Efficient Code Optimization Technique for Itanium2 Cache System and Scientific Computing

William Jalby, Christophe Lemuet, Sid Touati

Efficient Code Optimization Technique for Itanium2 Cache System and Scientific Computing

William JALBY, Christophe LEMUET, Sid-Ahmed-Ali TOUATI
{jalby, lemuet, touati }@prism.uvsq.fr
PRiSM Laboratory, University of Versailles, France

Abstract

To keep up with a large degree of ILP, Itanium2 L2 cache system uses a complex organization scheme: load/store queues, banking and interleaving. In this paper, we study the impact of this cache system on memory instruction scheduling. We demonstrate that for scientific codes, "memory access vectorization" allows to generate very efficient code (up to the maximum of 4 loads per cycle). The impact of such "vectorization" on register pressure is analyzed: various register allocation schemes are proposed and evaluated.

1 Introduction

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

The good side of the story is that these mechanisms allow to offer excellent peak performance numbers. The bad side of the story is that these mechanisms requires from codes specific characteristics (in particular "good/appropriate memory reference patterns") to reach peak performance.

Previously, in the "old days" of vector machines, a similar trend was observed. To reach decent memory bandwidth, Cray XMP had a very complex memory subsystem organized in banks, sections, subsections, etc... Already a careful instruction scheduling was necessary to avois spurious bank conflicts [6].

In this paper, the Itanium2 architecture [5, 4, 7] was selected as a target, first, because it offers a large degree of ILP (manageable directly by the compiler) and, second, because its cache subsystem is complex (three levels, high degree of parallelism, sophisticated prefetch capabilities...).

Due to the already complex nature of the problem, our study is currently restricted to the L2, L3 cache subsystem (excluding memory) and to simple vector codes (BLAS 1 type: Copy, Daxpy). Although simple, such BLAS1 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 (easy to analyze statically) and the Itanium2 architecture (very well designed for exploiting static information).

Even with this rather limited scope in terms of application codes, our study reveals that performance behavior is rather complex and hard to analyze. In particular, the banking/interleaving structure of the L2 cache is shown to have a major interaction with address stream, potentially inducing large performance loss.

We demonstrate that vectorizing memory access in a clever manner allows to get rid of most of the L2 cache bank conflicts. Unfortunately, such techniques may increase considerably register pressure. Therefore, several techniques to reduce register pressure are presented and evaluated.

Section 2 describes our experimental setup: hardware platform as well as software platform (compiler and OS). In Section 3, our target codes and expeimental methodology are presented. In Section 4, our scheduling strategy for memory access is detailed. In Section 5, experimental results on BLAS1 kernels are presented allowing to validate our "memory access vectorization" technqiues. Section 6 presents our register allocation technique with their experiments in Section 7. Finally, few concluding remarks and directions for future work are given.

2 Experimental Setup

The machine used in our experiments is an uniprocessor Itanium2 based system equipped with 900Mhz pro-

cessor and 3GB memory. The ”general” processor architecture (an interesting combination of VLIW and advanced speculative mechanisms) is very well described in the literature [4, 7, 2, 5]. The Itanium2 offers a wide degree of parallelism:

- Six general purpose ALU, two integer units and one shift unit;
- The Data cache unit contains four memory ports allowing to service either four loads requests or two load and two store requests in the same cycle;
- Two floating point Multiply Add units allowing to execute up two floating point multiply add operations per cycle;
- Six multimedia functional units;
- Three Branch units, ...

All of the computational units are fully pipelined, so each functional unit 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.

On our test machine, caches are organized in three levels: L1 (not available for storing floating point data), L2 (unified, 256 KB, 8 way associative, Write Back Allocate, 128 Bytes cache line), 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 16 Bytes interleaving of 16Bytes: address 0 and 8 are located in bank 0, address 16 and 24 in bank 1, etc .... Finally, the cache interface (for both L1 and L2) is equipped with a Load/Store queue allowing hits to bypass miss (provided thatthe adress are correctly dismabiguated).

In addition to standard Load and Store instructions on floating point operands (single and double precision), the Itanium instruction set offers Load Floating Pair instruction capable of loading 16 Bytes at once, provided that the corresponding address are lined up on a 16 Bytes boundary.

Our test machine was running Linux IA-64 Red Hat 7.1 based on the 2.4.18 smp kernel. The page size used by the system was 16Kbytes and we used the following compiler: Intel C++ Compiler Version 7.0 Beta, Build 20020703. Various compiler options have been tested, however for our simple BLAS1 kernels, it was found that the combination of -O3 and -restrict was very powerful, fully using most of the advanced features of Itanium2 architecture: software pipelining, prefetch instructions, predication, rotating registers In getting top performance, the ”-restrict” option was essential because it allowed the compiler to assume that distinct arrays were pointing to disjoint memory regions, therefore, allowing a full reordering of loads and stores. Fortran compiler was also tested, but again for our simple loops, the code generated was almost identical to the one obtained with C language.

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

Besides the compiler and OS, the perfmon toolkit, allowing direct access to various performance counters has been used.

3 Target Codes and Measurement

The BLAS1 kernels tested are simple vector loops. Due to lack of space, detailed results will only be given for two of them: Copy (Y(I) = X(I)) and Daxpy (Y(I) = Y(I) + a * X(I)).

All of the arrays used are double precision real (8 bytes elements). 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, Y) is studied in depth. To achieve this goal, the parameters Offset X (resp. Offset Y) are introduced, according to the following relations:

- Virtual address for X[i]: 512K + Offset X + 8*i
- Virtual address for Y[i]: 512K + DIST1 + Offset Y + 8*i

DIST1 is an additional parameter mainly used to avoid array overlap, i.e. making sure that array X and Y are located in disjoint portions of memory space . In most of our experiments DIST1 remains constant, equal to 32 KB. Changing the values of Offset X and Offset Y, will allow to change the L2 bank access pattern: 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 Y(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 the array X accessed during the whole loop execution. Impact of the ”computation size” was not directly studied: for example all of the problems arising with very short ”computation size” were not tackled.

In our study, the main focus is on steady state behavior, using a typical ”computation size” of at least 1440

(corresponding to the computation of 1440 elements) which is large enough for reaching peak performance while still allowing us to keep the operands in the L2 cache.

Measurements were performed on a stand-alone system (i.e. our benchmark code was the only user application running), only one measurement being performed at a time.

All of the timing measurements were performed using the *mov ar.itc* instruction to read the cycle counter of the processor itself.

Perfmon toolkit reading the various cache miss counters were used first to verify 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 with respect to one iteration: i.e. the measurements corresponds to the "average" number of cycles to compute one element of the result in the case of the BLAS1 kernels.

One of the major point of focus, will be the impact of offsets on performance. Therefore, 2D plots (isosurface) will be systematically displayed. A "geographic" color code is used: dark blue corresponds to the best performance (lowest number of cycles) while dark red corresponds to the worst performance. These 2D plots will be very useful in understanding qualitatively the spatial nature of the phenomenon. Also, standard curves corresponding to cuts through the 2D plots will be presented to give a precise quantitative measure. The values for these cuts correspond to fixed values of offset Y. These values (offset Y = 0 and 392) were somewhat picked up arbitrarily, the main goal of these "cuts" is to provide precise performance numbers.

4 Our Code Optimization Strategy

Since we are dealing with vector loops, in which iterations are independant, most of our optimisation will be focussed on specific ordering of Loads and Store instructions.

To illustrate the specific problems occuring due to the Itanium L2 banking organization, we will use two simple kernels only consisting only of memory instructions: the first one LxLy corresponds to a loop body in which two arrays X and Y are regularly accessed through Loads, the second one LxSy corresponds to a loop body in which one array X (resp. Y) is accessed through Loads (resp. Stores).

For the simple LxLy kernel, a first "naive" code generation would be to alternate between acces between array X and Y: i.e. Load X(0), Load Y(0), Load X(1), Load Y(1), Load X(2), Load Y(2), ...Such a code generation (called Interleaved LxLy) result in a large number of bank conflicts depending upon Offset X and Y values. The experimental results (Figure 1a and 1b) perfectly describes this problem with a grid pattern and three diagonal (corresponding to to the maximum number of bank conflicts).

Therefore, our load instruction scheduling consists first in vectorizing memory access then grouping together odd and even elements. The code generated looks like, Load X(0), Load X(2), Load X(4), Load X(6), Load X(1), Load X(3), Load X(5), Load X(7), Load Y(0), Load Y(2), etc ... With such a load reordering strategy, the Loads can be grouped by pack of 4, such that within a pack, the banks addressed are distinct. The result of such a reordering are depicted on Figurea 1c and 1d. The performance is pertectly flat, reaching the optimum of 0.5 cycle per Load X(i) Load Y(i) pair. All of the bank conflicts are eliminated.

Now, for the LxSy kernel, the strategy used for the LxLy can no longer be applied here because grouping stores by blocks of 4 stores per cycle hits a key performance barrier: a maximum of two stores can be issued per cycle.

Therefore the technique used consisting in grouping Load and Stores in the following manner: Load X(0), Load X(2), Store Y(0), Store Y(2), Load X(1), Load X(3), Store Y(1), Store Y(3) and so on ....

This scheduling technique will not completely eliminate all of the potential bank conflicts. However they will occur for very precise pairs of offset X and Y values: such cases will be depicted by narrow diagonals in the Offset X and Y plane. For addressing this last point, two variants differing by their software pipeline degrees, are generated. The bad zones will be located in different (disjoint) areas in the Offset X and Y plane. Then, by inserting a switch selecting the right variant depending upon offsets X and Y values, bank conflicts can be completly eliminated.

Finally, all of the BLAS 1 kernels (considered) were decomposed in terms of these basic templates: Load/Load and Load/Store to schedule memory instructions.

Due to the simplicity of the kernels, register allocation was performed after the load and store reordering. The strategy used was fairly straighforward, registers were grouped by blocks of four and managed as "vector registers" of length 4. This resulted in a fairly register pressure, it is why specific techniques are developped in the sequel of this paper.

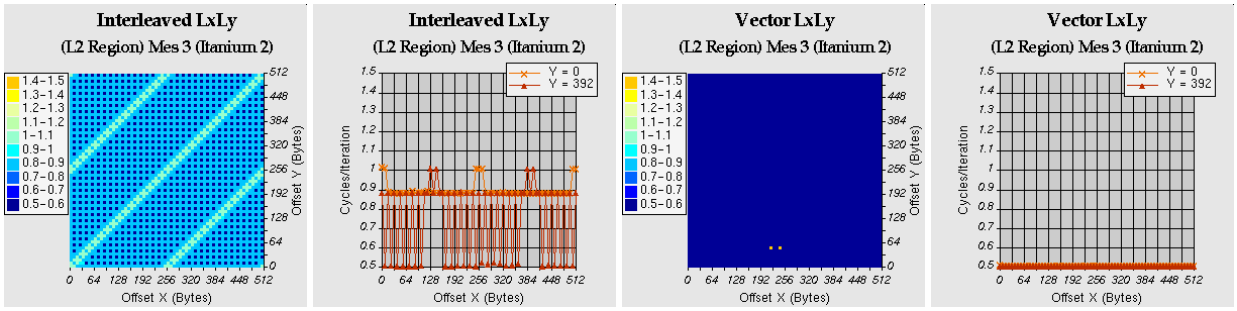

Figure 1. Load/Load Interleaved ((a) and (b)) and Vector Load/Load ((c) and (d)) on Itanium2

5 Experimental Results

Systematic experiments were performed, using more kernels than Daxpy and Copy, testing the various levels of the memory hierarchy). Also various compiler options and MKL libraries were evaluated. However, due to lack of space, only the most interesting results are presented here.

5.1 Copy Results

The Copy MKL results are given in Figure 2 (a) and Figure 2 (b). There is a complex grid pattern resulting in performance oscillations between 0.9 and 1.4 cycles. Superimposed with this grid pattern, a few diagonals (256 Bytes apart) are clearly visible. The code used by the MKL library is complex: loop is unrolled 8 times then Load Floating pair instructions are used. However, no attention was paid to bank conflicts, therefore the performance is oscillating mainly due to bank conflicts.

Our Copy Optimized results are given in Figures 2 (c) and (d). First, the performance is flat, fully independant of Offset X and Y values. The impact of bank conflicts is completly eliminated. Second the performance is close to optimal: without prefetch instructions, the optimal value would be 0.5 cycle per iteration (corresponding to a sustained rate of 2 Loads and two Stores per cycle). Now adding prefetch instructions, degrades slightly performance, a bit above 0.6 cycles.

5.2 Daxpy Results

The code generated by the V7.0 compiler is fairly complex. It contains three variants which are used depending upon loop length and Offset y values:

- Variant 1 corresponds to a code unrolled 8 times and uses Load Floating Point pair instructions on Y. Therefore this variant is only used when offset Y is a multiple of 16 Bytes;
- Variant 2 corresponds to a code also unrolled 8 times but without the use of Load Floating Pair instructions. Therefore it is used when offset Y is not a multiple of 16 Bytes;
- Variant 3 is not unrolled and seems to be used for loop count.

For the three variants, prefetch instructions on X and Y are inserted. Performance (see Figures 3 (a) and 3 (b)) is oscillating a lot. First a grid pattern can be observed somewhat similar to the one observed on Load Load Interleaved and Load Store Interleaved kernels. A detailed analysis of the code reveals that this grid pattern is clearly generated by bank conflicts: reference to arrays X and Y coexist in the same bundle. Again most of the diagonals patterns could also be attributed to bank conflicts.

Our Daxpy optimized code is obtained by combing the copy memory access pattern (Load on X and Store on Y) and the Load/Load access pattern (Load on Y).

As for the Copy, two variants are necessary to get rid of all of the diagonal zones. The performance results are presented Figures 3 (c) and 3 (d). Performance is perfectly stable/flat and always better than the V7 compiler. The performance of 0.9 cycles per iteration might seem disappointing while a simple count would lead to 0.75 cycles beause one iteration requires two loads and one store (each of them costing 0.25 cycles). However taking into account necessary prefetch instructions, performance at best is 0.82 cycles. Now our optimized version was only unrolled 8 times and therefore, unnecessary prefetch instructions were inserted. Unrolling 16 times would lead to the optimal performance.

5.3 Experiments Summary

The result presented above proves that vectorizing load/store operations is an efficient code optimization

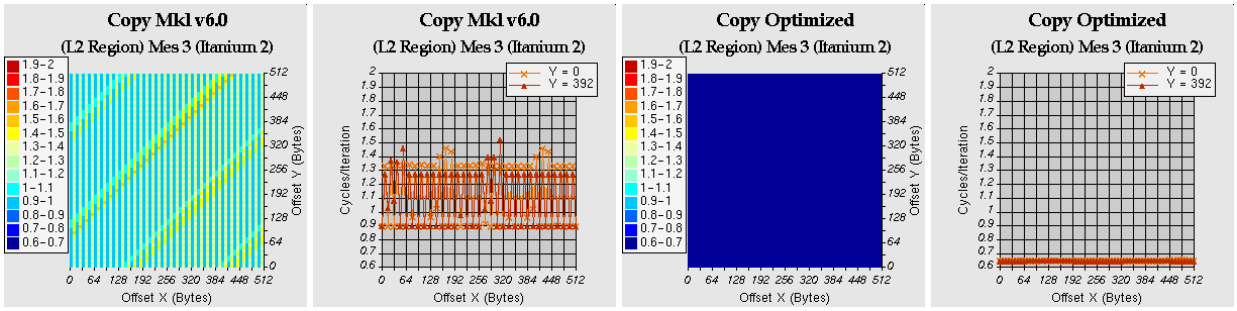

Figure 2. Copy MKL v6.0 ((a) and (b)) and Copy Optimized ((c) and (d)) on Itanium2

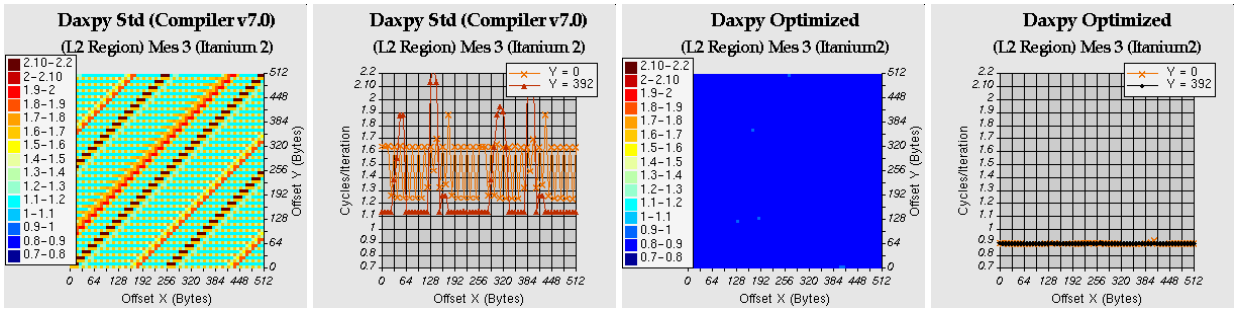

Figure 3. Daxpy Std v7.0 ((a) and (b)) and Daxpy Optimized ((c) and (d)) on Itanium2

technique. Experiments, not presented here due to lack of space, showed that when operands are located in L3 (instead of L2), similar bank conflict problems and can be solved again with our Load/Store reordering strategy.

However, the very naive strategy for managing registers generated a very high register pressure: close to 80 registers were necessary for our optimized DAXPY (fortunately Itanium provides 128 FP registers).

A first approach (that we used manually) would be to abstract the register file and the instruction set: i.e. consider that the 128 FP registers are organized as 16 Vector Registers of length 8 and that instructions are "extended" to vector versions operating on vector operands of length 8. Then perform a standard scheduling and allocation pass using these vector instructions and only 16 registers. Then in a post pass, every instruction is expanded/replaced by a corresponding sequence of 8 "scalar instructions". Such a strategy generates three problems: first it goes beyond what is stricly required (only reordering of Loads and Stores are required), second it does address the specific reordering needed for Load/Store pairs and third, then if R "vector" registers are consumed during the allocation pass, this will turn into $8 * R$ registers consumed (registers are handled by increment of 8).

The next section investigates these problems and develop specific technqiues for handling our Loads and Store reordering constraints.

6 Our Register Allocation Technique

In this section, we present our method that apply a cyclic register allocation in a loop that consist of arithmetic expressions. We exploit the fine grain parallelism within the operations by using the software pipelining technique. Furthermore, we vectorize memory access operations in order to avoid bank and load/store queues conflicts. We directly work on the loop DDG and modify it in order to guarantee the register pressure of any further subsequent software pipelining pass. This idea has already been presented in [9, 8]. Let us begin by a brief recall.

6.1 Our Register Allocation Technique

We consider a simple innermost loop that consist of a set of arithmetic expressions. It is represented by a graph $G = (V, E, \delta, \lambda)$, such that :

- V is the set of the statements in the loop body. The instance of the statement u (an operation) of the iteration i is noted $u(i)$. By default, the operation u denotes the operation $u(i)$;
- E is the set of precedence constraints (flow dependences, or other serial constraints), any edge e has the form $e = (u, v)$, where $\delta(e)$ is the latency of the

edge e in terms of processor clock cycles and $\lambda(e)$ is the distance of the edge e in terms of number of iterations.

Our instruction set model is assumed to be RISC-style (however multiple instructions can be issued during the same cycle), and admits multiple register types. We make a difference between statements and precedence constraints, depending if they refer to values to be stored in registers or not :

1. V_R is the set of values to be stored in registers.
2. E_R is the set of flow dependence edges. Since we focus on arithmetic expressions, the considered DDG is a tree necessarily (or a forest of trees). So, each variable of type t has a unique reader (killer), that we note :

$$k_u = v \in V \mid (u, v) \in E_R$$

A software pipelining is a function σ that assigns to each statement u a scheduling date (in terms of clock cycle) that satisfies the precedence constraints. It is defined by an initiation interval, noted II, and the scheduling date σ_u for the operations of the first iteration. Iteration i of operation u is scheduled at time $\sigma_u + (i-1).II$. For all edge $e = (u, v) \in E$, this schedule must satisfy:

$$\sigma_u + \delta(e) \leq \sigma_v + \lambda(e).II$$

Classically, by adding all such inequalities on any circuit C of G, we find that II must be greater than or equal to $\max_C \frac{\sum_{e \in C} \delta(e)}{\sum_{e \in C} \lambda(e)}$, that we will denote in the sequel as MII.

We consider now a register pressure ρ and all the schedules that have no more than ρ simultaneously alive variables. Any actual following register allocation will induce new dependencies in the DDG, hence register pressure has influence on the expected II, even if we assume unbounded resources. What we want to analyze here is the minimum II that can be expected for any schedule using less than ρ registers.

A register allocation scheme consists of defining the edges and the distances of reuse. That is, we define for each $u(i)$ the operation v and iteration $\mu_{u,v}$ such that $v(i + \mu_{u,v})$ reuses the same destination register as $u(i)$. This reuse creates a new anti-dependence from k_u to v with a distance $\mu_{u,v}$ to be defined. We proved in [8] that the number of allocated register is equal to the sum of all anti-dependence distances, i.e., $\rho = \sum \mu$.

Hence, controlling register pressure means, first, determining which operation should reuse the register killed by another operation (*where should anti-dependences be added?*). Secondly, we have to determine variable lifetimes, or equivalently register requirement (*how many iterations later (μ) should reuse occur*)? The lower is the μ, the lower is the register requirement, but also the larger is the MII.

The reuse relation between the values (variables) is described by defining a new graph called *a reuse graph* that model the reuse relation between statements. Two statements u, v are connected by an edge iff v reuses the register freed by u. This means that there is an anti-dependence from k_u to v with a distance $\mu_{u,v}$. For general loops with n statements, we have an exponential number of reuse choices.

If the underlying processor contains a rotating register file, the reuse graph must be hamiltonian necessarily [8]. The cost of such constraint is at most one extra register when compared to other reuse decisions.

Looking for an optimal reuse graph with optimal anti-dependence distances is an NP-complete problem[8]. Our experiments show that the compilation time becomes intractable when the loops contains more than 10 statements. However, if we fix the reuse graph, then looking for a minimal register allocation becomes easier. The next section explores this problem.

6.2 Fixing Reuse Edges

We consider now the problem of minimal register allocation under a fixed execution rate II when we fix reuse edges (i.e., anti-dependences). In following, we assume that $E' \subseteq E$ is the set of these already fixed anti-dependences edges (their distances have to be computed). Deciding (at compile) time for fixed reuse decisions greatly simplifies problem of cyclic register allocation. It can be solved by the following integer linear (intLP) program, assuming a fixed desired initiation interval II.

$$\begin{array}{ll} \text{Minimize} & \rho = \sum_{(k_u,v) \in E'} \mu_{u,v} \\ \\ \text{Subject to:} & \\ II \times \mu_{u,v} + \sigma_v - \sigma_{k_u} \geq \delta' & \forall (k_u, v) \in E' \\ \sigma_v - \sigma_u \geq \delta(e) - II \times \lambda(e) & \forall e = (u, v) \in E - E' \end{array} \quad (1)$$

Here, δ' is the latency of the anti-dependence, which depends on the target architecture.

Vectorizing load/store operations introduces new serial constraints that our register allocation must respect. The next section extends our model to take into account vectorization.

6.3 Extension to Vector Register Allocation

Our model can be used to take into account vectorized operations. For instance, if we want to vectorize k operations $u_1, ..., u_k$, we only have to add a circuit C (with a null latency and distance) that joins these operations. Such circuit reflects the fact that the connected operatins are constrained to be executed in parallel, which is similar to form a single vector instruction. Then, we can apply our register allocation technique. Note that, sometimes, we have to unroll the loop k times to vectorize the k considered operations. Fig. 4 shows an example of a vectorization that connects 4 operations into a null circuit.

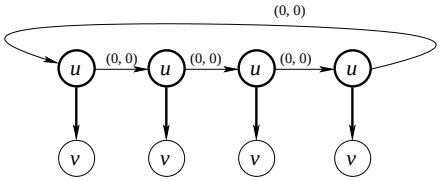

Figure 4. Vectorized Loop Example

This technique is better that classical vector register allocation, since it applies vectorization to only a subset of operations. The non-vectorized operations are free to be scheduled at the fine-grain level, so as to minimize the register requirement and to minimize II. Classical vector register allocation techniques, such that [1, 3], 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.

7 Experiments with Register Allocation

We have developed a tool that perform cyclic register allocation on a rotating register file by fixing a hamiltonian reuse circuit. We focus on floating point registers. We used 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 load/store vectorization on the register requirement versus the initiation interval II. We experiment four variants (vectorization versions) for each micro benchmark. Note that the 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, i+1, i+2, i+3)]
[St Y(i, i+1, i+2, i+3)]
```

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

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

Variant 3 The loop is unrolled 8 times, and each 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, i+2, i+4, i+6)]
[Ld X(i+1, i+3, i+5, i+7)]
[St Y(i, i+2, i+4, i+6)]
[St Y(i+1, i+3, i+5, i+7)]
```

Variant 4 The loop is unrolled 4 times. Each 2 loads are vectorized with 2 stores, i.e., each vector operation consists of 2 loads and 2 stores. Each block of 4 loads (stores) accesses only even or odd elements. Furthermore, the distance of stores is kept as a parameter. Actually, we experimented a distance of 1, 4 and 8 iterations.

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

Figure 5 to 8 plots our experiments for each vectorization variant. We plot the results of non vectorized loops (the unrolled loops) in order to highlight the difference in terms of register requirement.

All these experiments provide us two main trends, depending on the vectorization technique.

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

1. such vectorization techniques does not alter MII ;

2. the register requirement is increased. 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 the variant 4. In all these codes, loads and stores are packed into the same vector instructions. We have two main conclusions :

1. such vectorization techniques has a high impact on MII. In all experiments of variant 4, the MII increases. Indeed, the vectorization introduces new critical circuits into the loops, since we are connecting loads with stores into the same circuits. The value of the new MII depends on the distance d of the stores : the higher is d, the lower is MII.

2. the register requirement is substantially increased. This is because we vectorize loads with stores, thus the variables lifetimes are constrained 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 is d, the higher is the register requirement.

Note that the register requirement in our experiments can be reduced by improving our register allocation technique. Since our method fixes an arbitrary hamiltonian reuse circuit, we may connect some statements belonging to the same vector instruction. Thus, since such statements are constrained to be executed in parallel, the register requirement may not be reduced in the best way. Our future work will be to think for better reuse circuits for such vectorized loops.

8 Conclusion and Future Work

Modern microprocessors rely on complex cache systems to deliver top performance. The dark side of the coin is the resulting complexity, not only for designing them but also for exploiting them efficiently even on simple codes.

Our studies of simple BLAS1 kernels on the Itanium2 have clearly shown that the banking structure of the L2 has a major impact on performance: ignoring the interleaving of the L2 cache, would result in highly oscillating perforamnce. We demonstrated that a clever memory instruction scheduling based on vectorization could get rid of all of the bank conflicts. The cost of such techniques in terms of register space was then analyzed and different optimizations were proposed.

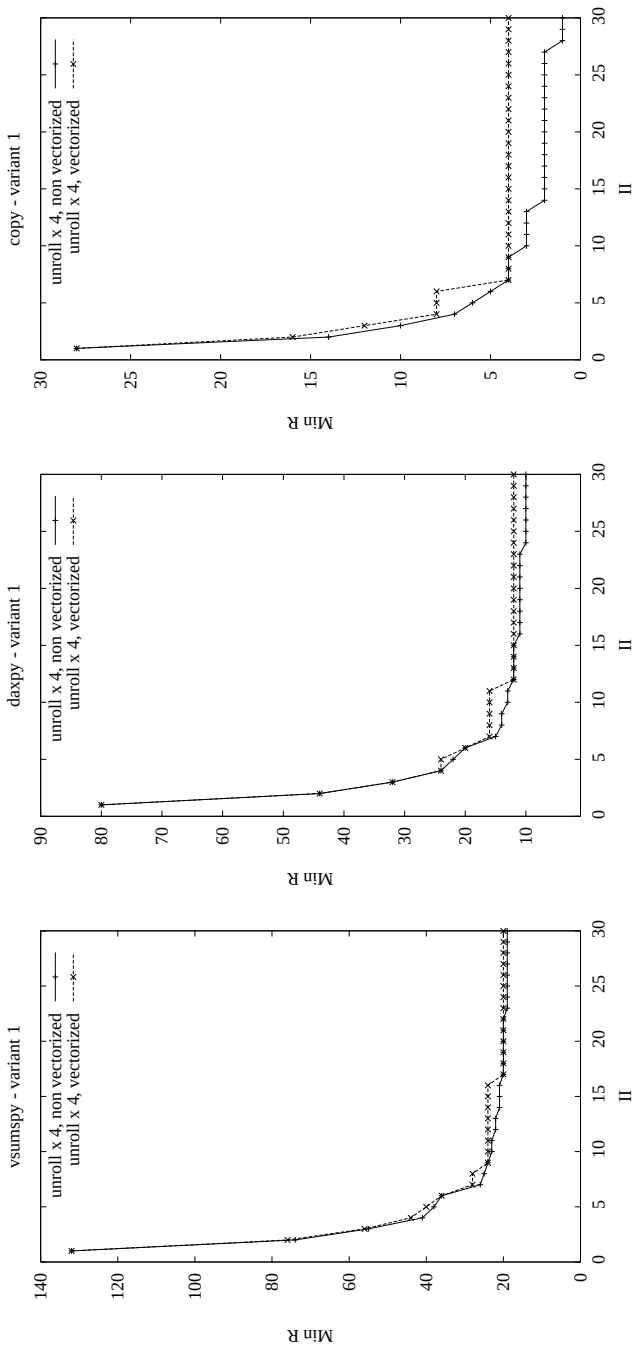

Figure 5. Register Requirement for Variant 1

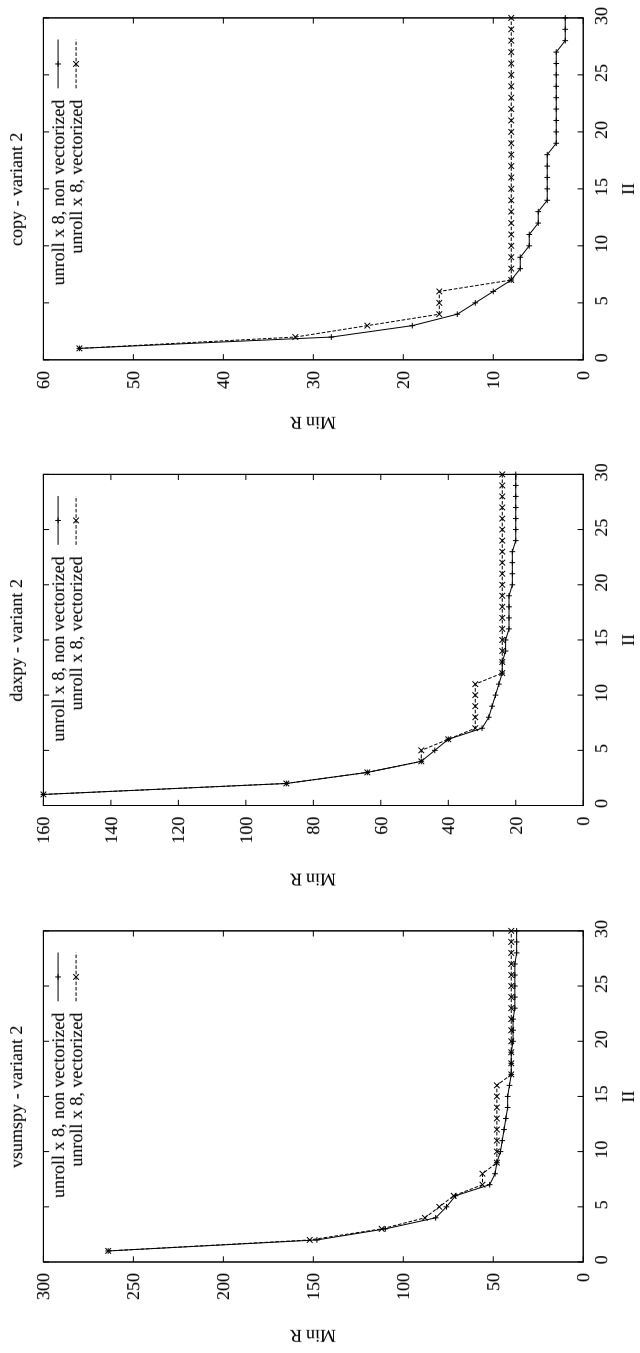

Figure 6. Register Requirement for Variant 2

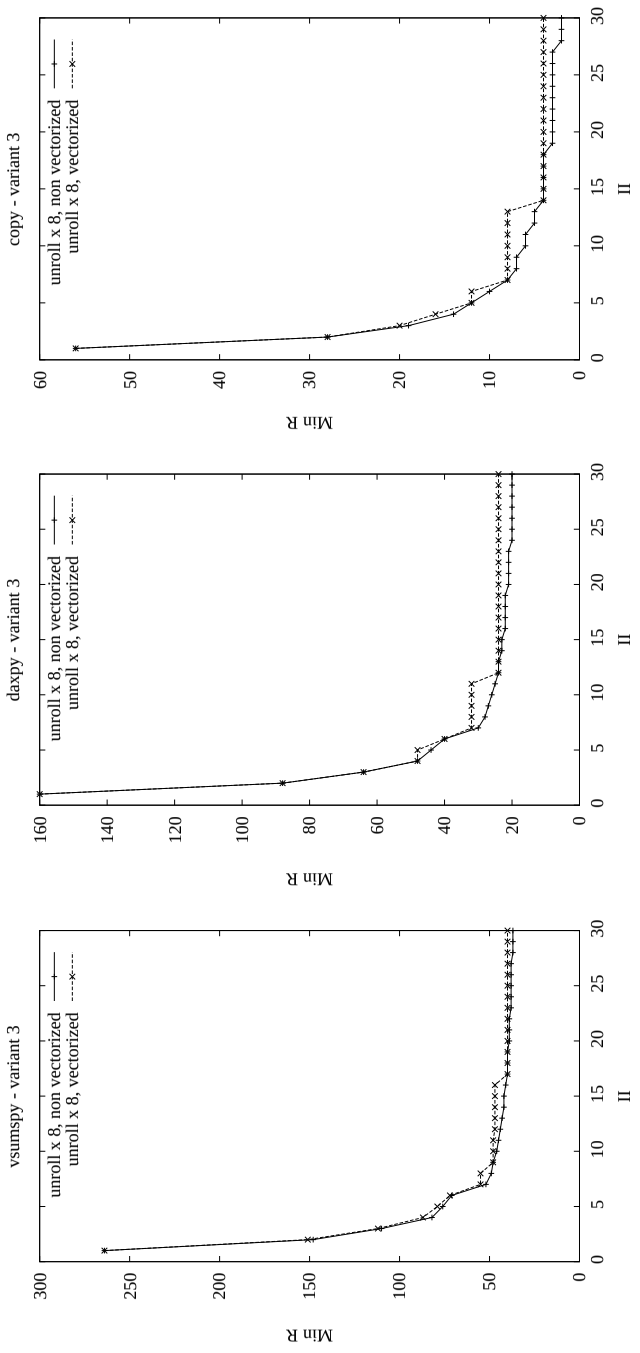

Figure 7. Register Requirement for Variant 3

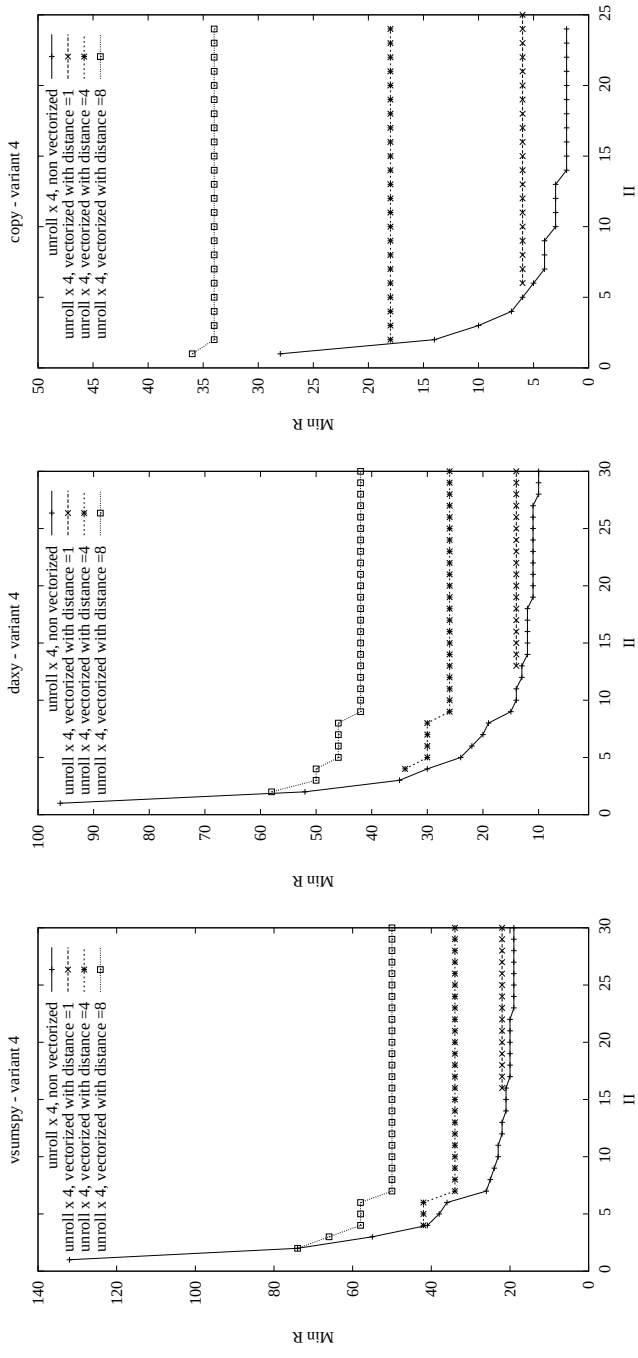

Figure 8. Register Requirement for Variant 4

Now, we performed a similar analysis on superscalar processors (such as Power 4 and Alpha 21264). Interestingly enough, for these architectures, vectorization is also usefull. However, it is so for other reasons, essentially, because it limits conflicts in the load/store queue due to limited disambiguation capabilities (cf Figures 9 and 10). As it can be seen, on these figures vectorization does not allow to get rid of all of the pathological behavior but it allows to reduce the magnitude of the performance loss.

This work will be extended into two major directions:

- More complex kernels involving a larger number of arrays and more complex arithmetic operations should be studied. Although our preliminary results on register allocation are promising, they need to be tested and analyzed in a more general framework.

- Main memory access deserves a similar study. Already, some preliminary tests have confirmed us with the good performance capabilities of the vectorization strategy.

References

[1] R. Allen and K. Kennedy. Vector Register Allocation. *IEEE Transactions on Computers*, C-41(10):1290–1317, Oct. 1992.

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

[3] D. Bernstein, H. Boral, and R. Y. Pinter. Optimal Chaining in Expression Trees. *IEEE Transactions on Computers*, 37(11):1366–1374, Nov. 1988.

[4] J. C. Huck, D. Morris, J. Ross, A. D. Knies, H. Mulder, and R. Zahir. Introducing the IA64 Architecture. In *IEEE Micro*, Sept. 2000.

[5] Intel. Intel Itanium2 Processor Reference Manual for Software Develoment Optimization. (251110-001), June 2002.

[6] W. Oed and O. Lange. On the Effective Bandwith of Onterleaved Memories in Vector Systems. *IEEE Transactions on Computers*, C(34(10)):949–957, Oct. 1985.

[7] H. Sharangpani and K. Arora. Itanium Processor Microarchitecture. *IEEE Micro*, 20(5):24–43, Sept./Oct. 2000.

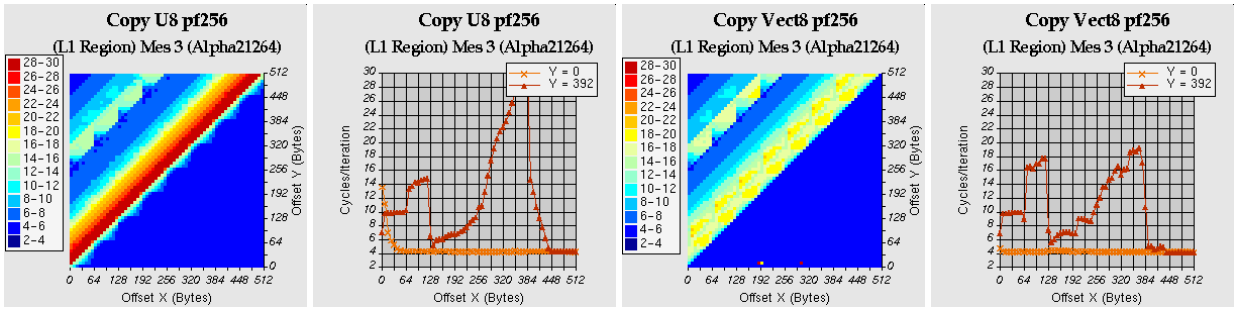

Figure 9. Copy U8 ((a) and (b)) and Copy Vect8 ((c) and (d)) on Alpha21264

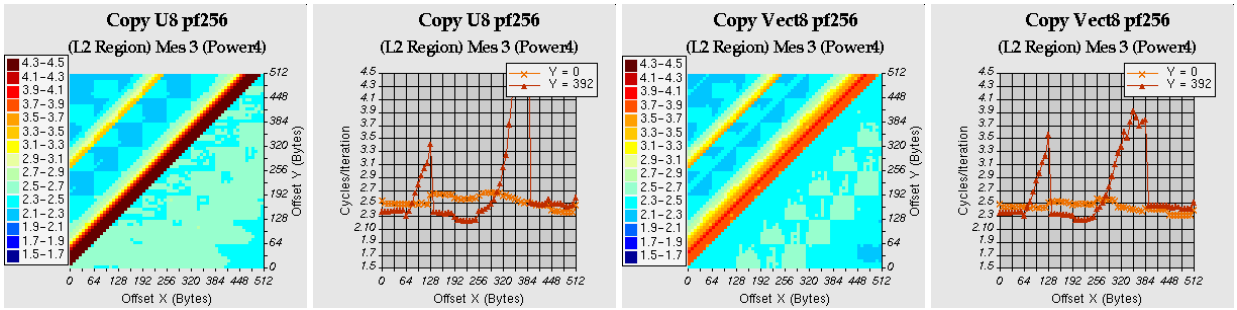

Figure 10. Copy U8 ((a) and (b)) and Copy Vect8 ((c) and (d)) on Power4

[8] S.-A.-A. Touati. *Register Pressure in Instruction Level Parallelisme.* PhD thesis, Universit de Versailles, France, June 2002. ftp.inria.fr/INRIA/Projects/a3/touati/thesis.

[9] S.-A.-A. Touati. Minimizing Register Requirement in Loop Data Dependence Graphs. In *Compilers for Parallel Computers*, Amsterdam, The Netherlands, Jan. 2003.