Improving load/store queues usage in scientific computing

Christophe Lemuet, William Jalby, Sid Touati

Improving Load/Store Queues Usage in Scientific Computing

Christophe LEMUET William JALBY Sid-Ahmed-Ali TOUATI
PRiSM Laboratory, University of Versailles, France

Abstract

Memory disambiguation mechanisms, coupled with load/store queues in out-of-order processors, are crucial to increase Instruction Level Parallelism (ILP), especially for memory-bound scientific codes. Designing ideal memory disambiguation mechanisms is too complex because it would require precise address bits comparators; thus, modern microprocessors implement simplified and imprecise ones that perform only partial address comparisons. In this paper, we study the impact of such simplifications on the sustained performance of some real processors such that Alpha 21264, Power 4 and Itanium 2. Despite all the advanced features of these processors, we demonstrate in this article that memory address disambiguation mechanisms can cause significant performance loss. We demonstrate that, even if data are located in low cache levels and enough ILP exist, the performance degradation can be up to 21 times slower if no care is taken on the order of accessing independent memory addresses. Instead of proposing a hardware solution to improve load/store queues, as done in [1, 6, 5, 7, 4], we show that a software (compilation) technique is possible. Such solution is based on the classical (and robust) ld/st vectorization. Our experiments highlight the effectiveness of such method on BLAS 1 codes that are representative of vector scientific loops.

1. Introduction

Memory system performance is essential to today's processors. Therefore, computer architects have spent, and are still spending, large efforts in inventing sophisticated mechanisms to improve data access rate, in terms of latency and bandwidth: multi-level and non-blocking caches, ld/st queues for out-of-order execution, prefetch mechanisms to tolerate/hide memory latencies, banking and interleaving to increase bandwidth, etc.

One key mechanism to tolerate/hide memory latency is the out-of-order processing of memory requests. With the advent of superscalar processors, the concept of ld/st queues has become a standard. The basic principle is simple: consecutive issued memory requests are stored in a queue and simultaneously processed. This allows the requests with shorter processing time (in the case of cache hits) to bypass requests with a longer processing time (in the case of cache misses for example). Unfortunately, data dependences may exist between memory requests: for instance, a load followed by a store (or vice-versa) addressing both exactly the same memory location have to be executed strictly in order to preserve program semantics. This is done on-the-fly by specific hardware mechanisms whose task is, first, to detect memory request dependences and, second, to satisfy such dependences (if necessary). These mechanisms are under high pressure in memory-bound programs, because numerous "in-flight" memory requests have to be treated.

In order to satisfy this high request rate, memory dependence detection mechanisms are simplified at the expense of accuracy and performance [3]. To be accurate, the memory dependence detection must be performed on complete address bits ; this might be complex and expensive. In practice, the comparison between two accessed memory locations is carried out on a short part of the addresses: usually, few low order bits. If these low order bits match, the hardware takes a conservative action, i.e., it considers that the whole addresses match and triggers the procedure for a collision case (serialization of the memory requests).

In this article, we study in details the dynamic behavior of memory request processing on three modern processors (Alpha 21264, Power 4, Itanium 2). Because of the high complexity of such analysis, our work is focused on the different memory hierarchy levels (L1, L2, L3), excluding the main memory. Our benchmarking codes are simple floating point vector loops (memory-bound) which account for a large fraction of execution time in our scientific computing target area. Additionally, the structure of their address streams is regular,

making it possible a detailed performance analysis of the interaction between these address streams with the dependence detection mechanisms and bank conflicts. Our aim is not to analyze or optimize a whole program behavior, but only small fractions that consist of simple scientific computing loops (libraries). One of the reasons is that the ld/st queue conflicts that we are interested in are *local* phenomena because, first, they strictly involve in-flight instructions (present in the instruction window). Second, they are not influenced by the context of a whole application as other events such as caches activities. So, there is no need to run complete complex applications to isolate these local events that we can highlight with micro-benchmarking (explained in Sect. 3). Third and last, the number of side effects and pollution of the cache performance in whole complex applications (such as SPEC codes) makes the potential benefits smoothened. We will show that our micro-benchmarks are a good diagnostic tool. We can precisely quantify the effects of load/store vectorization on poor memory disambiguation. It allows us to reveal the limitations of the dynamic memory dependences check ; they may lead to severe performance loss and make the code performance very dependent on the order of independent memory accesses.

We organize our article as follows. Sect. 2 presents some related work about the problem of improving ld/st queues and memory disambiguation mechanisms. Sect. 3 gives a description of our experimental environment. Then, Sect. 4 shows the most important results of our experiments that highlight some problems in modern cache systems, such that memory dependence detection mechanisms and bank conflicts. We propose in Sect. 5 an optimization method that groups the memory requests in a vectorized way. We demonstrate by our experiments that this method is effective, then we conclude.

2. Related Work

Improving ld/st queues and memory disambiguation mechanisms is an issue of active research. Chrysos and Emer in [1] proposed store sets as a hardware solution for increasing the accuracy of memory dependence prediction. Their experiments were conclusive by demonstrating that they can nearly achieve the peak performance with the context of large instruction windows. Park *et al* in [5] proposed an improved design of ld/st queues that scale better, that is, they have improved the design complexity of memory disambiguation. Another similar hardware improvement has been proposed by Sethumadhavan *et al* in [5]. A speculative technique for memory dependence prediction has been proposed by Yoaz *et al* in [7]: the hardware tries to predict colliding loads, relying on the fact that such loads tend to repeat their delinquent behavior. Another speculative technique devoted to superscalar processors was presented by S. Onder [4]. The author presented a hardware mechanism that classifies the loads and stores to an appropriate speculative level for memory dependence prediction.

All the above sophisticated techniques are hardware solutions. In the domain of scientific computing, the codes are often regular, making it possible to achieve effective compile time optimizations. Thus, we do not require such costly dynamic techniques. In this paper, we show that a simple ld/st vectorization is useful (in the context of scientific loops) to solve the same problems tackled in [1, 5, 7, 4]. Coupling our costless software optimization technique with the actual imprecise memory disambiguation mechanisms is less expensive than pure hardware methods, giving nonetheless good performance improvement.

3. Experimental Environment

In order to analyze the interaction between the processors (typically the cache systems) with the applications, we have designed a set of micro-benchmarks, as presented in the following section.

3.1. Our Micro-Benchmarks

Our set of micro-benchmarks consists of simple vector loops (memory-bound) which consume large fractions of execution times in scientific applications. Besides their representativity, these vector loops present two key advantages : first, they are simple, and second they can be easily transformed since they are fully parallel.

We divide our micro-benchmarks into two families:

1. *memory stress kernels* are artificial loops which aim to only send consecutive bursts of independent loads and stores in order to study the impact of memory address streams on the peak performance.[1] Such loops do not contain any data dependences.
   (a) the first one, called *LxLy*, corresponds to a loop in which two arrays X and Y are regularly accessed with only loads : `Load X(0)`,

[1] In this paper, the peak performances refers to the ideal one, i.e., the maximal theoretical performance as computed from hardware specifications.

`Load Y(0)`, `Load X(1)`, `Load Y(1)`, `Load X(2)`, `Load Y(2)`, etc.

(b) the second one, called *LxSy* corresponds to a loop in which one array X is accessed with loads, while the Y one is accessed with stores: `Load X(0)`, `Store Y(0)`, `Load X(1)`, `Store Y(1)`, `Load X(2)`, `Store Y(2)`, etc.

2. *BLAS 1 kernels* are simple vector loops that contain flow dependences. In this article, we use three simple loops:

   - copy : `Y(i)` $\leftarrow$ `X(i)`;
   - vsum : `Z(i)` $\leftarrow$ `X(i)` + `Y(i)`;
   - daxpy : `Y(i)` $\leftarrow$ `Y(i)` + a $\times$ `X(i)`;

Despite the fact that we have experimented other BLAS 1 codes with various number of arrays, we chose these simple ones as illustrative examples, since they clearly exhibit the pathological behavior that we are interested in.

3.2. Experimentation Methodology

The performance of our micro-kernels are sensitive to several parameters that we explore. We focus in this paper on two major ones which are:

1. **Absolute Array Offsets**: the impact of the exact starting address of each array[2] is analyzed. This is because varying such offsets changes the accessed addresses of the vector elements, and thus it has a significant impact on ld/st queues behavior: we know that the memory address of the double floating point element X(i) is Offset$(X) + 8 \times i$.

2. **Data Location**: since we are interested in exploring the cache performance (L1, L2, L3), we parameterize our micro-kernels in order to lock all our arrays in the desired memory hierarchy level. By choosing adequate vector lengths, and by using dummy loops that flush the data from the non desired cache levels, we guarantee that our array elements are located exactly in the experimented cache level (checked by hardware performance counters).

Some other parameters, such that prefetch distances, have been carefully analyzed. However, because of the lack of space, we restrict ourselves in this paper to the two parameters described above. Prefetch distances are fixed to those that produce the best performances. Note that in all our experiments, the number of TLB misses is extremely negligible.

After presenting the experimental environment, the next section studies the performance of the memory hierarchy levels in various target processors.

[2] It is the address of the first array element that we simply call the array offset. The address zero is the beginning of a memory page.

4. Experimental Study of Cache Behavior

This section presents a synthesis of our experimental results on three micro-processors: Alpha 21264, Power 4 and Itanium 2. Alpha 21264 and Power 4 are two representative ones of out-of-order superscalar processors, while Itanium 2 represents an in-order processor (an interesting combination between superscalar and VLIW).

In all our experiments, we focus on the performance of our micro-benchmarks expressed in terms of number of clock cycles (execution time), reported by the hardware performance counters available in each processor. Our measurements are normalized as follows:

- in the case of memory stress kernels, we report the minimal number of clock cycles needed to perform two memory accesses: depending on the kernel, it might be a pair of loads (LxLy kernel), or a load and a store (LxSy kernel);

- in the case of BLAS 1 kernels, we report the minimal number of clock cycles needed to compute one vector element. For instance, the performance of the vsum kernel is the minimal time needed to perform one instruction `Z(i)`$\leftarrow$`X(i)+Y(i)`. Since all our micro-benchmarks are memory-bound, the performance is not sensitive to any floating point computations.

One of the major point of focus is the impact of array offsets on the performance. Since most of our micro-benchmarks access only two arrays (except vsum that accesses three arrays), we explore the combination of two dimensions of offsets (offset X vs. offset Y). Therefore, 2D plots (ISO-surface) are used. A *geographical* color code is used: light colors correspond to the best performance (lowest number of cycles) while dark colors correspond to the worst performance. The legends show the discrete scales of the performances (minimal number of clock cycles). For instance, [1-3] means that the number of clock cycles belongs to this interval.

In the following sections, we detail the most important and representative experiments that allow to make a clear synthesis on each hardware platform.

4.1. Alpha 21264 Processor

The plot in Fig. 1 uses intensity to encode the performance of the LxSy kernel for each combination of array offsets. The lighter regions of the graph represent 1.3 cycle per iteration. In the darkest diagonal in the figure, the performance degradation is greater than 23 times when compared with the best one. This main diagonal corresponds to the effects of the interactions between a stream of a load followed by a store, both accessing two distinct memory locations (`Load X(i)` followed by `Store Y(i)`). However, the hardware assumes that these memory operations are dependent because they have the same k address lower-bits. This diagonal is periodic (not reported in this figure) and arises when the offset of X (resp. Y) is a multiple of 32 KB, which means that $k = 15$ bits. The magnitude of performance degradation depends on the frequency of the false memory collisions, and the distance between them : the nearer is the issue time of two false colliding memory addresses, the higher is the penalty. The degradation in the secondary diagonal on the upper left of Fig. 1 corresponds to 11 cycles per iteration. It is due to the effects of interactions between the prefetch instructions of the X elements and the stores of Y elements. The periodicity of this diagonal is 32 KB too.

These performance penalties occur for all BLAS 1 kernels. This is due to the compiler optimization strategy. Indeed, the Compaq compiler (version 6.3) generates a well optimized code (loop unrolling with fine-grain scheduling) but keeps the same order of memory access as described by the C program (`Load X(i)` followed by `Store Y(i)`). This code generation allowed to reach peak performance only with ideal combination of array offsets, that are not controlled by the compiler.

4.2. Power 4 Processor

For this processor, we show the performance of some BLAS 1 kernels. The IBM compiler (version 5.02) generates also a well optimized code. The loops are unrolled and optimized at the fine-grain level, but they perform the same order of the memory accesses as described by the source program (`Load X(i)` followed by `Store Y(i)` for copy kernel, and `Load X(i)`, `Load Y(i)` followed by `Store Z(i)` for vsum). Prefetch instructions are not inserted by the compiler, since data prefetching is automatically done by hardware.

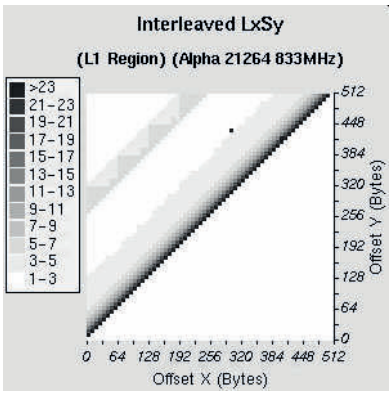


Figure 1. Cache Behavior of Alpha 21264

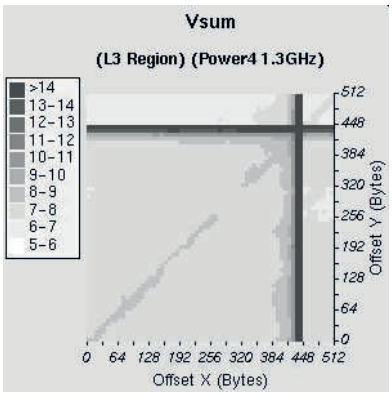


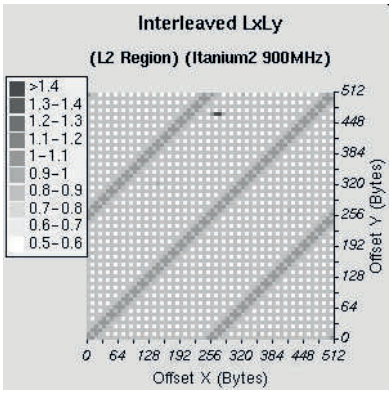
Figure 2. Cache Behavior of Power 4

Fig. 2 plots the performance of vsum code when the operands are located in L3. This figure is more complex:

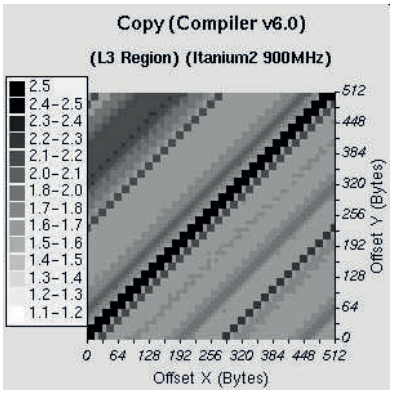
- along the main diagonal, a stripe is visible with a moderate performance loss (around 17 %). This is due to the interaction between the two load address streams (loads of X and Y elements).

- a vertical (resp. horizontal) stripe can be observed where the execution times are larger (25 cycles instead of 7). This is due to the interaction between the loads of X elements (resp. Y elements) with the stores of Z elements.

In all cases, the "bad" vertical and diagonal zones appear periodically every 4 KB offset. It confirms that the processor performs partial address comparison on 12 low-order bits.



(a)



(b)

Figure 3. Cache Behavior of Itanium 2

4.3. Itanium 2 Processor

Contrary to the two previous processors, Itanium 2 is in-order. The instruction level parallelism is expressed by the program using instruction groups and bundles. Thus, analyzing the behavior of the memory operations is little easier. In this section, we show that the banking architecture of the L2 cache level and the memory disambiguation mechanisms may cause significant performance degradation. While the memory stress kernels are coded at low level, we use the Intel compiler (version version 7.0 beta) to generate optimized codes for our BLAS 1 loops. Software pipelining, loop unrolling and data prefetching are used to enhance the fine-grain parallelism. The experiments are performed in L2 and L3, since on Itanium 2, L1 cannot contain floating point operands (this is a design choice of the Itanium family architecture).

Processor	L1	L2	L3
Alpha 21264	21.54	12	-
Power 4	2.11	3.64	2.57
Itanium 2	-	2.17	1.5

Table 1. Performance Degradation Factors

First, let us examine the impact of L2 banking architecture. Fig. 3(a) plots the performance of the LxLy kernel (two streams of independent loads). The best execution time is 0.6 cycle, which is the optimal one. However, some regions exhibit performance loss, depending on the array offsets. Basically, two types of phenomenon can be observed:

1. three diagonals separated by 256 B in which the performance is 1.2 cycle instead of 0.6 cycle;
2. a grid pattern (crossed by the three diagonal stripes). Inside this grid, the execution times in some points are 0.6 cycle, but 1 cycle in others.

Both phenomena can be easily attributed to bank conflicts resulting from the interactions between the L2 interleaving scheme and the address streams.

All the performance troubles observed in L2 still exist in L3 level. Fig. 3(b) shows the performance of the copy kernel. The memory disambiguation problem is accentuated (wider diagonal stripes) because of the interaction between independent loads and stores. Another problem is highlighted by the upper-left diagonal zone, which is in fact due to the interferences between prefetch instructions (that behave as loads) and the store instructions.

Summary This section presented the behavior of the cache systems in Alpha 21264, Power 4 and Itanium 2 processors. We showed that the effectiveness of an enhanced instruction scheduling is not sufficient to sustain the best performance even in very simple codes, when we expect a maximal ILP extraction. We demonstrated that memory disambiguation mechanisms cause significant performance loss depending on array offsets. Bank conflicts in Itanium 2 are also an important source of performance troubles. Tab. 1 recapitulates the performance degradation factors[3] caused by these micro-architectural restrictions.

We can use many code optimization techniques to reduce the performance penalties previously exposed (for

[3]Counted as the ratio between the best and worst performance.

instance, array padding, array copying and code vectorization). In the next section of this article, we investigate the impact of vectorization.

5. The Effectiveness of Ld/St Vectorization

The performance degradation depicted in the last section arises when a program performs parallel accesses to distinct arrays. Theoretically, if the underlying processor has enough functional units (FUs), and if the different caches have enough ports, such memory operations can be executed in parallel. Unfortunately, for implementation reasons (design complexity), memory disambiguation mechanisms in actual ILP processors do not perform complete comparisons on address bits. Furthermore, some caches, as those implemented on Itanium 2, contain several banks and do not allow to sustain full access bandwidth. Thus, parallel memory operations are serialized during execution, even if enough FUs and ILP exist, and even if data are located in low cache levels.

Let us think about new ways to avoid the dynamic conflicts between memory operations. One of the ways to reduce these troubles is ld/st vectorization. This is not a novel technique, and we do not aim to bring a new one; we only want to demonstrate that the classical vectorization is a simple and yet elegant solution to a difficult problem. We schedule memory access operations not only according to data dependences and resources constraints, but we must also take into account the accessed address streams (even if independent). Since we do not know the exact array offsets at compile time, we cannot determine precisely all memory locations (physical addresses) that we access. However, we can rely on their relative address locations as defined by the arrays. For instance, we can determine at compile time the relative address between `X(i)` and `X(i+1)`, but not between `X(i)` and `Y(i)` since array offsets are determined at linking time in the case of static arrays, or at execute time in the case of dynamically allocated arrays. Thus, we are sure at compile time that the different addresses of the elements `X(i)`, `X(i+1)`,..., `X(i+k)` do not share the same lower-order bits. This fact makes us group memory operations accessing to the same vector since we know their relative address. Such memory access grouping is similar to vectorization, except that only loads and store are vectorized. The other operations, such that the floating point ones, are not vectorized, and hence they are kept free to be scheduled at the fine-grain level to enhance the performance.

Vectorization is a complex technology, and many studies have been performed on this scope. In our framework, the problem is simplified since we tackle fully parallel innermost loops. We only seek a convenient vectorization degree. Ideally, the higher is this degree, the higher is the performance, but the higher is the register pressure too. Thus, we are constrained by the number of available registers. We showed in [2] how we can modify the register allocation step by combining ld/st vectorization at the data dependence graph (DDG) level without hurting ILP extraction. This previous study [2] shows how we can seek a convenient vectorization degree which satisfies register file constraints and ILP extraction. To simplify the explanation, if a non-vectorized loop consumes r registers, then the vectorized version with degree k requires at most $k \times r$ registers. Thus, if the processor has $\mathcal{R}$ available registers, a trivial valid vectorization degree is $k = \lfloor \frac{\mathcal{R}}{r} \rfloor$. The next sections explore the effectiveness of ld/st vectorization.

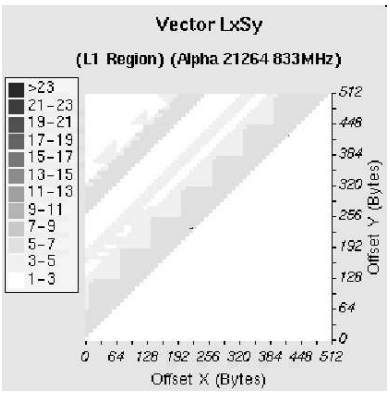


Figure 4. Vectorization on Alpha 21264

5.1. Alpha 21264 Processor

Fig. 4 shows the impact of vectorization on the LxSy kernel (compare it to Fig. 1). Even if all the troubles do not disappear, the worst execution times in this case are less than 7 cycles instead of 28 cycles previously. The best performance remains the same for the two versions, i.e., 1.3 cycle. This improvement is confirmed for all BLAS 1 kernels and in all cache levels. Tab. 2 presents the speedup resulted from vectorization. It is counted as the gain ratio between the worst performance of the vectorized codes and the worst performance of the original codes. The best performance of all the micro-benchmarks are not altered by vectorization.

Cache	LxLy	LxSy	Copy	Vsum	Daxpy
L1	0%	53.57%	45.83%	80%	29.17%
L2	26.32%	75%	48.15%	80%	30.77%

Table 2. Worst-Case Performance Gain on Alpha 21264

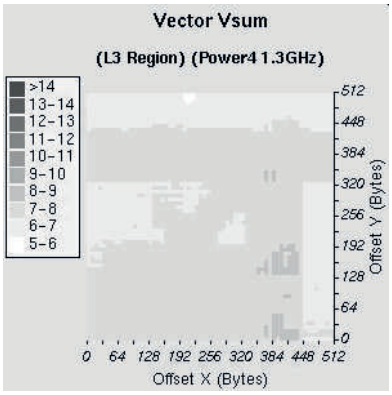


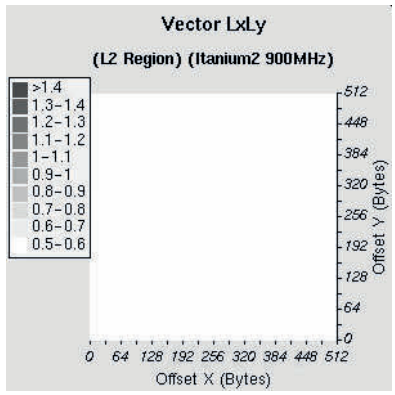
Figure 5. Vectorization on Power 4

5.2. Power 4 Processor

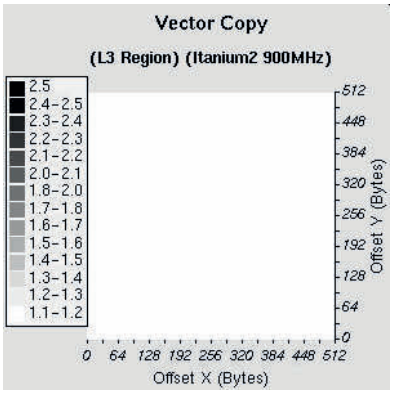
Fig. 5 shows the performance of vectorized vsum kernel when the operands are located in L3 (compare it to Fig. 2). As it can be seen, all the stripes of bad performance disappear. Vectorizing memory operations improves the worst performance of all our micro-benchmarks in all cache levels by reducing the number of conflicts between the memory operations. The best performance of all the micro-benchmarks are not degraded by vectorization.

5.3. Itanium 2 Processor

The case of Itanium 2 processor needs more efforts since there are bank conflicts in addition to imprecise memory disambiguation. Thus, the ld/st vectorization is not as naive as for the previous out-of-order processors. In order to eliminate bank conflicts, memory access operations are packed into instruction groups that access even or odd indexed vector elements. For instance `Load X(i)`, `Load X(i+2)`, `Load X(i+4)`,... and `Load X(i+1)`, `Load X(i+3)`, `Load X(i+5)`, etc. Thus, each instruction group accesses a distinct cache bank. Since each bank can contain 16 bytes of consecutive data, two consecutive double FP elements may be assigned to the same bank. This fact prohibits accessing both elements at the same clock cycle (bank conflict). This is why we grouped the accesses in an odd/even way. Fig. 6(a) plots the performance of the vectorized LxLy kernel (compare it to Fig. 3(a)). As it can be seen, all bank conflicts and memory disambiguation problems disappear. The sustained performance is the peak one (ideal) for any vector offsets. When stores are performed, Fig. 6(b) shows the L3 behavior for the vectorized copy kernel (compare it to Fig. 3(b)). The original grid patterns are smoothed.



(a)



(b)

Figure 6. Vectorization on Itanium 2

This improvement occurs for all our micro--benchmarks and in all cache levels. Tab. 3 shows the speedup resulted from vectorization, counted as the gain ratio between the worst performance of the vectorized codes and the worst performance of the original codes.

Cache	LxLy	LxSy	Copy	Daxpy
L2	45.45%	18.18%	47.62%	40.91%
L3	28.57%	18.75%	54.55%	33.33%

Table 3. Worst-Case Performance Gain on Itanium 2

Again, ld/st vectorization does not alter the peak performance in all cases.

6. Conclusion and Future Work

Memory-bound programs rely on advanced compilation techniques that try to keep data into the cache levels, hoping to fully utilize a maximal amount of ILP on the underlying hardware functional units. Even in ideal cases when operands are located in lower cache levels, and when compilers generate codes that can statically be considered as "good", our article demonstrates that it is not sufficient for sustaining the peak performance.

First, the memory disambiguation mechanisms in modern ILP processors do not perform comparisons on whole address bits. If two memory operations access two distinct memory locations but share the same lower-order bits in their addresses, the hardware detects a false dependence and triggers a serialization mechanism. Consequently, ld/st queues cannot be fully utilized to re-order the independent memory operations.

Second, the banking structure of some caches prevent from sustaining entire access bandwidth. If two elements are mapped to the same bank, independent loads are restricted to be executed sequentially, even if enough FUs are idle. This fact is a well known source of troubles, but current compilers still does not take it into account (even with highly optimized, hand tuned, libraries provided by vendors), and the generated codes can be 2 times slower on Itanium 2.

Our study demonstrates that a simple existing compilation technique can help to generate faster codes that reduce the ld/st queue conflicts. Consecutive accesses to the same array are grouped together since we know at compile time their relative addresses. Coupling our simple vectorization technique with other classical ILP scheduling ones is demonstrated to be effective to sustain the peak performance. Even if we do not avoid all situations of bad relative array offsets in all hardware platforms, and thus few memory disambiguation penalties persist, we showed that we still get high speedups in all experimented processors (up to 54% of performance gain). This simple software solution coupled with imprecise memory disambiguation mechanisms are less expensive than sophisticated totally hardware approaches such as [1, 6, 5, 7, 4].

Vectorization is not the only way that may solve the performance penalties highlighted in this paper. Array padding for instance can change the memory layout in order to produce ideal array offset combinations. However, array padding requires to analyze the whole application. In the case of scientific libraries on which we are focusing, we cannot apply this technique since the arrays are declared outside the functions (not available at the compilation time of the library).

In a future work, we will explore the main memory behavior with the same methodology. Already, some preliminary tests have confirmed us with the good performance capabilities of the vectorization strategy. A second future work may be devoted to generalize our vectorization methodology to take into account more complex scientific codes.

References

[1] G. Chrysos and J. Emer. Memory Dependence Prediction using Store Sets. In *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA-98)*, volume 26,3 of *ACM Computer Architecture News*, pages 142–154, New York, June 1998. ACM Press.

[2] W. Jalby, C. Lemuet, and S.-A.-A. Touati. An Efficient Memory Operations Optimization Technique for Vector Loops on Itanium 2 Processors. *Conucurrency and Computation: Practice and Experience*, 2004 (to appear). Wiley Interscience.

[3] M. Johnson. *Superscalar Microprocessor Design*. Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[4] S. Onder. Cost Effective Memory Dependence Prediction using Speculation Levels and Color Sets. In *2002 International Conference on Parallel Architectures and Compilation Techniques (PACT'02)*, Virginia, Sept. 2002. IEEE.

[5] I. Park, C. L. Ooi, and T. N. Vijaykumar. Reducing Design Complexity of the Load/Store Queue. In *Proceedings of the 36th International Symposium on Microarchitecture (MICRO-36 2003)*, San Diego, Dec. 2003. IEEE.

[6] S. Sethumadhavan, R. Desikan, D. Burger, C. R. Moore, and S. W. Keckler. Scalable Hardware Memory Disambiguation for High ILP Processors. In *Proceedings of the 36th International Symposium on Microarchitecture (MICRO-36 2003)*, San Diego, Dec. 2003. IEEE.

[7] A. Yoaz, M. Erez, R. Ronen, and S. Jourdan. Speculation Techniques for Improving Load Related Instruction Scheduling. In *26th Annual International Symposium on Computer Architecture (26th ISCA'99), Computer Architecture News*, volume 27, pages 42–53. ACM SIGARCH, May 1999.