Squeezing more CPU performance out of a CRAY-2 by vector block sheduling

Christine Eisenbeis, W. Jalby, A. Lichnewsky

INRIA

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél. (1) 39 63 55 11

Rapports de Recherche

N° 841

SQUEEZING MORE CPU PERFORMANCE OUT OF A CRAY-2 BY VECTOR BLOCK SHEDULING

Christine EISENBEIS
William JALBY
Alain LICHNEWSKY

MAI 1988

RR-0841

SQUEEZING MORE CPU PERFORMANCE OUT OF A CRAY-2 BY VECTOR BLOCK SCHEDULING

Christine Eisenbeis
William Jalby
Alain Lichnewsky
I.N.R.I.A.
Domaine de Voluceau
78153 Le Chesnay CEDEX

Résumé

Ce rapport traite de l'ordonnancement des instructions vectorielles du CRAY 2 à la compilation. L' approche retenue utilise une modélisation simplifié du fonctionnemment en mode vectoriel, et se prête a' une utilisation inhabituelle des techniques de compaction de microcode développées par les auteurs. la méthode utilise deux passes, la première pour ordonnancer les instructions, la seconde pour l'allocation des registres et de la sauvegarde en memoire locale. Cette approche est validée par des résultats expérimentaux montrant des améliorations allant jusqu' à 50% par rapport aux compilateurs CFT77. De plus les résultats obtenus ouvrent des perspectives nouvelles sur l'évaluation de l'intérêt du mécanisme de chaînages dans les architectures vectorielles.

Abstract

Compile time scheduling of vector activities on the CRAY 2[1] is studied using a simplified model of the vector instruction stream. Due to several of the hardware characteristics of the machine, an approach using much know-how obtained on Array-Processor microcode scheduling by the authors is shown practical. It calls for a pass of loop scheduling followed by a pass of resource allocation. Actual benchmarks of the resulting code are shown, exhibiting speed-ups as large as 50% over the current CFT77 compiler. Our results also give a new perspective in the comparison of vector chaining and non-chaining processor architectures.

[1] Computations performed on the Cray 2 at CCVR, Palaiseau (France)

INTRODUCTION

In the race towards peak performance, all the forms of parallelism present in the code have to be exploited. In fact, the challenge is double: parallelism must first be detected and then matched with the architecture. Moreover, it has to be determined if these problems have to be solved by hardware, software or a subtle cooperation of the two.

At the higher level, the advent of vectorizers and more recently parallelizers, has provided a very powerful and successful approach. The success of vector architecture has been largely due to the availability of vectorizing technology enabling the user to benefit from the speedup of pipelined units.

A lower level of parallelism exists in most modern architectures, permitting the simultaneous operation of independent units for computation, memory access, control. This means that parallel activities must be scheduled and synchronized, in order to maintain the code semantics while enhancing the overall performance. The memory units often have an asynchronous behaviour that must be matched to a general synchronous organization.

The hardware conflict resolution approach is embodied in the Tomasulo algorithm, the Scoreboard of the CDC - 7600 and Cray "issue" strategy. (cf.[1,26,6,7]) These techniques have very different run time scheduling capabilities.

The other extreme relies on the software to solve these problems, and is exhibited by RISC and horizontally micro coded – or VLIW – architectures. Going from the first set to the second, the requirements for the software increase: good scheduling is an optimization feature in the first case which presents no interlock detection need, the second assumes a very good scheduling capability and conflict resolution.

The purpose of this paper is to describe code scheduling techniques well adapted for the Cray 2 making a very efficient use of the hardware resources for vector loops. An amazing corollary of our proposed techniques, is that we show that our code scheduling algorithm overcomes the absence of chaining on the Cray 2. More precisely, we achieve similar performance as for an extended architecture incorporating full chaining mechanism, with the same memory bandwidth.

In section 1, we give an overview of the different techniques used for instruction scheduling.

In section 2, we describe how the Cray 2 is modeled in our study. We consider vector instructions as atomic blocks, whose interaction is very simple, due to the absence of chaining. The latency of the memory access pipeline, which is a well known characteristic of this machine, is adequately rendered using a two stage block level pipe. Moreover, our model can be treated without the complex linear programming methods of Aryia [2] taking into account potential chaining in the Cray 1.

In section 3, we describe the software scheduling of the arithmetic units and memory access units. The technique used is cyclic scheduling which is basically local scheduling of the body of the loop taking into account the cyclic nature of the loop. This technique was developed for optimizing code for horizontally microcoded architectures such as the ST100, by one of the authors. The asymptotic speed obtained is very similar to the one obtained by trace scheduling; moreover due to the use of cyclic nature of a loop, the peak performance is reached for relatively small number of iterations (relatively small $N_{\frac{1}{2}}$).

In section 4, we describe the register allocation strategy. In fact, the scheduling algorithm of the section 2 allows us to overcome the lack of chaining, the price to pay is a higher register usage (increased residence time of data in registers): it is the classical well known

space time tradeoffs. This problem is solved by a graph coloring algorithms; however, the graphs derived from our vector loops, exhibit a strong regularity allowing us to devise an optimal polynomial register allocation algorithm. In this phase, spilling is also introduced, using the fact that spilling can be done in local CRAY 2 memory in complete overlap with other operations, i.e. spilling can be free when scheduling allows.

In section 5, we present systematic benchmarking of our proposed algorithms. We describe experimental results on a CRAY 2 obtained using code generated according to our techniques. We compare these results against those obtained using the current CRAY 2 compilers. Our techniques is clearly superior even compared with the CFT77 compiler which is using some sort of trace scheduling technique: as an example, we get 50% performance improvement on simple loops such as the multiplication of 2 complex vectors. In many cases, we obtain speeds which cannot be beaten even by hand compiling, because we saturate one of the hardware resources, which is often the global memory path.

1 MOTIVATIONS

In this section, we review briefly the main approaches used for using parallelism at the low level between the functional units.

For scalar processors, very sophisticated and complex hardware techniques have been designed (Scoreboard, Tomasulo's algorithm) in order to schedule dynamically at runtime, the instruction flow. At the opposite, recently the RISC and VLIW architectures are using simple hardware mechanisms, leaving most of the burden of code scheduling to the compiler. Although extensive data flow analysis is required at compile time, this approach appears very promising.

For vector processors, the associated concepts of vector instructions and registers make the instruction scheduling problem harder and more crucial: the granularity being larger both expressed in processor cycles and allocated resources, the interactions between them are more complex and the performance is more sensitive to the quality of scheduling.

A first approach is the Vector-CISC technique (CDC 205, ETA 10, IBM 3090 VF, ALLIANT FX8) which imbeds most of the lower level parallelism into the instruction set. The instructions are issued sequentially, thus not requiring any run time scheduling. The role of the compiler is to optimally decompose, for instance by pattern matching techniques, arithmetic expressions into templates corresponding to the instructions supported by the hardware. [13] This technique although constituting an interesting solution due to its simplicity and relatively good performance for simple cases, suffers severely from its lack of flexibility: a very large instruction set may be required in order to cover all the practical cases for which high performance is sought. In this context vector registers can be used to realize optimizations such as reuse of data and intermediate results, and simplify the pipeline organization by breaking them into simpler parts.

On the other hand, vector RISC architecture such as Cray and Fujitsu permits to combine intimately hardware and software optimizations, for an increased global efficiency. This requires registers and the ability to execute several vector instructions concurrently. All the present machines of this class, except the Cray-2, use the concept of chaining. This allows to build up dynamically macropipelines, achieving similar run time flow of data and therefore performance as the CISC machines. However, the performance achievable is very dependent upon the compiler, which has to order the instructions ex-

ploiting data dependencies permitting chaining while avoiding dependencies which result in wasted cycles due to interlocks. At first glance, the CRAY2 with the absence of chaining seems to appear as a step back. We will show in this paper that much of the overall performance can nevertheless be obtained by software scheduling techniques, making intensive use of the vector registers and the local memory, thereby giving a new perspective to the speed / complexity tradeoff.

2 Machine Model

In the first part of this section we describe the general structure of the target architecture and how its behavior is modeled for the code optimization procedure. Then, we explain how we imbed the Cray 2 architecture in this model. The key point here is to abstract the architecture in order to apply efficiently optimization tools, while preserving the essence of the architecture. As we will see in section 5, the pertinence of the model is clearly validated by the experimental results obtained.

2.1 Description of the generic architecture

In our architecture model, we distinguish two kinds of hardware resources:

- Functional Units: floating point and integer arithmetic, memory access, data paths subject to reservations.
- Storage Units: registers, local memory, main memory.

To operate on the run time behaviour of these resources, the concept of reservation tables permits to describe the use of resources for a series of discrete time steps. The time steps, also called granules, are not necessarily cycles. The table explicits, for each of these time steps, the occupancy of the resources subject to reservation. These resources are defined in terms of the above hardware elements, and of their organization with respect of the flow of data. Thus, when functional units are pipelined, their different stages are considered as different reservation entities.

Following this correspondance, instructions, which control the run time activity of physical resources, are associated with elementary reservation tables – also called templates – which describe the impact of the instruction execution on the reservation table. Each template may specify reservations of several resources spanning several time granules. Scheduling the basic operations and checking for interlocks now amounts to placing the templates onto the reservation table. The role of the scheduler is to compact the reservation table templates without violating dependence and resource constraints thus minimizing the actual execution time.

Array processor architectures (FPS 164, ST 100) fit naturally in this model, letting templates describe related sets of micro-operations. The code scheduling becomes the microcode compaction problem. However, as shown next, this framework can be applied to code optimization to a larger class of architectures.

2.2 Modeling the Cray 2

In this section, we describe how we choose the model of the Cray 2, which contains the time granule, the reservation table and the templates.

Since we are primarily interested in vector loop optimization, we choose a granularity corresponding to a full length vector elementary operation, or a reservation implied by it. The practical size of a granule, or macrocycle, is

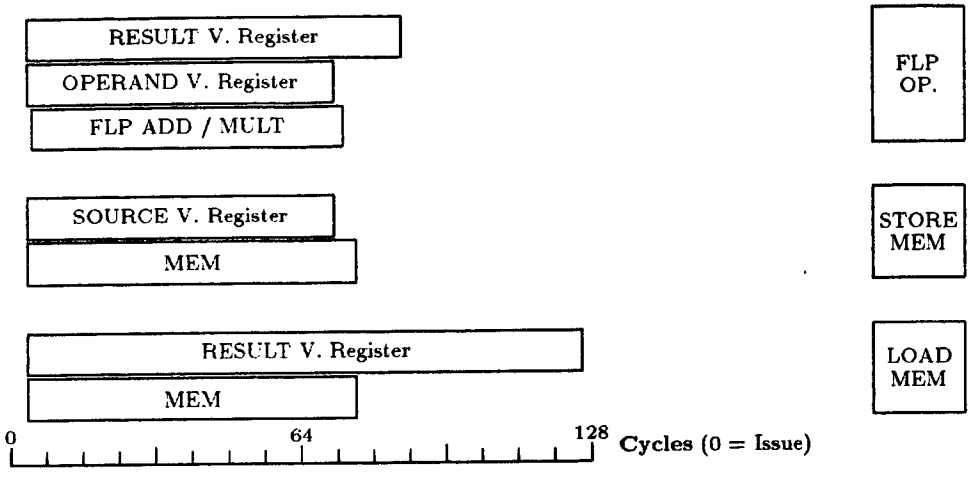

Figure 1: Instructions and Resource Reservations

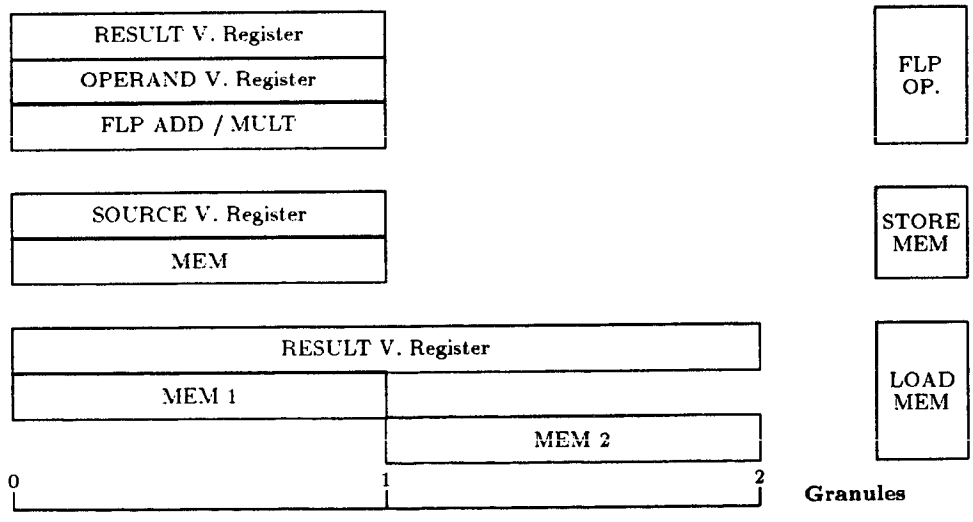

Figure 2: Templates

thus between 64 and 82 cycles. Some of our results will indicate explicitely the granule size used, in the others a size of 82 is implicit. The experimental data below will also show the result of our scheduling for smaller vector length, but it must be emphasized here that the whole optimization process is only conducted for vectors of length 64. Neglecting to optimize vector instructions of length less than 64 is not too severe since this occurs only once per loop after the classical strip mining loop optimization. The complete code optimization process has become hierarchical, and a second pass is necessary to emit and schedule the scalar instructions. It also performs local – within time granule – vector code fine tuning, working with a time granularity of a single cycle, which is more that one order of magnitude smaller. This second pass can be made quite easily because the most crucial vector block scheduling is known, and because many reservation table slots remain free to schedule the scalar operations.

The resources which must be considered in the reservation table are the functional units which are all considered single stage, except the main memory access path, which is considered a 2 stage pipeline. Thus, in the case of a main memory store or load the access is split in 2 macrocycles and main memory accesses might be partially overlapped. The local memory access path is single stage. The vector registers should have entries in the reservation table, but we take them into account only in a separate phase, following the basic loop scheduling of the previous resources, which uses virtual registers.

The templates are derived straightforwardly from the description of the instructions, by marking the resources occupied during each time granule. The busy times are simply rounded up to the next granule. For example, figure 2 gives the templates with a macrocycle of 82 clock periods. The number 82 corresponds to longest reservation time, in processor cycles, which occurs for the result register in a floating point add. (Cf. Figure 1).

Now that the modelling process has been described, the choice of the unique granule period can be examined more closely. At a first glance, this approximation may seem an oversimplification in the sense that in our model, for example the floating point adder will be considered busy for 82 ticks while, in reality, this number drops to 68. In fact, the error introduced is less than 20 %, as pointed out by the comparison figures 1 and 2 . A more detailed analysis reveals that our model introduces 2 kinds of wasted cycles: first at the level of the storage units – which does not constitute a penalty for the execution time, but shows higher register usage – and second at the level of the functional units. In the latter case the cycles left over can be systematically used to schedule scalar operations.

Going further in that direction shows that the choice of the macrocycle and the reservation tables can be changed for different optimization purposes. For example, a macrocycle of 72 cycles can be adopted. This choice will give a better usage of the functional units, because the approximation is more accurate for their reservation time, at the cost of a more expensive allocation for the result register. This might be worthwhile for vector loops without scalar operations. Moreover, the reservation tables can be used to force the code scheduler to increase the distance between 2 dependent instructions. This might be extremely useful for damping the variations of the main memory latency and throughput. For example, the result register for a load can be reserved for 3 macrocycles, inducing the scheduler to delay the use of that register. This will avoid to get an interlock at execution time due to a memory access conflicts. Another case is the scatter store operation, for which actual behaviour can be modelled quite closely, after adequate

experimentation. Our model offers a large degree of flexibility which can be adjusted for various optimization purpose.

3 Scheduling of the Functional units

As explained in the previous section, our code optimization strategy distinguishes between vector operations and scalar operations: in a first coarse granularity pass, we schedule vector operations, then. in a subsequent pass, we schedule scalar operations working at the machine cycle granularity.

The optimization of vector operations , itself is decomposed in two consecutive phases:

- scheduling of the functional units assuming an unbounded amount of storage resources (infinite number of registers). This means that the physical registers entries in the reservation table can be ignored. However, the life span of virtual registers is computed and used in the allocation phase.

- management of the storage units: register allocation, spilling in local memory

Assigning real registers too early during the scheduling phase is unnecessary due to the interchangeable character of the registers. Furthermore, such an allocation would require to add several lines to the reservation table, making the scheduling more complex and less efficient.

3.1 Preprocessing

We are working on a vector loop which has been already detected and processed by a vectorizer (in our case VATIL [23]). The dependence graph [19] is available at this stage and several optimizations including strip mining have been performed. The dependence information – intra- and inter-iterations – is used during the scheduling process to preserve the semantics of the program. The strip mining operation consists in breaking – blocking – all the vectorized loops by blocks of 64 iterations. For a loop of N iterations, the result is an outermost loop of $\lceil \frac{N}{64} \rceil$ and an innermost loop of 64 iterations.

Since vector operations are considered as atomic blocks, the term iteration (resp. loop) will denote exclusively outermost iteration (resp. outermost loop).

3.2 Code Generation of the Loop Body

In this phase, we generate an intermediate vector code for the loop body, which is used as basic pattern for the cyclic scheduling of the iterations.

The registers are handled in a virtual manner according to a single assignment rule: the number of registers is supposed unbounded, at each time a register is needed to hold a value resulting from a load or an operation, a new register is allocated. In the cyclic scheduling, each iteration of the loop body will be represented by a similar copy of the generated intermediate code. The virtual register defined in the generic loop body are named R_k. The copy corresponding to iteration i will create the registers numbered $R_1(i)$ through $R_k(i)$. The virtual registers which are either live on loop entry or invariant, are named $R_l(0)$. Globally, as a consequence of the single assignment rule, each virtual register is written once but may be read several times. The main advantage of such a technique is to avoid introduction of artificial dependencies due to a bad allocation [10].

The intermediate code is represented by a graph (called I-graph) whose vertices are the instructions. The edges are of two kinds: first

```
CDIR$ DOVEC
      DO 22 i = 1 ,1000 ,1
      zr(i) = xr(i)*yr(i)-xi(i)*yi(i)
      zi(i) = xi(i)*yr(i)+xr(i)*yi(i)
22    CONTINUE
```

```
BEST CYCLE LENGTH        LMM   = 6
LOOP LENGTH              LB    = 6
INSTRUCTION COUNT        TMC   = 25
N 1/2 ITERATIONS         NIT   = 1.16
N 1/2 VECTOR LENGTH      NVL   = 75
ITERATIONS PER SECOND    VIT   = 36.13367 MITS
THEORETICAL SPEED        VOP   = 216.802 MFLOPS
```

Figure 3: Source Code and MIMOSA Statistics

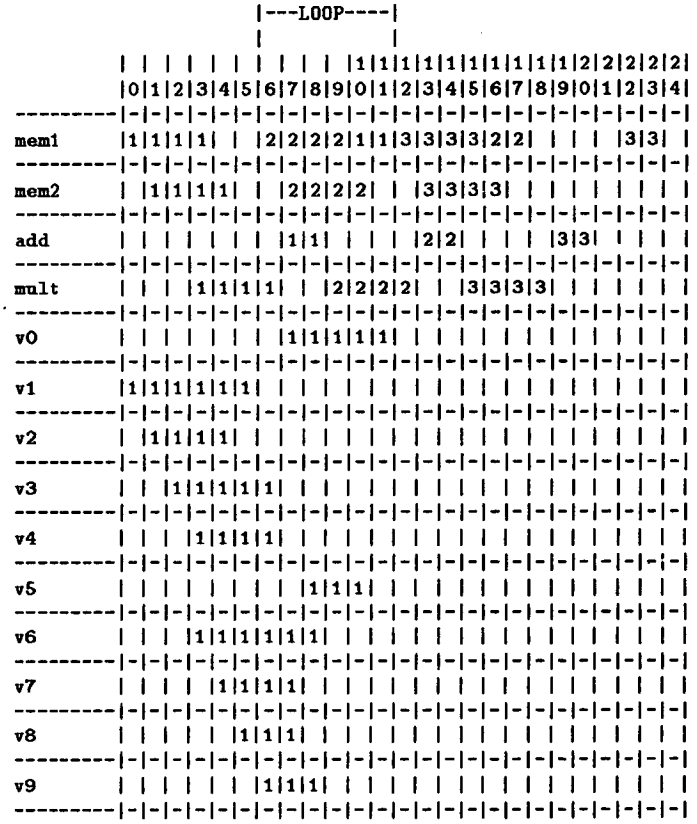

Figure 4: Reservation Table after Pass 1

those representing the intra-loop data flow which are labeled by the corresponding virtual register name and second those representing the inter-iteration dependence information (distance vector and dependence type). Classical optimization procedures such as finding common subexpressions, redundant and dead-code elimination are performed on it.

3.3 Scheduling

The scheduling of the whole loop involves two intimately related subproblems: scheduling of the generic loop body code, and then scheduling of the successive iterations. Our strategy will be to use the cyclic scheduling technique which was developed for array processor microcode compaction [24], and which tackles the two issues simultaneously.

In this method, all the iterations are scheduled exactly following the same pattern derived from the generic loop body. Therefore each iteration contributes to the reservation table by a translated copy of the generic loop body reservation table RT. Iterations are started with a constant unknown period of d granules. The global scheduling problem is to build a reservation table RT for the loop body and find a period d, such that:

- the sequence $RT+kd$ does not conflict: at each macrocycle, a given functional unit is not used more than once.
- the semantic dependence constraints are satisfied.
- the total execution time is minimized.

Let us notice that the execution of one iteration may span more than d granules and therefore at a given cycle several successive iterations might be concurrently executing. Consequently the local scheduling has to take *cyclicity constraints* into account in order to avoid resource conflicts between iterations. This simply means that dependences are satisfied under the periodic initiation mode, and cyclicity constraints can readily be computed from the dependence graph as a function of d. The payoff of that additional complexity is the perfect chaining between iterations.

As a preliminary step, we first complete the I-graph with a dummy terminal sink node S. Then we label each vertex v with the length of longest path from v to S. After, the vortices are sorted into a list $L = \{v_1, ...v_k\}$ by decreasing label.

For a given period d, we schedule the loop body according to a list scheduling strategy slightly modified in order to take into account both semantic and cyclicity constraints. In

```
begin
  n = 0
    while L ≠∅ do
      while any v in L
        can be scheduled at cycle n
         then
          schedule v
          L = L \ {v}
          update functional unit usage
          update cyclic constraints
        endwhile
      n = n +1
    endwhile
end
```

Figure 5: Basic Scheduling Algorithm

figure 5, "can be scheduled" means that the semantic and cyclic constraints are preserved. Taking into account cyclic constraints while scheduling the generic loop body allows to optimize globally the execution time, by permitting a small value of the period d. In fact, this is equivalent to the insertion of delays suggested in [11] and [20], but we have more flexibility in the process because much semantic information is available. However, since at that

stage, we are not considering physical register assignment, our introduction of delays will not add complexity to the reservation table describing functional units.

The period d is simply determined by computing for each functional unit t, the number of macrocycles n_t where t used is during one iteration. In our current implementation, d is chosen starting at the maximum value of n_t; this value of d corresponds to the saturation of one of the functional unit. Further work is required to investigate this choice.

The success of such a strategy of scheduling is mainly due to the fact that the templates are extremely simple: most FUs is not used more than one macrocycle.

3.4 Global Register Usage

In this section, we describe how to keep track of the register requirements during the scheduling phase.

We call the birth date of a virtual register the cycle where it is assigned a value. Correspondingly, its the death date is the cycle where its content is used for the last time. Between its birth and death, the virtual register is said live. At each cycle, the total number of live registers is determined and the maximum number of live registers over the whole execution is called the critical register quantity (CRQ). Note that the number of live register during a period of d cycles does vary during startup and pipe flushing times, and that CRQ correspond to the steady state of the execution. This number clearly corresponds to the minimum number of physical registers needed to execute the loop without spilling.

Many useful properties of registers are common of a register class, which correspond to the virtual registers of the generic loop body. Namely, the class R_k contains :

$$R_k = \bigcup_{i=1..n} R_k(i).$$

A register class C is said critical if suppressing the reservations associated with elements of C reduces CRQ. For example, register classes whose elements have a life span of d granules are critical. This occurs for the registers **v1** and **v6** in figure 4. This notion helps to select good candidates for the allocation of a physical register or for spilling.

In the scheduling procedure described above, there is no direct to try to minimize that quantity. However, as we observed in practice, the resulting CRQ is generally not far from the optimal due to the fact that we are scheduling as early as possible, the executable templates, which in fact, directly reduces the lifespan of virtual register names and therefore CRQ. To refine our approach, we are currently testing a level scheduling strategy which instead linearly ordering the templates, allows to choose for execution between several templates. The criteria used for selection is the local minimization of the number of simultaneously alive registers.

4 Register Allocation and Spilling

In this section, we describe how to map the virtual registers onto the physical ones. Depending upon the value of CRQ, relative to the number NPR of physical registers available, two cases occur:

- If CRQ is less or equal than NPR, the specific allocation algorithm described in section 4.2 uses exactly CRQ registers, which is optimal.
- If CRQ is greater than NPR, we study the life of the virtual registers to determine the ones to be spilled, in order to reduce CRQ, as described in section 4.3. This process is iteratively repeated up to the point where CRQ is less than than

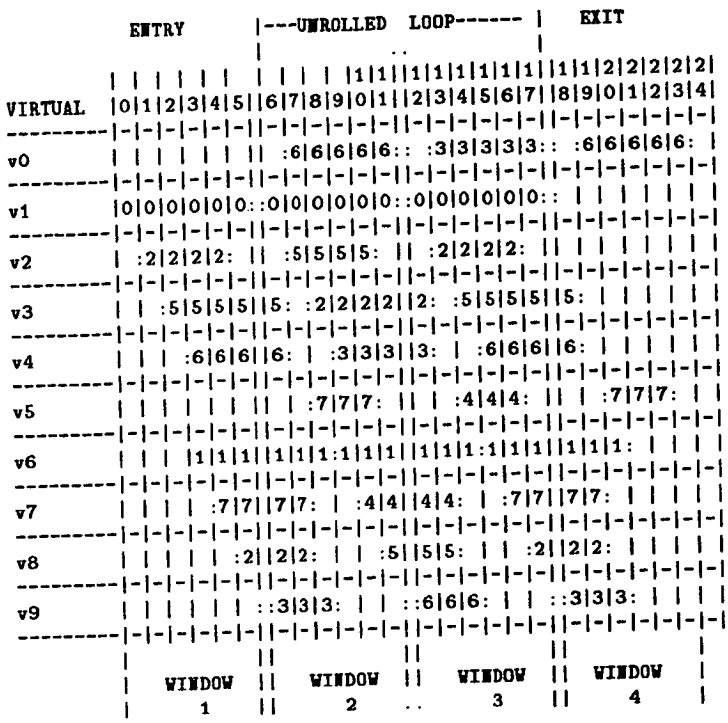

Figure 6: Allocation of Real to Virtual Registers

NPR which is relevant of the previous case. During that phase, spilling code is introduced and scheduled. In many practical cases, the spill will take place in parallel with other operations, without delaying any arithmetic or main memory operation.

4.1 Register Allocation Framework

In that section we assume that CRQ is less or equal than NPR, and show how to produce a physical register assignment requiring CRQ registers at most. The register assignment will be described by a function ϕ associating to each virtual register a physical one in $RV[1..NPR]$.

First, if the loop contained only one iteration, the problem could be easily solved by graph coloring (Cf.[5]). We would first build an interference graph in the following way:

- the vertices are the virtual registers.
- the edges are drawn between all the pair of virtual registers simultaneously alive

A graph of intervals on the real line is thus

obtained, which can be easily colored in polynomial time. (Cf. [3] and [14]).

Now, for a loop with more than one iteration, if we forced each iteration to follow the same assignment rule

$$\phi(R_l(i)) = \phi(R_l(i+1))$$

the same code could be generated for all iterations. This involves building a similar graph:

- the vertices are the virtual register classes R_1 through R_k.
- an edge is drawn between R_i and R_j if and only if there are two pairs $(i,m) \neq (j,n)$ such that $R_i(m)$ and $R_j(n)$ are simultaneously alive.

However, in that case, the resulting graph is not a graph of intervals on the real line anymore but a graph of arcs on a circle of d granules long. This is due to the periodic nature of the loop. The resulting graph may be difficult or even impossible to color: if the life of a virtual register is greater than the period of the loop d, there will be a self cycle, making no coloring possible.

To solve this problem, we must enlarge the possible choices of allocation functions. Specifically, we look for periodic allocation functions of integer period t :

$$\exists t \in \mathcal{N} \quad \phi(R_l(i)) = \phi(R_l(i+t)).$$

When such a couple (ϕ, t) is found, we just unroll the loop t times, and we generate the code for the block of t iterations, spanning td granules, according to the mapping defined by ϕ.

4.2 Unrolling Strategy

As a preliminary step, the critical names are allocated. On the example of figures 4 and 6, v1 is assigned to RV0 and v6 to RV1, leaving 6 critical registers. This has reduced the complexity of the problems without any loss; if s physical registers have been used for holding the s critical names, the critical number of registers required for the remaining names has been decreased by at least the same amount s. Now we can focus on the allocation of the remaining names.

In the sequel, the term window will denote d consecutive granules, and the loop execution is decomposed in a sequence of windows. On the example, the first window consists of granules 0 up to to 5, the second from 6 to 11,... Except for the startup and pipeline flush windows, the execution pattern on these windows is exactly identical. One of them is arbitrarily selected as a reference window, (window 2 in the example).

We will construct,on a window by window basis, a function ϕ satisfying the following constraints:

- first over each window considered separately the function ϕ is an admissible allocation: any pair of virtual registers simultaneouly alive in a cycle of the window will be assigned through ϕ different register names
- between two consecutive windows, the function ϕ preserves the continuity of the assignment: a register name whose life span several windows, is assigned across all the windows the same number

It can be readily checked that such a function, constitutes an admissible assignment for the whole loop.

To satisfy the first constraint, we start the construction by building an assignment $\phi_r$over the reference window. This is a problem which can be solved in polynomial time using exactly CRQ registers. The assignment over the other windows is now defined by a permutation σ of the physical register number:

$$\text{over any window:} \quad \phi = \sigma \circ \phi_r.$$

By construction, the first property is verified. The problem is now to find the proper permutation σ such that the second property also holds.

Let us notice that over the reference window, two different names belonging to the same class may appear such as $v3(i)$ and $v3(i+1)$. During the allocation phase, these names may be allocated to different physical registers.

In our example, as a result of the assignment on the reference window, $v0$ is assigned to $RV6$, $v2$ to $RV5$, $v3$ to $RV5$ then to $RV[2]$, $v4$ to $RV6$ then to $RV3$, $v5$ to $RV7$, $v7$ to $RV7$ and $RV4$, $v8$ to $RV2$ then to $RV5$, and finally $v9$ to $RV3$.

Now, let us determine the proper permutation σ to satisfy the continuity constraint. In fact our problems is that some names have a lifespan of several windows. For sake of simplicity, let us assume that the execution of one iteration does not span on more than two windows. It is convenient to introduce here the set of names (denoted RIGHT), whose life span over the reference window and the next one:

$$RIGHT = \{R_{i_1}(j_1), ...R_{i_s}(j_s)\}$$

Similarly, we define LEFT as the set of names alive in the reference window and in the previous one. Due to the cyclic nature of the loop and our scheduling strategy, there is a one to one mapping between the elements of $LEFT$ and $RIGHT$ associating names in the same class. Therefore:

$$LEFT = \{R_{i_1}(k_1), ...R_{i_s}(k_s)\}$$

In our example:

$$\begin{aligned} RIGHT &= \{v3(2), v4(2), v7(2), v8(2)\} \\ LEFT &= \{v3(1), v4(1), v7(1), v8(1)\} \end{aligned}$$

Looking closer at the physical registers allocated at the element of left and right, we define a partial mapping θ over the physical register numbers by:

$$\begin{aligned} \theta(R_{i_1}(k_1)) &= R_{i_1}(j_1) \\ &..... \\ \theta(R_{i_s}(k_s)) &= R_{i_s}(j_s) \end{aligned}$$

This an injective partial mapping which can be completed into a bijection σ. It is easily checked that such a permutation satifies the continuity property by construction. The order t of σ will give the period t, recalling that

$$\sigma^t = \text{Identity}.$$

In our example,

$$\begin{aligned} \theta(v3(1)) &= \theta(5) = 2 \\ \theta(v4(1)) &= \theta(6) = 3 \\ \theta(v7(1)) &= \theta(7) = 4 \\ \theta(v4(1)) &= \theta(2) = 5 \end{aligned}$$

$$\begin{aligned} \sigma(2) &= 5 \\ \sigma(3) &= 6 \\ \sigma(4) &= 7 \\ \sigma(5) &= 2 \\ \sigma(6) &= 3 \\ \sigma(7) &= 4 \end{aligned}$$

There are many ways to complete θ in order to get σ. For this purpose, we build the partial graph of θ and complete it to limit the length of the cycles. The reason is that the order of σ (and therefore the degree of unrolling) is the least common multiple of the length of the cycles. By the same token, for eight registers of the Cray, the maximal t is 15. In practice, this is a very large upper bound; for all the loops described in the benchmark section,the unroling factor is not greater than 3 .

4.3 Spilling

We present here the simplest of our strategies for spilling, more elaborated ones are under development. Although fairly simple, this technique seems well adapted to the frequent case where CRQ is not too large compared to NPR. It can be assumed that the local memory is large enough to hold all the spilled data, the limit in the local memory of the Cray 2 being of 256 full vector registers.Several problems need to be solved:

- selecting a canditate for spilling
- scheduling of the transfers
- bookeeping in the register data structures of the compiler

The registers are spilled one by one as long as CRQ is greater than NPR. The criteria to select the next register to spill is

- if a register being spilled reduces CRQ, select it immediately.
- otherwise, spill the register with the longuest life.

The move to or from local memory is simply inserted in the reservation table on the line corresponding to the local memory access unit. If there is a conflict on the local memory unit, we try to schedule later or earlier but while preserving the decrease in CRQ. If none of these costless solutions is feasible, a granule of delay is inserted. The bookeeping amounts to mark as killed a register spilled at the first time after it has entered the local memory. When the data is needed again, a new name is created.

The other strategies under development for spilling incorporate spilling directly into the scheduling phase. This amounts to add constraints in the same spirit as "register pressure".

5 Experimental Results

We present here some benchmark results and compare our code generation techniques with the main FORTRAN compilers available on the Cray2. The machine used was a Cray2 using dynamic main memory implying a long latency. Some experiments were performed under a normal load and also some others on dedicated time in order to get an accurate view of the performance. In our tables, CFT77 denotes the Fortran compiler CFT77 2.0, CFT2 denotes the Fortran compiler CFT2 3.0b and "ours" denotes the code generated according to our method. In our case, the vector code was automatically generated using VATIL for the vectorization and dependence analysis and a version of the microcode compacter MIMOSA modified to performed as described in this paper. At the time of the experiments, the address generation, done in scalar, was not fully operationnal, so the address computations were inserted semi-manually. This does not affect the results due to their minor impact on the overall performance.

Our benchmarks (Cf. Figures 8,9,10, 11 and 12) contained vector loops – m mv mvf families – and two larger kernels (Cf. 7) – multiplication of two complex vectors: mulcomp and applying a Givens rotation: drot –. The goal of the m, mv and mvf families is to get a systematic evaluation of the behavior of the different code generators for various possible 2, 3 and 4 adic vector operations; we wanted to check that our method was not sensitive to some patterns of operations. All these loops are very good candidates for the use of chaining mechanism.

As a first conclusion, for our benchmark, CFT77 achieved better performance than CFT2, in particuliar for vector length greater than 256. This is due to the fact that CFT77 unrolls the loops by blocks of 256 iterations, then on block of 256 iterations, it resched-

ules the operations in fashion quite similar to trace scheduling. In both cases our method achieves better performance, and might be of interest for the evolution of commercial compilers, which obviously have to work under more stringent reliability requirements. On most cases, our methods permits to use fully the possible memory bandwidth.

Conclusions

A new approach to vector code generation and vector code scheduling for the Cray-2 has been described. It is built upon ideas from microcode compaction, program restructuring, and an original allocation strategy adapted to the periodicity of the loops. Our present implementation is based on the MIMOSA microcode optimizer, and has permitted to validate the approach as well as test several strategies. The experimental results show the quality of the code produced, and the adequacy of the simple machine model used. The fact that speeds near the peak potential performance of the machines can be reached, also give some insights on the efficiency aspects of the non chaining Cray 2 Architecture. Directions for further research contain comparative studies of spilling strategies, and optimization of the use of local memories.

References

[1] ANDERSON D. W., SPARACIO F. J. AND TOMASULO F. M., "The IBM System/360 Model 91: Machine Philosophy and Instruction-Handling", *IBM J.*, vol 11, January 1967, Reprinted in C.G. BELL, A. NEWELL AND D.P.SIEWIOREK, **"Computer Structures : Principles and Examples"**, *Mc Graw Hill.*

[2] ARYA,S., "Optimal Instruction Scheduling for a Class of Vector Processors: an Integer Programming Approach", *Report CRL-TR-19-83, University of Michigan*, 1983.

[3] BERGE, C., **"Graphes"**, *Gauthier Villars, Paris*, 1970.

[4] CHAILLOUX, J. ,"Le_Lisp de l'INRIA, le Manuel de Reference",*Decembre 1984.*

[5] CHAITIN,G.J., "Register Allocation and Spilling via Graph Coloring", *Proc. ACM SIGPLAN Symp. on Compiler Construction*, Boston, 1982.

[6] CONTROL DATA CORPORATION, **"6600 CENTRAL PROCESSOR Vol 1 & 2"**, *Ref. 020167 & 0911466*, 1967.

[7] CRAY RESEARCH INC., **"Cray 1S Series Hardware Reference Manual"**, *Ref. HR-0808*, 1980.

[8] CRAY RESEARCH INC., "Cray-XMP Computer Systems", *Main Frame Reference, 1980.*

[9] CYTRON, R.G., "Doacross: Beyond Vectorization for Multiprocessors", *Proc. International Conference on Parallel Processing, August 1986.*

[10] CYTRON,R., FERRANTE,J., "What's in a name? The value of re-

naming for parallelism detection and storage allocation", *ICPP '87*, 1987.

[11] DAVIDSON, E.S.,"The Design and Control of Pipelined Function Generators",*Proc. 1971 Int. IEEE Conf. on Syst., Networks, and Computers, Oaxtepec, Mexico, January 1971.*

[12] FISHER, J.A., ELLIS, J.R., RUTTENBERG, J.C., NICOLAU, A., "Parallel processing:a smart compiler and a dumb machine", *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction, June 1984.*

[13] GANAPATHI,M., FISCHER,C.N., "Affix grammar driven code generation ", *ACM-TOPLAS, VOL.7, N0.4, Oct.1985, pp.560-599,* , 1985.

[14] GONDRAND,M., MINOUX,M., **"Graphes et Algorithmes"**, *Eyrolles, Paris*, 1985.

[15] GOODMAN, J.R., YOUNG, H.C., "Code Scheduling Methods for Some Architectural Features in PIPE", *CSTR 579, University of Wisconsin-Madison.*

[16] HANEN, C., "Optimizing Static Microprogrammable Pipelines: a Timed Petri Net Model", Proc of the 2nd International Conference on Supercomputing, Santa Clara, 1987.

[17] HOCKNEY, R.W., JESSHOPE, C.R., "Parallel Computers ", *Adam Hilger, Bristol, 1981*

[18] KENNEDY, K., "Automatic Vectorization of Fortran Programs to Vector Form", *Technical Report, Rice University, Houston, TX, October 1980.*

[19] KUCK, D.J., KUHN, R., PADUA, D., LEASURE, B., WOLFE, M., "Dependence Graphs and Compiler Optimizations", *Proc. 8th ACM Symp. POPL, Williamsburgh, VA, 1981.*

[20] KOGGE, P.M., "The Architecture of Pipelined Processors", *McGraw-Hill, New York 1981.*

[21] LANDSKOV, D., DAVIDSON, S., SHRIVER, B., MALLETT, P.W., "Local Microcode Compaction Techniques", *Computing surveys, vol. 12, n 3, September 1980.*

[22] LICHNEWSKY, A., LOYER, M., "Un Module Vectoriel Flottant sur SPS7. Pourquoi?", *Bulletin de Liaison de Liaison de la Recherche en Informatique et en Automatique, no 112, 1987.*

[23] LICHNEWSKY,A., THOMASSET, F., "Techniques de base pour l'exploitation automatique du parallelisme dans les programmes", *Rapport de Recherche INRIA, N 460, 1985.*

[24] LICHNEWSKY, A., THOMASSET, F., EISENBEIS, C., "Automatic Detection of Parallelism in Scientific Programs with Application to Array-Processors", *Proc.of IBM Institute, North Holland, 1986.*

[25] MACKE, T., HUSON, C., DAVIES, J., LEASURE, B., WOLFE, M., "The KAP/ST-100 : A Fortran Translator for the ST100 Attached Processor", *International Conference on Parallel Processing 1986.*

[26] NEC CORPORATION, **"NEC Supercomputer SX-1/SX-2. General Description"**, *GAZ01E*, 1983.

[27] NICOLAU, A., "Uniform Parallelism Exploitation in Ordinary Programs", *International Conference on Parallel Computing 1985.*

[28] RAU, B.R., GLAESER, C.D., PICARD, R.L., "Efficient Code Generation for Horizontal Architectures: Compiler Techniques and Architectural Support", *Computer Architecture News, Vol 10, No 3, April 1982.*

[29] SHAR, L.E., "Design and scheduling of statically configured pipelines", *Stanford Univ., Technical Report n 42, 1972.*

[30] WEISS,S., SMITH,J.E., "Instruction Issue Logic in Pipelined Supercomputers", *IEEE Tr. Comp., Vol. C-33, N0.11, Nov. 1984, pp 1013-1022*, 1984.

[31] STAR TECHNOLOGIES, INC. "ST100: The 100 Megaflops array processor", *Hardware and Software Reference Manuals.*

[32] TOMASULO,R.M., "An Efficient Algorithm for Exploiting Multiple Arithmetic Units", *IBM Journal*, 1967.

[33] TOUZEAU, R.F., "A Fortran Compiler for the FPS-164 Scientific Computer", *SIGPLAN Notices, Vol. 19, N 6, June 1984.*

m1	y(i) = (a * x(i)) + b
m2	y(i) = (a + x(i)) * b
mv1	z(i) = ((x(i) * y (i)) + a) * b
mv2	z(i) = ((x(i) * b + a) * y(i))
mv3	z(i) = (y(i) + (a * x(i))) + b
mv4	z(i) = (a * x(i)) + (y(i) + b)
mv5	z(i) = ((a * x(i)) + b) + y(i)
mv6	z(i) = ((x(i) + b) * a) + y(i)
mv7	z(i) = ((x(i) + y(i)) * a) + b
mv8	z(i) = ((x(i) + y(i)) + a) * b
mv9	z(i) = (((x(i) + a) + y(i))) * b
mvf1	t(i) = ((x(i) * a) + y(i)) * z(i)
mvf2	t(i) = ((x(i) + y(i)) * z(i)) + a
mvf3	t(i) = (x(i) + (y(i) * z(i))) * a
mvf4	t(i) = x(i) + ((y(i) + a) * z(i))
mvf5	t(i) = (x(i) + y(i)) + (a * z(i))
mvf6	t(i) = (x(i) + (y(i) * a)) + z(i)
mvf7	t(i) = ((x(i) * y(i)) + a) * z(i)
mvf8	t(i) = ((x(i) + a) * y(i)) * z(i)
mulcomp	zr(i) = xr(i) * yr(i) - xi(i) * yi(i) zi(i) = xr(i) * yi(i) + xi(i) * yr(i)
mulcomp	x(i) = c * z(i) - s * t(i) y(i) = s * z(i) + c * t(i)

Figure 7: Program Kernels used in the Experiments

Performance for V. Length = 234				
Loop	CFT 2	CFT 77	Ours	Speedup
m1	62.92	71.93	102.07	1.42
m2	62.11	57.23	100.12	1.61
mv1	67.39	70.81	109.54	1.55
mv2	78.54	80.84	122.47	1.51
mv3	79.06	87.21	116.79	1.34
mv4	75.68	82.88	121.34	1.46
mv5	82.39	81.82	122.04	1.48
mv6	79.04	80.26	119.06	1.48
mv7	67.79	71.50	88.160	1.23
mv8	67.87	73.03	106.08	1.45
mv9	75.85	82.71	117.67	1.42
mvf1	71.25	70.81	107.21	1.50
mvf2	67.02	72.10	99.250	1.38
mvf3	64.80	71.08	100.77	1.42
mvf4	72.80	74.27	103.08	1.39
mvf5	62.61	69.37	102.34	1.48
mvf6	70.27	80.37	109.33	1.36
mvf7	68.06	74.49	96.570	1.30
mvf8	70.61	75.39	113.39	1.50
mulcomp	98.52	108.5	128.44	1.18
drot	127.6	121.3	148.95	1.17

Figure 8: Performance in MFLOPS under Average Load

Performance for V. Length = 490				
Loop	CFT 2	CFT 77	Ours	Speedup
m1	67.99	90.73	113.22	1.24
m2	66.62	93.03	108.89	1.17
mv1	70.87	89.99	123.25	1.37
mv2	84.64	106.93	137.90	1.29
mv3	78.95	111.57	146.94	1.32
mv4	78.36	99.43	145.51	1.46
mv5	85.57	106.98	149.32	1.39
mv6	82.34	103.86	135.86	1.31
mv7	72.86	84.05	115.36	1.37
mv8	72.30	87.45	127.86	1.46
mv9	81.33	104.82	133.08	1.27
mvf1	74.72	92.69	109.81	1.18
mvf2	68.01	83.29	117.40	1.41
mvf3	64.65	87.08	97.48	1.12
mvf4	79.78	88.05	92.74	1.05
mvf5	64.73	85.15	111.24	1.31
mvf6	78.69	89.40	120.23	1.34
mvf7	71.64	87.07	117.13	1.34
mvf8	72.75	90.79	119.03	1.31
mulcomp	104.71	111.37	152.12	1.36
drot	130.35	123.94	130.35	1.05

Figure 9: Performance in MFLOPS under Average Load

Performance for V. Length = 256			
Loop	CFT 77	Ours	Speedup
m1	78.09	121.24	1.55
m2	75.87	129.40	1.70
mv1	83.41	119.61	1.43
mv2	100.48	143.75	1.43
mv3	96.82	144.75	1.49
mv4	96.96	145.20	1.50
mv5	95.32	144.75	1.52
mv6	86.74	143.75	1.66
mv7	80.14	105.11	1.31
mv8	80.74	130.80	1.62
mv9	95.80	144.87	1.51
mvf1	87.62	115.77	1.32
mvf2	84.67	111.36	1.32
mvf3	83.17	114.21	1.37
mvf4	77.50	115.69	1.49
mvf5	84.09	113.45	1.35
mvf6	98.26	110.32	1.12
mvf7	84.17	111.23	1.32
mvf8	93.34	123.56	1.32
mulcomp	128.84	163.66	1.27
drot144	150.11	165.62	1.10

Figure 10: Performance in MFLOPS on dedicated machine

Performance for V. Length = 480			
Loop	CFT 77	Ours	Speedup
m1	105.23	126.84	1.20
m2	100.86	131.46	1.30
mv1	102.95	125.43	1.22
mv2	121.38	151.97	1.25
mv3	116.98	152.97	1.31
mv4	119.32	153.63	1.29
mv5	111.83	147.63	1.32
mv6	99.96	150.54	1.50
mv7	91.94	114.59	1.24
mv8	96.58	132.73	1.37
mv9	112.77	150.15	1.33
mvf1	100.06	119.17	1.19
mvf2	101.02	116.68	1.15
mvf3	101.77	121.23	1.19
mvf4	93.67	116.10	1.23
mvf5	99.32	118.53	1.19
mvf6	107.07	115.12	1.07
mvf7	100.13	116.68	1.16
mvf8	101.54	125.66	1.24
mulcomp	127.28	171.20	1.34
drot	148.46	203.42	1.37

Figure 11: Performance in MFLOPS in Single User

Performance for V. Length = 960			
Loop	CFT 77	Ours	Speedup
m1	131.787	156.202	1.18
m2	122.260	159.663	1.30
mv1	121.277	139.789	1.15
mv2	142.381	178.148	1.25
mv3	135.306	179.560	1.32
mv4	139.608	178.738	1.28
mv5	136.248	167.727	1.23
mv6	112.518	165.983	1.47
mv7	112.254	146.464	1.30
mv8	118.987	153.976	1.29
mv9	128.110	174.086	1.35
mvf1	107.678	125.998	1.17
mvf2	115.552	131.174	1.13
mvf3	118.210	138.521	1.17
mvf4	106.112	122.504	1.15
mvf5	111.679	132.461	1.18
mvf6	116.838	125.12	1.07
mvf7	115.461	131.101	1.13
mvf8	108.584	135.921	1.25
mulcomp	129.429	192.898	1.49
drot	151.892	234.655	1.54

Figure 12: Performance in MFLOPS in Single User

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique