ratio due to unit-stride access; there is no overlap between computation and fetching of data, and the overall speedup is low. But when the computations are performed in `vecskip-51` with non-unit stride data access, the speedup is much higher. In `vecskip-71`, the scalar is replaced by a vector, thereby reducing the computation to memory reference ratio; thus, fetching of data lags behind the computation and reduces the speedup.

Figure 7 shows the cache read hit ratio (gray bar) of the original loop nests and the non-stall read ratio for the local memory transformed loop nests (black bar). A non-stall read is one which does not have to wait for the block transfer unit to finish fetching the data. We do not show store misses, since they do not have any significant impact on the performance for the cache configuration we are using. Some of the kernels have a very low read hit ratio of zero. Except for `lud-lu1-20`, all loop nests have almost 100% no-stall read ratio. A lower no-stall ratio reflects that prefetching of data is lagging behind the computations.

Figure 8 shows the read hit and no-stall read ratio for the loop nests shown in Figure 5. The base code is a blocked version of the loop nest shown in Figure 5(a). The access stride is equal to the dimension of the matrix, and the block size is constant. The local memory version was appropriately sectioned. The results show a large variation in read hit ratio for different matrix sizes and access strides; however, the no-stall read ratio for the local memory remains constant.

The local memory transforms introduce additional

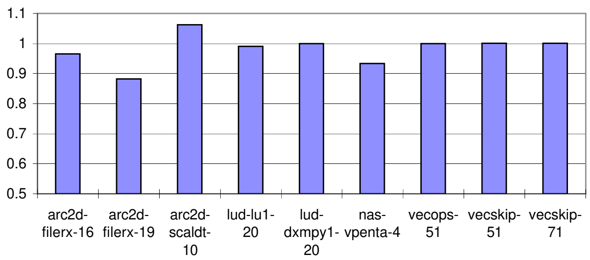

Figure 9: Local memory instruction count as a fraction of base instruction count

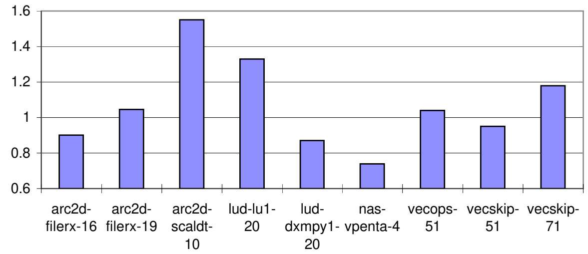

Figure 10: Local memory off-chip memory request count as a fraction of base memory request count

instructions into loops, but the reduction in loop overhead, as discussed in Section 4, frequently offsets the extra instructions. Figure 9 shows the local memory instruction count as a fraction of base instruction count. In some cases, the reduction in loop overhead is significant. In the case of `arc2d-scaldt-10`, the arrays referenced occupy different positions during the same iteration; thus, the loop overhead cannot be reduced by using the same address register for addressing different arrays; however, the additional instruction overhead is still only around 6%.

In general, prefetching increases memory traffic [6]. Figure 10 shows the off-chip memory request count in the local memory loop nests as a fraction of the base count. Except for `lud-lu1-20`, all the loop nests either show a moderate increase or a reduction in off-chip memory traffic. Though we don't present results for the same benchmarks, the increase in bus-traffic under our scheme is much lower than what is reported by Chen et al. [6].

We define the *utilization index* as the ratio of the number of read requests made to the cache/local memory and the number of data word brought into the cache/local memory. Figure 11 shows the utilization index for a conventional cache (gray bars) and the local memory (black bars). The cache utilization index is low except for loop nests with unit-stride references, such as `vecops-51`. For non-unit stride references, the cache utilizations index is always below 0.5, indicating

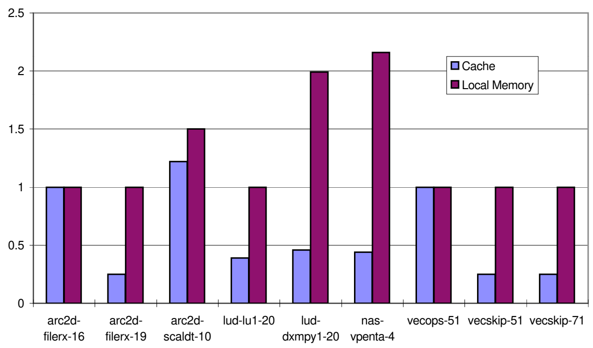

Figure 11: Utilization index for cache and local memory

that at least half the data brought into the cache is never used. In contrast, the local memory utilization index is always greater or equal to one; data is brought into the local memory only if it is needed.

`lud-lu1-20` achieves a speedup of 1.8, but at the expense of higher memory traffic. It interchanges two rows in a matrix, and the results shown here are for two adjacent rows. Access into a row is at a non-unit stride for a column-major memory layout. Since two rows are adjacent, their corresponding elements are at a unit-stride; thus the loop nest can exploit spatial locality. However, in the local memory, each row is represented as a separate block irrespective of the relative distance. Thus, our scheme can take advantage of the spatial locality within a row, but not between the rows of the matrix. However, as the distance between two rows increases, the performance of a conventional cache will degrade, but the performance of the local memory will remain constant.

6 Future Work

The transformations performed here are done by hand. Our next goal is to develop an effective compiler. A compiler would enable us to study the impact of the local memory on whole programs instead of loop nests. Beinitez et al. [5] discuss mechanisms for code generation for streaming. Streams are structured data stored in memory with a known, fixed displacement between successive elements. Our block fetch unit streams the data at a fixed stride from the secondary cache to the local memory. Techniques discussed by Allen et al. [1] are directly relevant to our local memory scheme.

We intend to extend our scheme to non-scientific applications which do not have good cache performance; for example, heap allocation with copying garbage collection. In this paper, we demonstrated the effectiveness of our scheme on regular, vectorizable scientific applications; we also intend to extend our scheme to scientific applications with irregular data access patterns.

7 Conclusion

In this paper, we have studied the performance of a hardware-based, compiler-assisted scheme to use cache as a local memory. By our quantitative evaluation, we have shown that using cache as a local memory results in much improved cache performance. The performance gains were achieved with minimum additional hardware overhead and with the addition of only a few new instructions. The instruction overhead in implementing in the scheme is insignificant; our results show that, unlike prefetching, the extra memory traffic generated is low. Since only useful data is brought into the local memory, in some cases there is a significant reduction in memory traffic. Lastly, we show that using cache as a local memory is an effective way of eliminating interference misses, and of making the cache performance immune to varying strides of data references.

References

[1] Randy Allen and Ken Kennedy. Vector register allocation. *IEEE Transactions on Computers*, 41(10):1290–1317, October 1992.

[2] J. L. Baer and T. F. Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Supercomputing'91*, pages 176–186, 1991.

[3] David H. Bailey. Unfavorable strides in cache memory system. RNR Tech. Report RNR-92-015, Nasa Ames Research Center, 1992.

[4] David H. Bailey. Risc microprocessors and scientific computing. In *Supercopmuting'93*, pages 645–654, 1993.

[5] M. E. Beinitez and Jack W. Davidson. Code generation for streaming an access/execute mechanism. In *Proceedings of the Fourth Conference on Architectural Support for Programming Languages and Operating Systems*, pages 132–141, April 1991.

[6] T. Chen and Jean-Loup Baer. A performance study of software and hardware data prefetching schemes. In *Proceedings of the 21th International Symposium on Computer Architecture*, pages 223–232, 1994.

[7] W. Y. Chen, S. A. Mahlke, P. P. Chang, and Wen mei W. Hwu. Data access microarchitectures for superscalar processors with compiler-assisted data prefetching. In *Proc. 24th Annual International Symposium on Microarchitecture*, pages 69–73. IEEE, Dec. 1991.

[8] F. Dahlgren and P. Stenstrom. Effectiveness of hardware-based stride and sequential prefetching in shared-memory multiprocessors. In *First International Symposium on High-Performance Computer Architecture*, pages 68–77, 1995.

[9] A. Diwan, D. Tarditi, and E. Moss. Memory subsystem performance of programs using copying garbage collection. In *21st Symposium on Principles of Programming Languages*, pages 1–13. ACM, January 1994.

[10] J. W. C. Fu and J. H. Patel. Data prefetching in multiprocessor vector cache memories. In *Proc. of the 18th International Symposium on Computer Architecture*, pages 54–63, June 1991.

[11] John W. C. Fu, J. H. Patel, , and Bob L. Janssens. Stride directed prefetching in scalar processors. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 102–110. IEEE, Dec. 1992.

[12] Mike Johnson. *Superscalar Microprocessor Design.* Prentice Hall, 1990.

[13] M. S. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimization of blocked algorithms. In *Fourth Int'l Conf. on Architectural Support for Prog. Lang. and Operating Systems*, pages 63–74, Apr. 1991.

[14] James Larus. Spim s20: A mips r2000 simulator, 1993.

[15] Sally A. McKee and Wm. A. Wulf. Access ordering and memory-concious cache utilization. In *First International Symposium on High-Performance Computer Architecture*, pages 253–262, 1995.

[16] MIPS Technologies, Inc., http://www.mips.com/r10k/T5_B.html. *MIPS R10000 Microprocessor User Manual – Version 1.0.*

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

[18] G. S. Sohi and W. C. Hsu. The use of intermediate memories for low-latency memory access in supercomputer scalar units. *The Journal of Supercomputing*, 4:5–21, 1990.

[19] O. Temam, C. Fricker, and W. Jalby. Impact of cache interferences on usual numerical dense loop nests. *Proceedings of the IEEE*, 81(8):1103–1115, Aug. 1993.

[20] O. Temam, E. D. Granston, and W. Jalby. To copy or not copy: A compile-time technique for assessing when data copying should be used to eliminate cache conflicts. In *Supercomputing'93*, pages 410–419, 1993.

[21] R. Wilson, R. French, C. Wilson, S. Amarasinghe, J. Anderson, S. Tjiang, S.-W. Liao, C.-W. Tseng, M. Hall, M. Lam, and J. Hennessy. The SUIF compiler system: A parallelizing and optimizing compiler. Technical Report CSL-TR-94-620, Computer Systems Laboratory, Stanford University, 1994.

[22] Yoji Yamada. *Data Relocation and Prefetching for Programs with Large Data Sets*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1995.

[23] Z. Zhang and J. Torrellas. Speeding up irregular applications in shared-memory multiprocessors: Memory binding and group prefetching. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 188–199, 1995.

[24] Hans Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers*. Addison Wesely, 1990.