An Efficient Memory Operations Optimization Technique for Vector Loops on Itanium 2 Processors

William JALBY, Christophe LEMUET, Sid-Ahmed-Ali TOUATI

April 10, 2009

Abstract

To keep up with a large degree of instruction level parallelism (ILP), the Itanium 2 cache systems use a complex organization scheme: load/store queues, banking and interleaving. In this paper, we study the impact of these cache systems on memory instructions scheduling. We demonstrate that, if no care is taken at compile time, the non-precise memory disambiguation mechanism and the banking structure cause severe performance loss, even for very simple regular codes. We also show that grouping the memory operations in a pseudo-vectorized way enables the compiler to generate more effective code for the Itanium 2 processor. The impact of this code optimization technique on register pressure is analyzed for various vectorization schemes.

keywords Performance Measurement, Cache Optimization, Memory Access Optimization, Bank Conflicts, Memory Address Disambiguation, Instruction Level Parallelism.

1 Introduction

To satisfy the ever increasing data request rate of modern microprocessors, large multilevel memory hierarchies are no longer the unique efficient solution [8]: computer architects have to rely on more and more sophisticated mechanisms [10, 6], in particular, load/store queues (to decouple memory access and arithmetic operations), banking and interleaving (to increase bandwidth), prefetch mechanisms (to offset latency).

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 load/store queues has become a standard. The basic principle is simple: consecutively 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 (for example, in the case of a cache miss). Unfortunately, dependencies may exist between memory requests: for example, a load followed by a store (or vice-versa) both addressing 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 dependencies and, second (if necessary), to satisfy such dependencies. These mechanisms are under high pressure in memory-bound programs, because numerous “in-flight” memory requests have to be treated.

Another mechanism that tries to reduce the design complexity of cache systems is banking, that was already present in the old vector machines such as Cray XMP [19]. Instead of creating a large number of ports, and thus increasing the area complexity, caches are divided into multiple banks which can then be accessed simultaneously. Consequently, if some requested memory addresses are mapped to the same bank, these memory operations cannot be serviced at the same time, even if they are independent (a sequence of reads for instance). This potential side effect may be hidden in the presence of load-store queues with large capacity.

In order to satisfy this high request rate, memory dependence detection mechanisms are simplified at the expense of accuracy and performance [15]. To be accurate, the memory dependence detection must be performed on full address bits, which 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, a few low order bits. If these low order bits match, the hardware takes a conservative action, i.e., it considers that the full addresses match and triggers the procedure for a collision (serialization of the memory requests).

In this paper, we use the Itanium 2 architecture [13, 12, 24] as a target hardware platform. First, because it offers a large degree of instruction level parallelism (ILP) that is directly manageable by the compiler. Second, because its cache subsystem is complex: three cache levels, sophisticated prefetch capabilities, etc.

Due to the already complex nature of the problem, our study is currently restricted to the L2 and L3 cache subsystem[1] (excluding memory) and to simple vector codes (BLAS 1 type: Copy, Daxpy). Although they are simple, such as BLAS 1 codes are fairly representative of memory address streams in scientific computing. The choice of scientific computing as a target application area is motivated by the excellent match between scientific codes (easier to analyze statically) and the Itanium 2 architecture (well designed for exploiting static information).

Even with this limited scope in terms of application codes, our study reveals that performance behavior is rather complex and hard to analyze on real processors. In particular, the banking/interleaving structure of the L2 and L3 caches is shown to have a major interaction with the accessed address streams, potentially inducing large performance loss. Furthermore, the non-precise memory disambiguation mechanism of Itanium 2 is responsible for bad dynamic ILP utilization if no care is taken at compile time.

We demonstrate that vectorizing memory accesses allows us to get rid of most of the L2 and L3 cache bank conflicts. Unfortunately, this technique may increase register pressure. Therefore, such impact on register pressure is presented and evaluated.

Note that we do not aim to optimize whole complex applications, such as SPEC codes. This is because the potential benefits of general code optimization may be smoothened out in real applications, for the reason that cache performance is polluted by numerous side effects. Thanks to our approach that uses micro-benchmarks, the new architectural insights are being highlighted.

Our article is organized as follows. Section 2 describes our experimental setup: hardware platform as well as software platform (compiler and OS). In Section 3, our target codes and experimental methodology are presented. Section 4 highlights the performance bugs caused by the non-precise memory disambiguation and the banking structure of the Itanium 2 cache levels. In Section 5, we propose an optimization strategy for memory accesses based on a pseudo-vectorization. We give our experimental results to validate the effectiveness of such a method. Section 6 studies the impact of the ld/st vectorization on register pressure. Section 7 presents related work in the field of micro-benchmarking and improving ld/st queues and memory disambiguation mechanisms. Finally, some concluding remarks and directions for future work are given.

2 Experimental Setup

The machine used in our experiments is a real uniprocessor Itanium 2 based system, equipped with a 900MHz processor and 3GB memory. The general processor architecture (an interesting combination of VLIW and advanced speculative mechanisms) is described in [12, 24, 3, 13]. The Itanium 2 processor offers a wide degree of parallelism:

- six general purpose ALUs, two integer units, and one shift unit;
- the data cache unit contains four memory ports enabling a service of either four loads requests or two loads and two stores requests in the same clock cycle;

[1] On Itanium 2, L1 cannot hold FP operands.

- two floating point multiply-add units allowing the processor to execute two floating point multiply-add operations per cycle
- six multimedia functional units;
- three branch units, etc.

All the computational units are fully pipelined, so each functional unit (FU) can accept one new instruction per clock cycle (in the absence of stalls). Instructions are grouped together in blocks of three instructions (called a bundle). Up to two bundles can be issued per cycle. Due to the wealth of functional units, a rate of six instructions executed per cycle can be sustained.

In our test machine, the caches are organized in three levels: L1 (restricted by the Itanium 2 design not to store floating point data), L2 (unified, 256 KB, 8 way associative, write back allocate, 128 bytes cache line), and L3 (unified, 1.5 MB, 12 way associative).

The L2 is capable of supporting up to four memory accesses per cycle: either four loads, or two loads and two stores. The L2 cache is organized in 16 banks, with an interleaving of 16 bytes, that is, a bank can contain two double FP elements. For instance, the addresses from 0 to 15 are located in bank 0, and those from 16 to 31 in bank 1, etc[2]. Thus, two double FP elements that are 256 bytes apart reside necessarily in the same bank. In other words, two memory elements reside in the same bank if their address share the same bit fields $\{7{,}6{,}5{,}4\}$. Finally, the cache interface (for both L1 and L2) is equipped with a ld/st queue allowing cache hits to bypass cache misses (provided that the addresses are correctly disambiguated).

In addition to the standard load and store instructions on floating point operands (single and double precision), the Itanium instruction set offers a load floating pair instruction that is capable of loading 16 bytes at once, provided that the corresponding accessed memory address is aligned on a 16 byte boundary.

Our test machine is running Linux IA-64 Red Hat 7.1 based on the 2.4.18 smp kernel. The page size used by the system is 16 KB and we use the following compiler: Intel C++ Compiler Version 7.0 Beta, Build 20020703. Although various compiler options have been tested for our simple BLAS 1 kernels, it was found that the combination of -O3 and -restrict was the most powerful. These compiler flags allow us to use most of the advanced features of the Itanium 2 architecture: software pipelining, prefetch instructions, predication and rotating registers. In order to reach the peak performance, the “-restrict” option is essential because it let the compiler assume that distinct arrays are pointing to disjoint memory regions, therefore allowing it to perform a full static reordering of loads and stores. The Fortran compiler was also tested but, for our simple loops, the code generated was almost identical to the one obtained with the C language.

Besides the compiler, our hand optimized versions were also compared to the Intel optimized library MKL 6.0.

All our experiments have been conducted with the perfmon toolkit in order to use the various hardware performance counters. Thanks to these hardware counters, we were able to precisely compute the number of cycles needed to execute a program fraction. This is done by accessing the special register *ar.itc*. Also, other hardware counters allow us to systematicly check the validity of all our experiments. For instance, we can observe that the TLB miss ratio is almost zero. Also, we can check the data location (in which cache level the data reside) by analyzing the miss and hit ratio of each cache level, etc. In other words, we can ensure that we measured what we desired to measure, and that all our experiments were correct.

3 Target Codes and Performance Measurement

The BLAS 1 kernels are simple vector loops. In this article, the detailed results are given for two of them:

[2] Remember that the accessed addresses are multiples of 8 since we use double FP data.

1. copy: $Y(i) \leftarrow X(i)$;
2. daxpy: $Y(i) \leftarrow Y(i) + a \times X(i)$;

All the FP arrays are in double precision (8 bytes FP elements). That is, $X(i)$ denotes the i^{th} element of the array X. In our experiments, the array layout in the virtual memory space is tightly controlled. In particular, the impact of the starting address of each array X and Y is studied in depth. To achieve this goal, the parameters Offset X (respectively, Offset Y) are introduced, according to the following relations:

- Virtual address of X(i)= Offset X + 8×i;
- Virtual address of Y(i)= Offset Y + 8×i;

In our article, the offset of an array is defined as its relative starting address to a page boundary. Changing the values of Offset X and Offset Y allows one to change the L2 and L3 bank access pattern. Furthermore, it permits to change the accessed address streams in order to check the ability of the hardware to fully detect independent memory operations. We will see that the hardware does not react as it should do, even if we execute a simple kernel of independent memory operations.

For example, let us assume that for the Copy kernel, the address streams of X and Y are interleaved, i.e., `Load X(0)`, `Store Y(0)`, `Load X(1)`, `Store Y(1)`, etc. Then, if both Offset X and Offset Y are equal modulo 256, the pair of load and store will systematically hit the same bank for every iteration.

The term *computation size* is used to denote the total number of distinct elements of array X accessed during the whole execution of the loop. The impact of the computation size was not directly studied: all of the problems arising with very short computation size were not tackled.

In our study, we focus on steady state behavior, using a typical computation size of at least 1440 (corresponding to the computation of 1440 elements), which is large enough to reach peak performance while still allowing us to keep the operands in the L2 cache. If we want to lock the operands in L3, we choose a larger computation size while flushing the data from L2. That is, when we experiment with the L3 cache, we ensure that no data can be kept in L2, while keeping all the data in L3.

The measurements were performed on a stand-alone system (i.e., each benchmark code was the unique running user application), only one measurement being performed at a time. All timing measurements were performed using the *mov ar.itc* instruction to read the cycle counter of the processor itself.

Thanks to the perfmon toolkit, reading the various cache miss counters allows us to check, first, our assumptions that operands were effectively kept in the desired cache level, and second, that the penalties associated with DTLB remained negligible.

All of the performance numbers presented are normalized, i.e., the measurements correspond to the average number of clock cycles needed to compute one vector element of the BLAS 1 results. For instance, in the case of Daxpy, it is the average number of cycles needed to perform one instruction $Y(i) \leftarrow Y(i) + a \times X(i)$.

One of the major points of focus in this paper is the impact of array offsets on performance. Therefore, 2D plots (isosurface) will be displayed. The X-axis (resp. Y-axis) corresponds to the offset of the X array (resp. Y array). A *geographic* color code is used: dark colors correspond to the worst performance (highest number of clock cycles) while light colors correspond to the best performance (lowest number of clock cycles). These 2D plots are very useful to qualitatively understand the spatial nature of the dynamic phenomena.

After explaining our experimental configuration, the next section presents our experimental results.

Offset Y	Offset X	Accessed Banks	Performance in cycles
Off Y =0	Off X =0	0 0 0 0 (quadruple conflicts on bank 0)	1
Off Y =0	Off X =8	0 0 1 0 (triple conflicts on bank 0)	1
Off Y =0	Off X =64	8 0 8 0 (two double conflicts on bank 0 and 8)	0.9
Off Y =0	Off X =72	8 0 8 0 (double conflicts on bank 0)	0.9
Off Y =8	Off X =0	0 0 0 1 (triple conflicts on bank 0)	1
Off Y =8	Off X =8	0 0 1 1 (two double conflicts on bank 0 and 1)	0.9
Off Y =8	Off X =64	8 0 8 1 (double conflicts on bank 8)	0.9
Off Y =8	Off X =72	8 0 9 1 (no conflict)	0.5

Table 1: Bank Conflicts on the Itanium 2 Processor

4 Itanium 2 Cache System Behavior

To illustrate the specific problems occuring due to the Itanium 2 cache systems, we use two simple kernels (which are the simplest codes that put stress on the cache systems). They only consist of a burst of independent memory operations:

1. the first one, called *LxLy*, corresponds to a loop in which two arrays X and Y are regularly accessed through loads only: `Load X(0)`, `Load Y(0)`, `Load X(1)`, `Load Y(1)`, `Load X(2)`, `Load Y(2)`, etc.

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

As can be seen, these micro-benchmarks contain only independent memory operations and do not carry out any useful computation. They are free from any data dependence.

For the simple LxLy kernel, a naive code generation would be to alternate the access between array X and Y: `Load X(0)`, `Load Y(0)`, `Load X(1)`, `Load Y(1)`, `Load X(2)`, `Load Y(2)`, etc. Such a code (called *Interleaved LxLy*) results in a large number of bank conflicts depending on the Offset X and Y values. The experimental results (Figure 1(a)) perfectly describe this problem.

Theoretically, we expect a peak performance of 0.5 cycles to perform a pair of load operations, assuming that the data is in L2 and performing up to 4 loads per cycle. However, as can be seen in Figure 1(a), we get only a few points of peak performance depending on array offsets (light points). We can observe two complex bad behavior patterns:

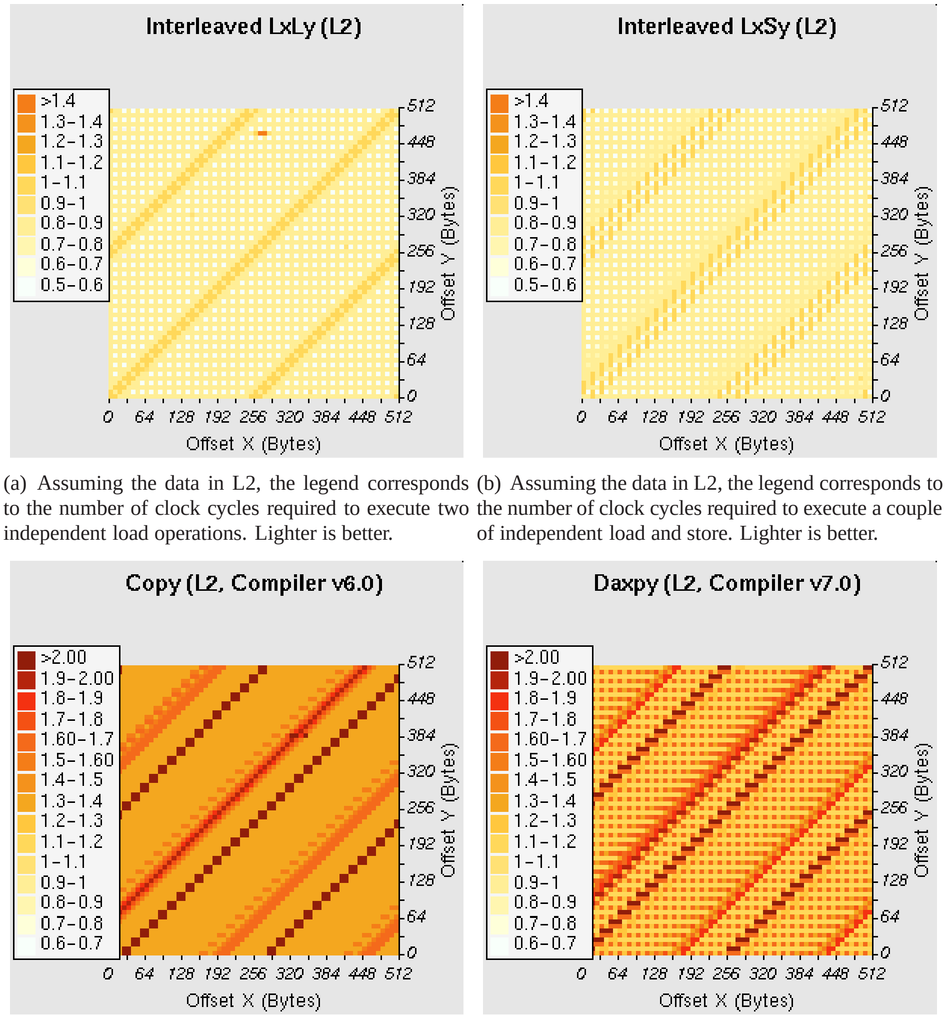
1. three diagonals separated by 256 bytes, where the performance degrades to 1 cycle instead of 0.5;

2. a grid pattern can be clearly observed (sometimes overwritten by the diagonals). On the "good" light points, the performance is maximal (0.5 cycles). In the other points, the performance degrades to 0.9 cycles.

Both phenomena can be attributed to bank conflicts caused by the interleaved LxLy micro-kernel. Table 1 summarizes the main bank conflicts. In this table, only the 4 first memory access are displayed, the other ones can be easily deduced from this initial pattern. The 256 bytes period of the diagonal stripes in Figure 1(a) is due to the fact that bank allocation is periodic with a 256 bytes period, i.e., two elements of the same array which are 256 bytes apart are allocated to the same bank.

Another micro-benchmark has been used, namely, LxSy. It is similar to LxLy, except that it performs stores on Y elements instead of loads. The performance of this code is displayed in Figure 1(b). Again, we get two complex bad behavior patterns: three diagonal stripes appears periodically each 16Kb and a grid pattern can be clearly observed.

The grid pattern is due to bank conflicts as was the case for LxLy. However, the three diagonals of bad performance are due to another hardware phenomena, in which the effects are aggregated with bank conflicts. They are caused by



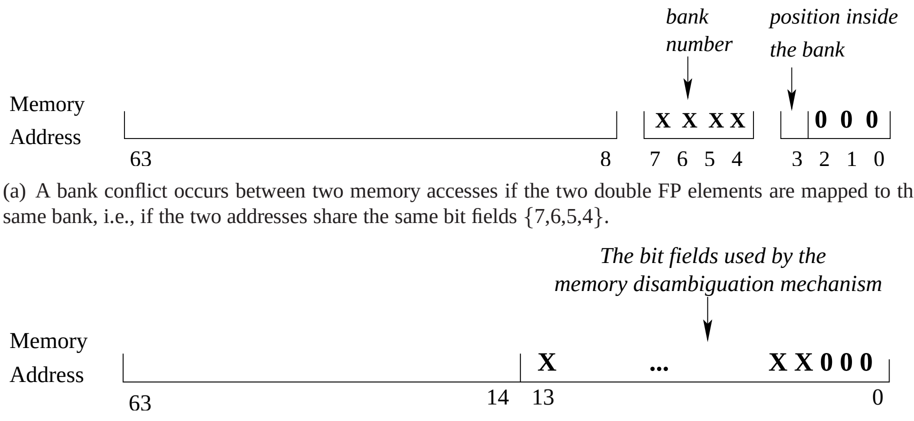
(a) Assuming the data in L2, the legend corresponds to the number of clock cycles required to execute two independent load operations. Lighter is better.

(b) Assuming the data in L2, the legend corresponds to the number of clock cycles required to execute a couple of independent load and store. Lighter is better.

(c) Assuming the data in L2, the legend corresponds to the number of clock cycles required to compute one vector element $Y(i) \leftarrow X(i)$. Lighter is better.

(d) Assuming the data in L2, the legend corresponds to the number of clock cycles required to compute one vector element $Y(i) \leftarrow Y(i) + a \times X(i)$. Lighter is better.

Figure 1: Itanium 2 L2 Cache Level Behavior



(a) A bank conflict occurs between two memory accesses if the two double FP elements are mapped to the same bank, i.e., if the two addresses share the same bit fields {7,6,5,4}.

(b) A Ld/St conflict occurs if the two accessed address share the same 14 low order bits.

Figure 2: Memory Operations Conflicts According to the Accessed Addresses

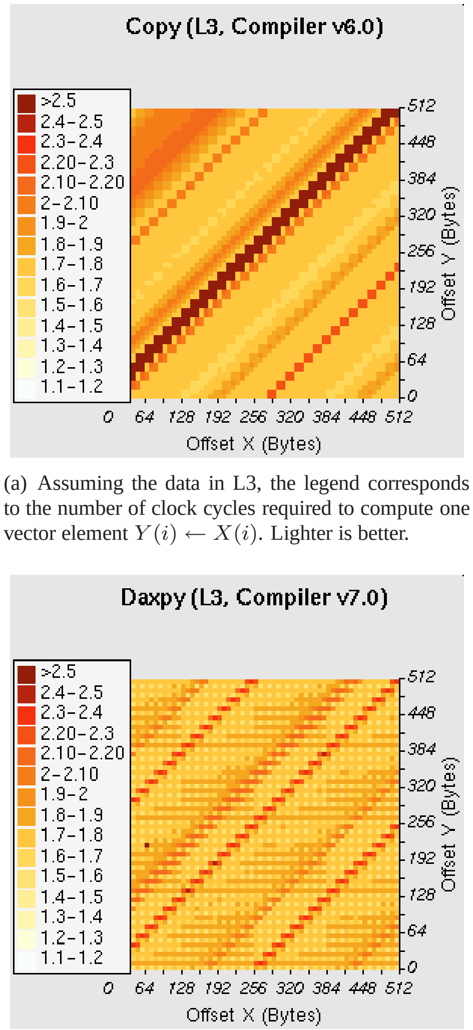
the non-precise memory disambiguation mechanism of Itanium 2. That is, if two memory operations (a store followed by a load) access two distinct memory addresses that share the same 14 low-order bits, the hardware detects a false memory dependence and triggers a sequential consistency mechanism. This is because in almost all modern ILP processors, memory disambiguation does not perform a full address bits comparison: this hardware simplification causes some independent memory operations to execute in serial, because the hardware does not detect their independence. It is the period of the diagonal stripes (16Kb) that reveals that only the 14 low-order address bits are checked in-flight by the hardware.

Both bank conflicts and non-precise memory disambiguation cause severe performance loss. Figure 2 summarizes the two situations where independent memory operations are serialized by the processor. Current compilers do not take into account this fact when they generate optimized code. Figures 1(c) and 1(d) show the performance of our BLAS 1 kernels. The compiler generates a well optimized code (software pipelining, prefetching, loop unrolling), but the peak performance is not reached even if the data is in L2. The situation in L3 is exactly the same, as shown in Figure 3(a) and 3(b). Note that the grid pattern does not appear in the copy case, because the compiler did not unroll the loop to statically schedule 4 loads in parallel.

5 Our Code Optimization Technique

Since we are dealing with vector loops, in which iterations are independent, most of our optimizations will be focused on specific ordering of load and store instructions. Let us think about a way 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 propose a new one; we only want to show that 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 also taking into account the address streams that they access. Since we do not know the exact array offsets at compile time,[3] we cannot determine precisely all memory locations 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 distance 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. We can also check if they are not mapped to the same bank This fact enables us to group memory operations accessing the same vector since we know their relative addresses. Such memory access grouping is similar

[3] However, we can fix the relative array offsets between arrays, for example, with array padding.



(a) Assuming the data in L3, the legend corresponds to the number of clock cycles required to compute one vector element $Y(i) \leftarrow X(i)$. Lighter is better.

(b) Assuming the data in L3, the legend corresponds to the number of clock cycles required to compute one vector element $Y(i) \leftarrow Y(i) + a \times X(i)$. Lighter is better.

Figure 3: Itanium 2 L3 Cache Level Behavior

to vectorization, except that only loads and stores are vectorized. Other operations, such as floating point operations, are not vectorized, and hence they are left free to be scheduled at the fine-grain level to enhance performance. A more formal explanation of our vectorization method is presented in the next section.

Our load instruction scheduling consists, first, of vectorizing the memory accesses, and then grouping together odd and even elements. For instance, the code generated for the LxLy kernel looks as follows.

```
[Ld X(0), Ld X(2), Ld X(4), Ld X(6)]
[Ld X(1), Ld X(3), Ld X(5), Ld X(7)]
[Ld Y(0), Ld Y(2), Ld Y(4), Ld Y(6)]
...
```

With such a load reordering strategy, the loads can be grouped in packets of 4 elements, such that within a packet, the addressed banks are distinct. The result of such a reordering is depicted in Figure 4(a) (compare it with Figure 1(a)). The performance is perfectly stable, reaching the optimum of 0.5 cycle per `Load X(i) Load Y(i)` pair. As can be seen, all of the bank conflicts are eliminated.

Now, for the LxSy kernel, the strategy used for the LxLy kernel can no longer be applied because a maximum of two stores can be issued per cycle and we cannot group 4 stores in the same packet. Therefore, our technique consists in grouping 2 loads with 2 stores, as follows:

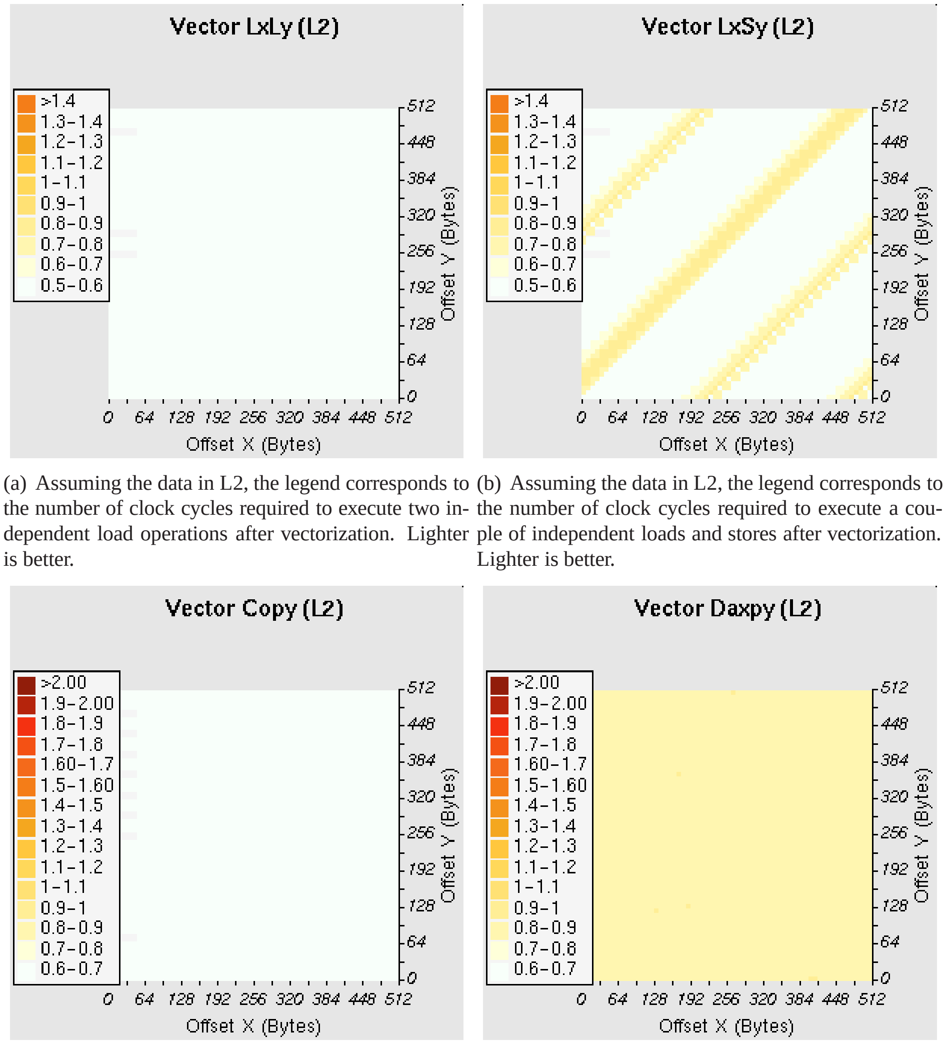
```
[Ld X(0), Ld X(2), St Y(0), St Y(2)]
[Ld X(1), Ld X(3), St Y(1), St Y(3)]
...
```

Note that in case of the existence of data dependences (BLAS 1 kernels), we have to first shift the stores by a constant factor (loop retiming), and then we group loads and stores as above. This scheduling technique completely eliminates all the potential bank conflicts and the grid pattern disappears, as can be seen in Figure 4(b) vs. Figure 1(b). However, we still have some diagonals of bad performance because of the non-precise memory disambiguation that results in ld/st conflicts. Fortunately, the performance loss in these diagonals is now lower than in the original version because we have reduced the conflict frequency between loads and stores. Furthermore, they occur for particular combinations of Offset X and Y values: such cases are depicted by narrow diagonals in the Offset X and Y plane.

In order to solve this last problem, two code variants differing by their software pipeline degrees are generated. That is, we shift the store elements, via loop retiming, by a constant factor. This results in shifting the diagonal stripes by the same factor. Hence, we have two codes that exhibit similar behavior, but the diagonals will be located in different (disjoint) areas in the Offset X and Y plane. Then, by inserting a switch that dynamically selects the best code variant depending on the Offset X and Y values, all performance troubles can be eliminated. We applied this technique to all our micro-benchmarks and the results were satisfactory. For instance, see Figures 4(c) and 4(d) (vs. Figures 1(c) and 1(d)). As can be seen, all our BLAS 1 kernels run now at peak performance for any array offset combination. The same peak performance can be sustained for the L3 cache (see Figures 5(a) and 5(b) vs Figures 3(a) and 3(b)).

If we do not apply dynamic code selection and we decide to keep only one version of the code, we still have an important performance gain. Figure 6 shows the worst case performance gain, counted as the worst performance of the whole original code against the worst performance of the whole optimized code. The peak (best) performance of the new codes are exactly the same as for the original ones (only the worst-case performance changes). Note that in Figure 6, we do not report the gain obtained by the shifting (dynamic code selection) strategy, but only the performance of the optimized codes even if they contain bad diagonal stripes.

The results presented above show that vectorizing ld/st operations is a robust code optimization technique. However, we must be aware that register pressure may increase, limiting our opportunity to produce long vector accesses. A first naive approach would be to abstract the register file and the instruction set, i.e., we would consider that the 128 FP registers are organized as k vector registers of size m: our ld/st vectorization would use these k vector registers. Next, we could perform a standard instruction scheduling and register allocation pass using these vector instructions assuming k available registers. Finally, in a post pass, every instruction is expanded/replaced by a corresponding sequence of m scalar instructions. Such a strategy has two problems: first, it goes beyond what is strictly required (only



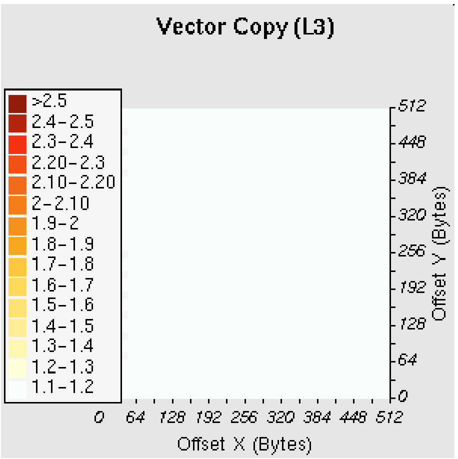
(a) Assuming the data in L2, the legend corresponds to the number of clock cycles required to execute two independent load operations after vectorization. Lighter is better.

(b) Assuming the data in L2, the legend corresponds to the number of clock cycles required to execute a couple of independent loads and stores after vectorization. Lighter is better.

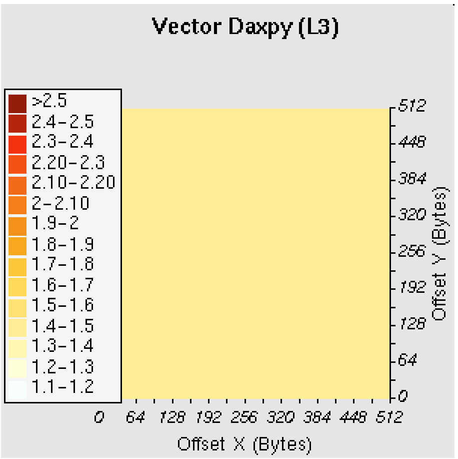
(c) Assuming the data in L2, the legend corresponds to the number of clock cycles required to compute one vector element $Y(i) \leftarrow X(i)$ after vectorization. Lighter is better.

(d) Assuming the data in L2, the legend corresponds to the number of clock cycles required to compute one vector element $Y(i) \leftarrow Y(i) + a \times X(i)$ after vectorization. Lighter is better.

Figure 4: Impact of ld/st Vectorization on L2 Behavior



(a) Assuming the data in L3, the legend corresponds to the number of clock cycles required to compute one vector element $Y(i) \leftarrow X(i)$ after vectorization. Lighter is better.



(b) Assuming the data in L2, the legend corresponds to the number of clock cycles required to compute one vector element $Y(i) \leftarrow Y(i) + a \times X(i)$ after vectorization. Lighter is better.

Figure 5: Impact of ld/st Vectorization on Itanium 2 L3 Behavior

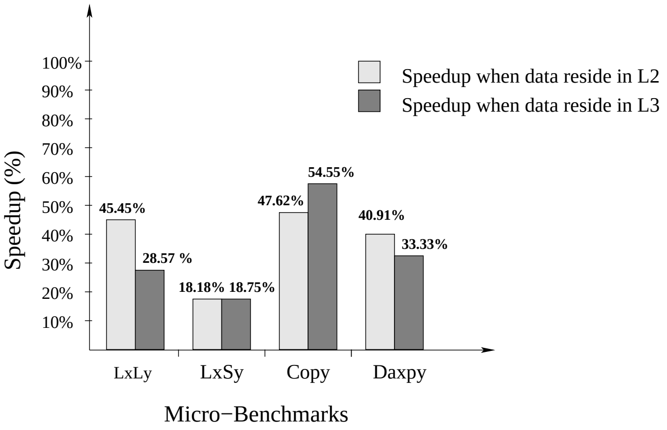


Figure 6: Worst-Case Performance Gain on the Itanium 2 Processor (Higher is Better)

reordering of loads and stores is required, not of other FP operations). Second, if R "vector" registers are consumed during the allocation pass, this will turn into $m \times R$ scalar registers consumed. This is a loss of registers since we should not restrict the FP operations to be vectorized.

The next section shows how we do an effective register allocation in a better way.

6 The impact on Register Pressure

In this section, we present our method that applies a cyclic register allocation in a loop that consists of arithmetic expressions. We exploit fine grain parallelism within the operations by using the software pipelining technique. Furthermore, we vectorize memory access operations in order to avoid bank and ld/st queue conflicts.

6.1 Vectorization Methods

To highlight the impact on register pressure caused by our ld/st vectorization, we use three micro benchmarks:

1. copy: $Y(i) \leftarrow X(i)$;
2. daxpy: $Y(i) \leftarrow Y(i) + a \times X(i)$;
3. vsumspy: $Y(i) \leftarrow X(i) \times Z(i) + T(i)$

Our experiments are devoted to study the impact of ld/st vectorization on the register requirement versus the initiation interval II of software pipelining. We study four variants (vectorization versions) for each micro benchmark. Floating point operations are not vectorized, thus they are free to be scheduled at the fine grain level. These vectorization variants are as follows, where we give examples for the copy benchmark.

Variant 1 The loop is unrolled 4 times and the loads (resp. stores) are packed, i.e., each vector operation consists of 4 loads (resp. 4 stores).

```
[Ld X(i), Ld X(i+1), Ld X(i+2), Ld X(i+3)]
[St Y(i), St Y(i+1), St Y(i+2), St Y(i+3)]
```

Variant 2 The loop is unrolled 8 times and 8 loads (resp. 8 stores) are packed, i.e., each vector operation consists of 8 loads (resp. 8 stores).

```
[Ld X(i), Ld X(i+1), Ld X(i+2),..., Ld X(i+7)]
[St Y(i), St Y(i+1), St Y(i+2),..., St Y(i+7)]
```

Variant 3 The loop is unrolled 8 times and every 4 loads (resp. stores) are packed, i.e., each vector operation consists of 4 loads (resp. 4 stores). Each block of 4 loads (stores) accesses only even or odd elements.

```
[Ld X(i),   Ld X(i+2), Ld X(i+4), Ld X(i+6)]
[Ld X(i+1), Ld X(i+3), Ld X(i+5), Ld X(i+7)]
[St Y(i),   St Y(i+2), St Y(i+4), St Y(i+6)]
[St Y(i+1), St Y(i+3), St Y(i+5), St Y(i+7)]
```

Variant 4 The loop is unrolled 4 times. We group 2 loads with 2 stores, i.e., each vector operation consists of 2 loads and 2 stores that access even or odd elements. Since there are more load operations inside the unrolled loop than stores, the remaining loads are grouped inside blocks of 4 loads that access even or odd elements. Note that the distance d of the store operations can be either fixed or kept as a parameter.

```
[Ld X(i),   Ld X(i+2), St Y(i-d),   St Y(i-d+2)]
[Ld X(i+1), Ld X(i+3), St Y(i-d+1), St Y(i-d+3)]
```

In order to group loads and stores in a vectorized way, we directly work on the data dependence graph (DDG) of the loop, as explained in the following section.

6.2 Ld/St Vectorization at the DDG Level

Let $G = (V, E, \delta, \lambda)$ a loop DDG that consists of:

- V is the set of the statements in the loop body. The instance of a statement u (an operation) in iteration i is denoted by $u(i)$.
- E is the set of precedence constraints (flow dependences or other serial constraints). Any edge e has the form $e = (u, v)$. The function $\delta(e)$ is the latency of the edge e in terms of processor clock cycles and $\lambda(e)$ is the distance of edge e in terms of number of iterations.

If we want to vectorize k memory operations $u_1, \dots, u_k$, we only have to add a cycle C (with a null latency and distance) that joins these operations. Such cycle reflects the fact that the connected operations are constrained to be executed in parallel, which is equivalent to forming a single vector instruction.

Note that, sometimes, we have to unroll the loop k times to vectorize the k operations considered. Figure 7 shows an example of a vectorization that connects 4 loads and 4 stores to produce a vectorized copy (variant 1). Bold circles denote the operations that write into FP registers and bold edges denote flow dependences through FP registers. The added edges (not in bold) do not refer to any data dependence, but are inserted to restrict the scheduler so as to vectorize the memory operations. Figure 8 shows a vectorized copy with variant 4: as can be seen, each packets consists of two loads connected with two stores. The stores are shifted with a factor d. This factor can be either fixed, or can be left free to be computed by our register allocator [26].

Now we can apply classical software pipelining and register allocation on this modified DDG. We can use either a post-pass register allocation after register-sensitive software pipelining [9], or a pre-pass register allocation that is sensitive to software pipelining [26].

Our technique that handles ld/st vectorization at the DDG level is better than classical vector register allocation. This is because it applies vectorization to only a subset of operations (memory access only). The non-vectorized operations are free to be scheduled at the fine-grain level, so as to minimize the register requirement and to minimize the initiation interval II of software pipelining. Classical vector register allocation techniques, such as described in [2, 4], would require to multiply the number of registers needed for the original loop by the unrolling factor. This is because they assume that all statements are vectorized.

The next section summarizes our experimental results.

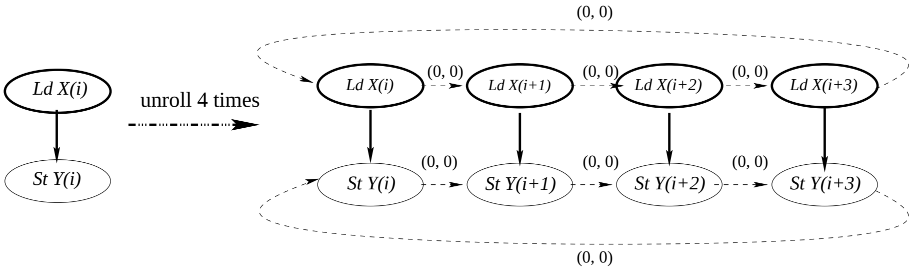


Figure 7: Vectorized copy (Variant 1)

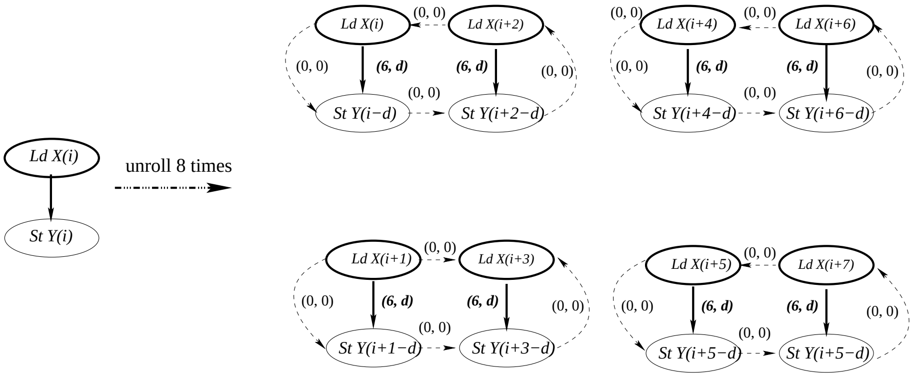


Figure 8: Vectorized Copy (Variant 4), with Shifted Stores

6.3 Experiments with Register Allocation

In our framework, it turns out that the approach of [26] is more convenient for us than other existing techniques for two main reasons:

1. The authors apply register allocation before software pipelining, and hence avoidable spill code is not introduced. This is a major issue, since our target codes are memory-bound, and we should not introduce additional memory operations;
2. The experiments conducted by the authors show that their technique is close to the optimal. Other existing heuristics do not provide comparisons with an optimal register allocation.

Figures 9 to 12 plot our experiments for each vectorization variant. We plot the results of the non vectorized loops (the original unrolled loops) in order to highlight the difference in terms of register requirements. Our experiments show two main trends, depending on the vectorization technique.

Trend 1 is highlighted by the experiments of variants 1, 2 and 3. In all these codes, the vectorization is applied to loads only, or to stores only. As can be seen, and as we expected from theoretical considerations, we have two main conclusions:

1. such vectorization techniques do not alter MII [4];
2. the register requirement is slightly increased for the lower values of II. However, the difference with the non vectorized codes is not greater than the vectorization degree, i.e., if we vectorize k loads, then the difference in terms of register requirement does not exceed k.

Trend 2 is highlighted by the experiments of variant 4. In all these codes, some loads and stores are packed in the same vector instructions. We have two main conclusions:

1. Such vectorization techniques have a high impact on MII. In all experiments of variant 4, the MII increases (see the starting point of each curve). Indeed, the vectorization introduces new critical cycles into the loops, since we are connecting loads with stores into the same cycles. The value of the new MII depends on the distance d of the stores: the higher d, the lower MII.
2. The register requirement is substantially increased. This is because we vectorize loads together with stores, and hence the lifetimes of variables are forced to be longer. The difference (in terms of register need) with the non vectorized codes depends on the distance d of the stores: the higher d, the higher the register requirement.

7 Related Work and Discussion

In this paper, we used some micro-benchmarks to precisely highlight performance bugs in real ld/st queues and to explore the effectiveness of cache banks. Micro-benchmarking (i.e., reduced and simple test codes) has been extensively used to characterize and analyze computer architectures. McCalpin [17] developed the Stream Benchmark whose main goal is to precisely calculate main memory performance. Unfortunately, the various cache levels are not covered by this benchmark. McVoy *et al* [18] introduced lmbench whose main goal is to detect performance bottlenecks. Saavedra *et al* [23] developed a suite of micro kernels to extract basic performance numbers and used these numbers to predict the performance of real applications. Iyer *et al* [14] extend the work of Abandah *et al* [1] to study the impact of several architectural choices used in the HP V class and SGI O2000.

The aim of our micro-benchmarks is to study the importance of the memory address stream. This aspect is not covered by existing micro-benchmarks. Furthermore, our approach is slightly different, in the sense that our final

[4] MII denotes the standard minimal initiation interval, that is, the starting point of each curve in our figures.

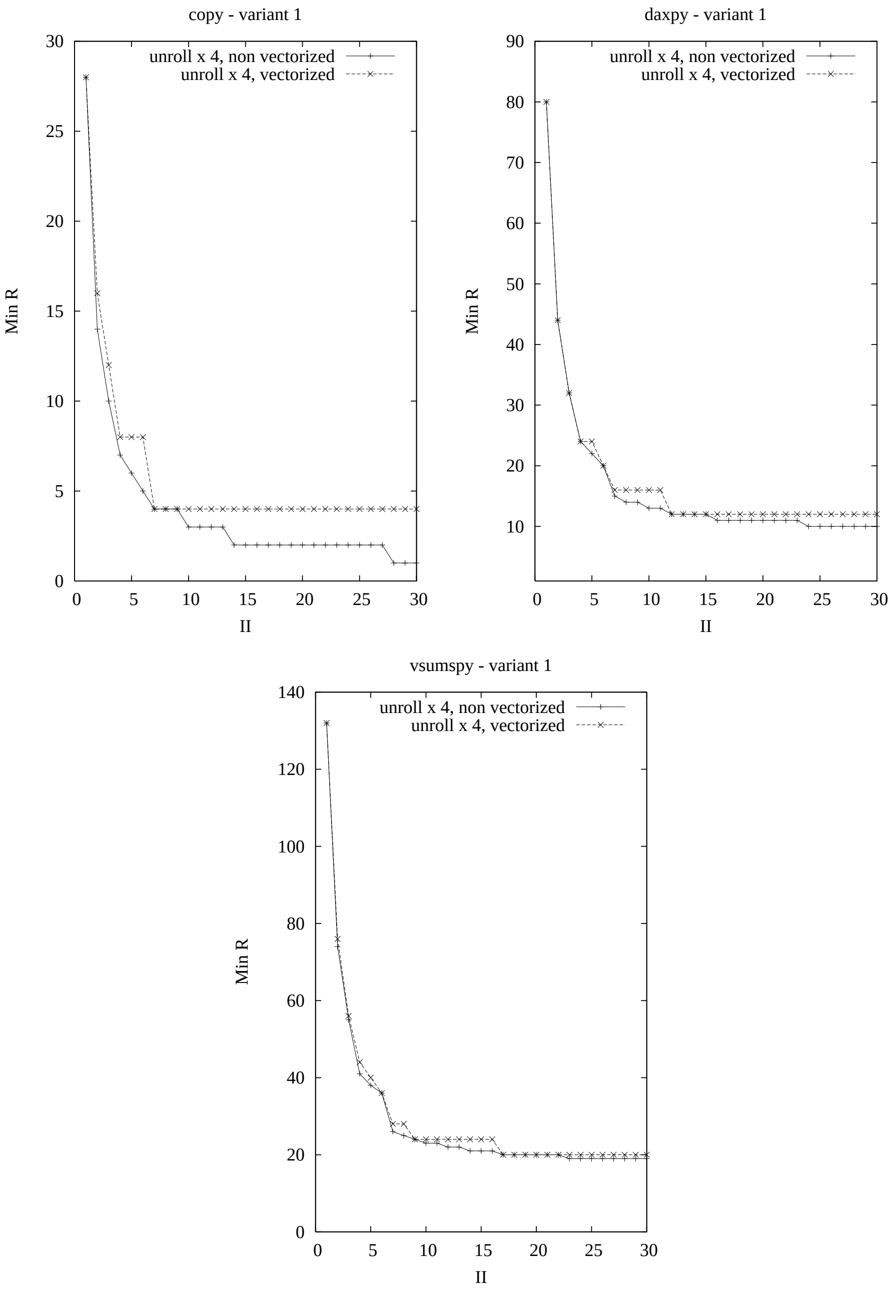


Figure 9: Register Requirements for Variant 1

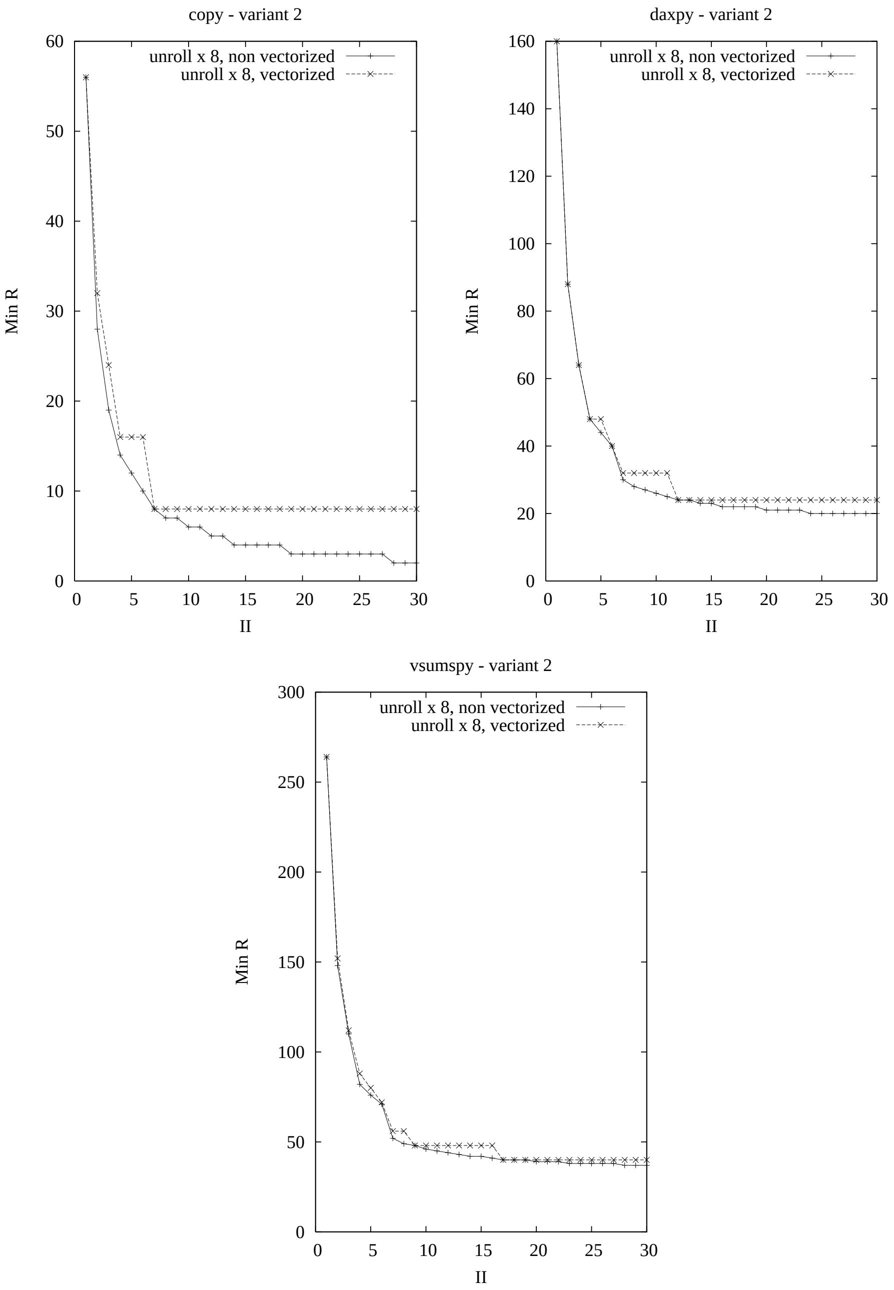


Figure 10: Register Requirements for Variant 2

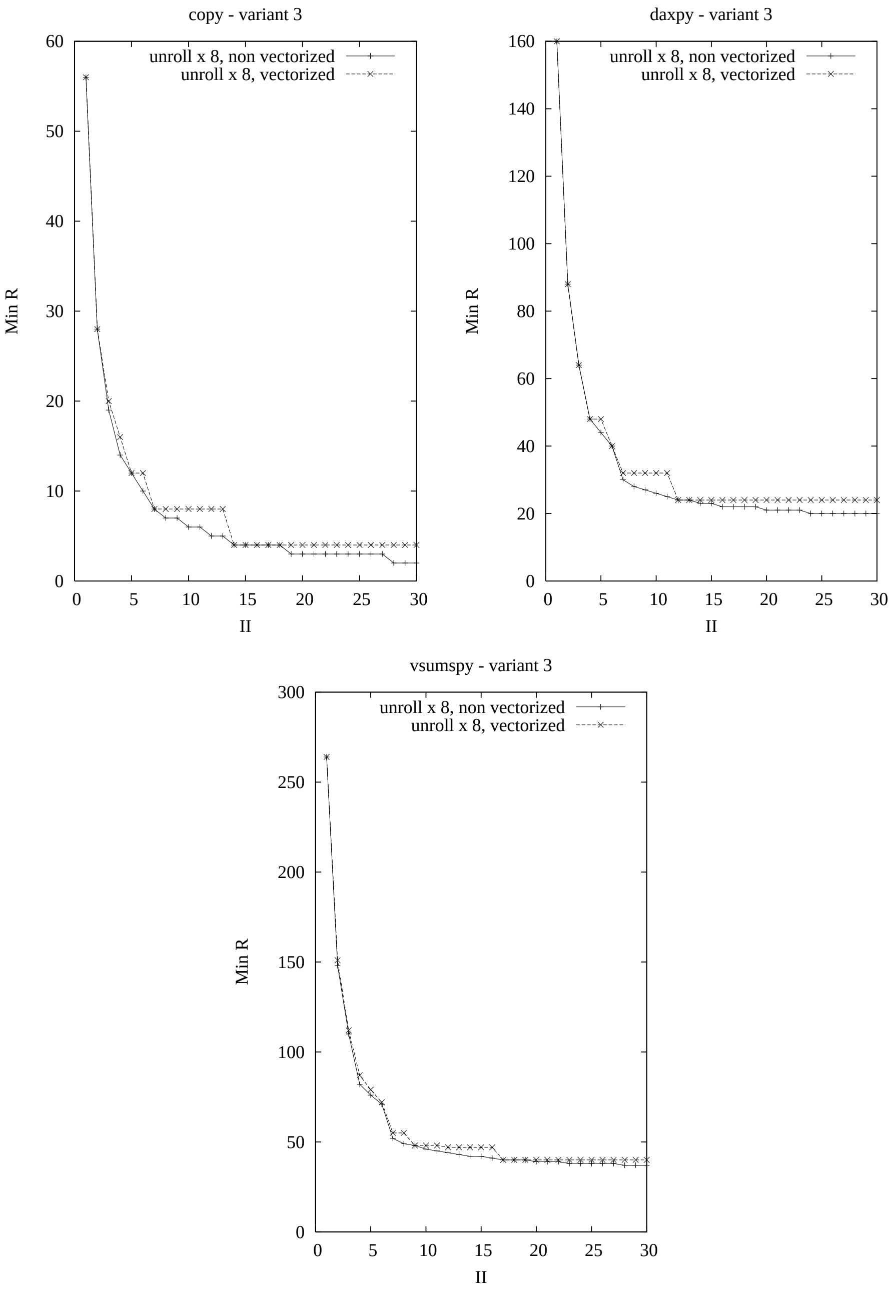


Figure 11: Register Requirements for Variant 3

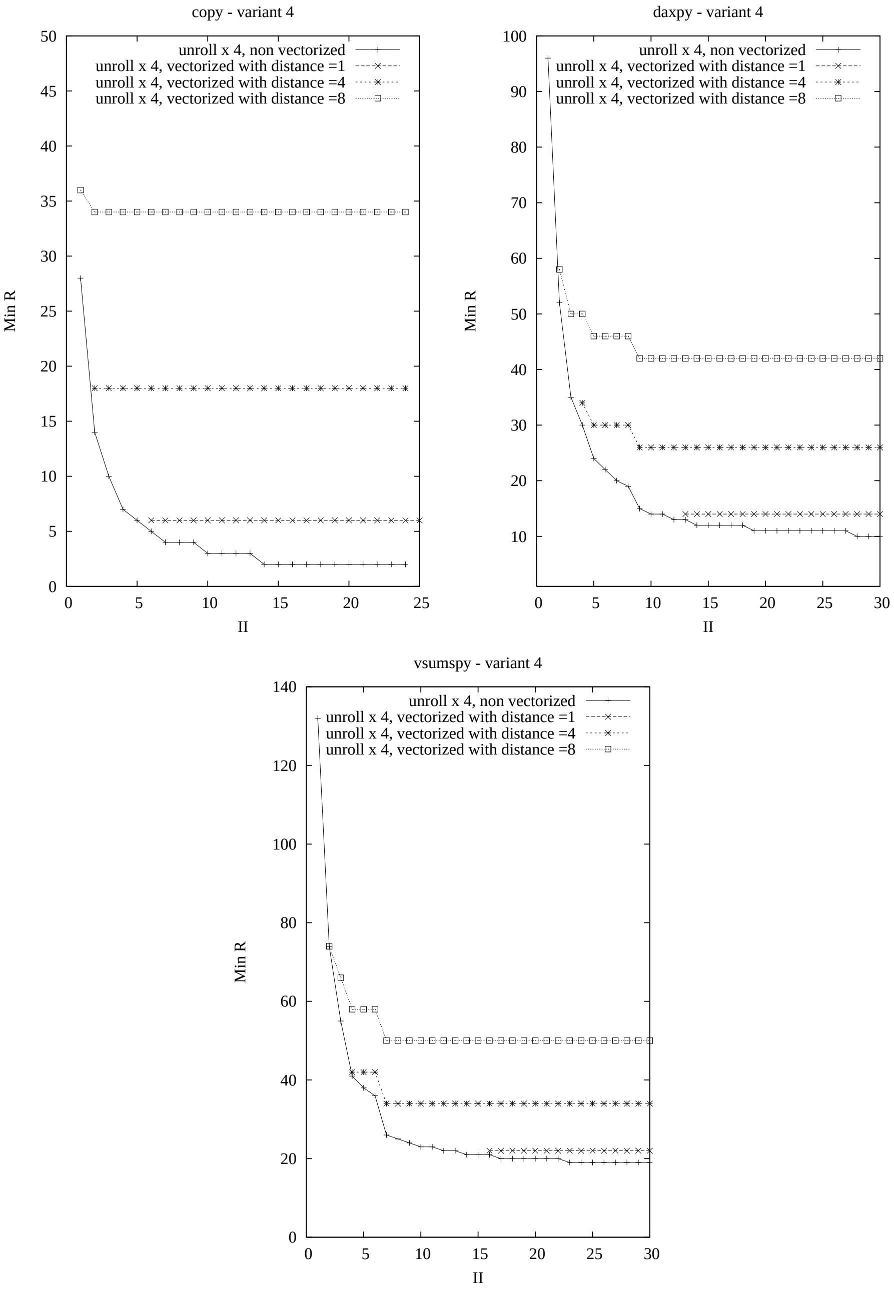


Figure 12: Register Requirements for Variant 4

goal is to improve code generation within a compiler and not only to test an architecture. We demonstrate that cache systems are sensitive not only to the classical miss ratio, but also to the ordering of the accessed addresses. Pai *et al* [27] already noticed a similar phenomenon when they introduced the read miss clustering: by changing the temporal distribution of cache misses (this was achieved by modifying the address streams), they improved code performance on a Convex Exemplar architecture.

On the other hand, improving ld/st queues and memory disambiguation mechanisms is an issue of active research. Chrysos and Emer [7] 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 peak performance in the context of large instruction windows. Park *et al* in [22] proposed an improved design of ld/st queues that scales better, that is, they have improved the design complexity of memory disambiguation. Another similar hardware improvement has been proposed by Sethumadhavan *et al* in [22]. A speculative technique for memory dependence prediction has been proposed by Yoaz *et al* in [28]: the hardware try 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 [20]. The author presented a hardware mechanism that assigns the loads and stores to an appropriate speculative level for memory dependence prediction.

All sophisticated techniques discussed above are hardware solutions. In the domain of scientific computation, 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 [7, 22, 22, 28, 20]. Coupling our cheaper software optimization technique with the actual non precise memory disambiguation mechanisms is less expensive than pure hardware methods.

Note that many compilation techniques are dealing with instruction scheduling and software pipelining to address the memory aliasing problem [11]. However, these schemes try to optimize loops with *dynamic* memory dependencies, that is, memory aliases that are not solvable at compile time. Thus, they propose to use some EPIC/IA64 features, such as predication, to generate faster codes. In our case, we deal with perfect regular codes, where no memory aliasing problem exists, since we are able to guarantee the independence of all memory operations. We seek a convenient way to take benefit from this high potential ILP by considering weak dynamic memory disambiguation mechanisms; this is the opposite approach of [11], since they rely on dynamic memory disambiguation to compensate weak static data dependence analysis and ILP extraction.

Other code optimization methods may help to avoid the performance bugs shown in our experiments. For instance, array padding [21] can be used to modify the relative array offsets, and it may be possible to guide it so that neither ld/st nor bank conflicts can arise. However, such a technique is more convenient for cache miss reduction than for eliminating ld/st conflicts: the problem discussed in this paper does not arise because data is not in the cache, or due to any other data locality problem; our actual experiments are performed on fully cached data. Other customizing data layout techniques that improve memory parallelism exist [29, 25] and need whole program analysis for data layout computation. However, they try to eliminate bank conflicts and not ld/st queue collisions. Furthermore, we aim to optimize scientific computation libraries alone, where arrays can be declared outside the functions, or can be dynamically allocated. In this case, customizing data layout is useless.

An alternative approach would be to copy multiple arrays into a single array and perform the computation on this single array, giving full control over the memory layout, but with the expense of extra overhead. Since we deal with memory-bound loops, each additional memory operation by array copying increases the bottleneck.

Thus, we prefer to use ld/st vectorization because, first, it moderates the negative effects of current memory address disambiguation strategies which use a subset of the address bits. Second, it is also convenient for bank conflicts. Third and last, it is applicable to all BLAS 1 routines, independently from the data layout of the caller program. Vectorization is a complex technology, and many studies have been performed on this topic [5, 16]. In our framework, the problem is simplified since we tackle fully parallel innermost loops. We only seek a convenient vectorization degree. Ideally, the higher this degree, the higher the performance, but the higher the register pressure too. Thus, we are constrained by the number of available registers. We showed in this paper how we can modify the register allocation step by combining ld/st vectorization at the data dependence graph (DDG) level without hurting ILP extraction by using our previous theoretical framework [26].

8 Conclusion and Future Work

Memory-bound programs rely on advanced compilation techniques that try to keep data in the caches, 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 this is not sufficient for sustaining peak performance. First, the memory disambiguation mechanisms in the Itanium 2 processor do not perform comparisons on full address bits. If two memory operations access two distinct memory locations but share the same 14 lower-order address bits, 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 Itanium 2 processors may prevent us to execute independent loads in parallel. If two elements are mapped to the same bank, independent loads are restricted to be executed sequentially, even if enough functional units are available. This fact is a well known source of troubles, but current compilers still do not take it into account (even with highly optimized, hand tuned libraries provided by the vendors), and the generated codes can be 2 times slower on Itanium 2.

We demonstrated that a memory instruction reordering based on the classical (but robust) vectorization can get rid of these performance bugs. The cost of such technique in terms of register need was analyzed. We have seen that grouping loads together, and stores together, does not have a major impact on register requirement. However, when mixing some stores with loads into the same instruction group, the register pressure may substantially increase depending on the distances of the stores. Thus, if not enough registers exist, we cannot generate a vectorized code. However, an extensive set of experiments on random trees showed us that the 128 available registers of the Itanium 2 processor are sufficient to allocate a great majority of the trees of height 4 (without performance loss, i.e., without increasing II).

The current vectorization strategy has been successfully implemented in our internal compiler devoted to optimize scientific vector loops. This work will be extended into two major directions. First, more complex kernels involving a larger number of arrays and more complex arithmetic operations will be studied. Although our preliminary results on register allocation are promising, they need to be tested and analyzed in a more general framework. Second, main memory access deserves a similar study. Already, some preliminary experiences have confirmed us with the good performance capabilities of the vectorization strategy.

9 Acknowledgement

This work has been supported by CEA-DAM, Bull and the French ministry of research.

References

[1] G. A. Abandah and E. S. Davidson. Characterizing Distributed Shared Memory Performance: A Case Study of the Convex SPP1000. *IEEE Transactions on Parallel and Distributed Systems*, 9(2):206–216, February 1998.

[2] Randy Allen and Ken Kennedy. Vector Register Allocation. *IEEE Transactions on Computers*, C-41(10):1290–1317, October 1992.

[3] D.H Bailey. Unfavorable Strides in Cache Memory Systems. In *Scientific Programming*, volume 4, pages 53–58, 1995.

[4] David Bernstein, Haran Boral, and Ron Y. Pinter. Optimal Chaining in Expression Trees. *IEEE Transactions on Computers*, 37(11):1366–1374, November 1988.

[5] D. A. Calahan, J. J. Dongarra, and D. Levine. Vectorizing compilers : A test suite and results. In *Supercomputer '88*, pages 98–105. IEEE Press, 1988.

[6] Tien-Fu Chen and Jean-Loup Baer. Effective Hardware Based Data Prefetching for High-Performance Processors. *IEEE Transactions on Computers*, 44(5):609–623, 1995.

[7] 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.

[8] David E. Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers, San Francisco, CA, 1999.

[9] Dominique de Werra, Christine Eisenbeis, Sylvain Lelait, and Bruno Marmol. On a Graph-Theoretical Model for Cyclic Register Allocation. *Discrete Applied Mathematics*, 93(2-3):191–203, July 1999.

[10] Keith I. Farkas, Norman P. Jouppi, and Paul Chow. How Useful Are Non-blocking Loads, Stream Buffers and Speculative Execution in Multiple Issue Processors? In *Proceedings of the First International Symposium on High-Performance Computer Architecture*, pages 78–89, Raleigh, North Carolina, January 1995. IEEE Computer Society TCCA.

[11] Benjamin Goldberg, Emily Crutcher, Chad Huneycutt, and Krishna Palem. Software Bubbles: Using Predication to Compensate for Aliasing in Software Pipelines. In *2002 International Conference on Parallel Architectures and Compilation Techniques (PACT'02)*, pages 211–221, Virginia, September 2002. IEEE.

[12] Jerome C. Huck, Dale Morris, Jonathan Ross, Allan D. Knies, Hans Mulder, and Rumi Zahir. Introducing the IA64 Architecture. In *IEEE Micro*, September 2000.

[13] Intel. Intel Itanium2 Processor Reference Manual for Software Develoment Optimization. Technical Report 251110-001, Intel, June 2002.

[14] Ravi Iyer, Nancy M. Amato, Lawrence Rauchwerger, and Laxmi Bhuyan. Comparing the Memory System Performance of the HP V-Class and SGI Origin 2000 Multiprocessors using Microbenchmarks and Scientific Applications. In *Conference Proceedings of the International Conference on Supercomputing*, pages 339–347, Rhodes, Greece, June 1999. ACM SIGARCH.

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

[16] David Levine, David Callaahan, and Jack Dongarra. A Comparative Study of Automatic Vectorizing Compilers. *Parallel Computing*, 17:1223–1244, 93.

[17] John D. McCalpin. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Technical Committee on Computer Architecture newsletter*, December 1995.

[18] Larry McVoy and Carl Staelin. lmbench: Portable Tools for Performance Analysis. In *Proceedings of the USENIX Annual Technical Conference*, pages 279–294, San Diego, CA, USA, January 1996. Usenix Association.

[19] W. Oed and O. Lange. On the Effective Bandwidth of Interleaved Memories in Vector Processor Systems. *IEEE Transactions on Computers*, C-34:949–957, 1985.

[20] Soner Onder. Cost Effective Memory Dependence Prediction using Speculation Levels and Color Sets. In *2002 International Conference on Parallel Architectures and Compilation Techniques (PACT'02)*, page 232, Virginia, September 2002. IEEE.

[21] Preeti Ranjan Panda, Hiroshi Nakamura, Nikil D. Dutt, and Alexandru Nicolau. Augmenting Loop Tiling with Data Alignment for Improved Cache Performance. *IEEE Transactions on Computers*, 48(2):142–149, 1999.

[22] Il Park, Chong liang Ooi, and T. N. Vijaykumar. Reducing Design Complexity of the Load/Store Queue. In *Proceedings of the 36th International Symposium on Microarchitecture (MICRO-36)*, pages 411–422, San Diego, December 2003. IEEE.

[23] R. H. Saavedra and A. J. Smith. Measuring Cache and TLB Performance and Their Effect on Benchmark Run Times. *IEEE Transactions on Computers*, C-44(10):1223–1235, October 1995.

[24] Harsh Sharangpani and Ken Arora. Itanium Processor Microarchitecture. *IEEE Micro*, 20(5):24–43, September/ October 2000.

[25] Byoungro So, Mary Hall, and Heidi Ziegler. Custom Data Layout for Memory Parallelism. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, pages 291–302, Palo Also, CA, March 2004. IEEE.

[26] Sid Ahmed Ali Touati and Christine Eisenbeis. Early Periodic Register Allocation on ILP Processors. *Parallel Processing Letters*, 14(2):287–313, 2004.

[27] Vijay S. Pai and Sarita Adve. Code Transformations to Improve Memory Parallelism. *The Journal of Instruction-Level Parallelism*, 2, May 2000. http://www.jilp.org/vol2.

[28] A. Yoaz, M. Erez, R. Ronen, and S. Jourdan. Speculation Techniques for Improving Load Related Instruction Scheduling. *26th Annual International Symposium on Computer Architecture (26th ISCA'99), Computer Architecture News*, 27(2):42–53, May 1999.

[29] Xiaotong Zhuang, Santosh Pande, and John S. Greenland Jr. A Framework for Parallelizing Load/Stores on Embedded Processors. In *2002 International Conference on Parallel Architectures and Compilation Techniques (PACT'02)*, page 68, Virginia, September 2002. IEEE.