A New class of algorithms for software pipelining with resource constraints

Christine Eisenbeis, D. Windheiser

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°2033

Programme 2

Calcul symbolique, Programmation
et Génie logiciel

A NEW CLASS OF ALGORITHMS FOR SOFTWARE PIPELINING WITH RESOURCE CONSTRAINTS

Christine EISENBEIS
Daniel WINDHEISER

Septembre 1993

A NEW CLASS OF ALGORITHMS FOR SOFTWARE PIPELINING WITH RESOURCE CONSTRAINTS *†

UNE NOUVELLE CLASSE D'ALGORITHMES POUR LE PIPELINE LOGICIEL AVEC CONTRAINTES DE RESSOURCE

Christine Eisenbeis ‡
INRIA Rocquencourt
Domaine de Voluceau, BP 105
78153 Le Chesnay Cedex
FRANCE

Daniel Windheiser §
The University of Michigan
2312 EECS building
1301 Beal Avenue
Ann Arbor, MI 48109-2122
ETATS-UNIS

*This work was partially supported by ESPRIT Project COMPARE.

†A concise version of this report ("Optimal Software Pipelining in Presence of Resource Constraints") appeared in the Proceedings of Parallel Computing Technologies (PaCT-93), August 31-September 3, 1993, Obninsk, Russia.

‡e-mail:Christine.Eisenbeis@inria.fr
§e-mail:windheis@suraj.eecs.umich.edu

Abstract

This report presents a new class of algorithms for loop software pipelining in the presence of resource constraints. This new approach allows to generate optimal code with respect to throughput even for processors with complex resource constraints which make them hard to program even by hand. It consists of two steps: first, we build the reservation table for the body of the software pipelined loop by packing the elementary reservation tables of the different tasks as if the tasks were independent, the length of the global reservation tables then determines the *throughput* of the loop; second we determine the schedule of each iteration of the loop so that all data dependencies are satisfied and the resource usage conforms with the previously computed reservation table.

We show that, in order to reach the optimal throughput (induced by the highest used resource or *critical* resource), the loop may need to be unrolled. In some important and general cases, we provide theoretical results on the unwinding degree necessary to achieve the optimal throughput. We also show how this class of software pipelining algorithms can be used to generate code for the explicitly advanced pipelines found in the Intel i860 processor.

Keywords: code scheduling, software pipelining, instruction-level parallelism, i860 processor, explictly advanced pipeline

Résumé

Ce rapport propose une nouvelle classe d'algorithmes pour le pipeline logiciel des boucles avec contraintes de ressource. Cette nouvelle approche nous permet de générer du code à débit optimal pour des processeurs à contraintes de ressource très complexes qui les rendent difficiles à programmer même à la main. L'algorithme consiste en deux étapes : d'abord, nous construisons une table de réservation globale pour la boucle, comme si les opérations étaient indépendantes, en compactant les tables de réservation élémentaires correspondant aux opérations de la boucle. La longueur de cette table de réservation détermine le débit de la boucle. Ensuite, nous rattachons des indices aux opérations ainsi ordonnancées de telle sorte que les contraintes liées aux dépendances de données soient vérifiées.

Nous montrons que pour atteindre un débit optimal (avec saturation de la ressource la plus utilisée), il peut être nécessaire de dérouler la boucle et donnons quelques résultats théoriques sur les degrés de déroulage qui peuvent atteindre ce débit. De plus, nous montrons comment on peut utiliser ces algorithmes pour générer du code pour les pipelines à avance explicite du processeur i860 d'Intel.

Mots-Clés : ordonnancement de code, pipeline logiciel, parallélisme entre instructions, i860, pipeline à avance explicite

Contents

1 Introduction

The latest high-performance processors make heavy use of internal fine grain parallelism for achieving ever higher performance. There are basically two ways for increasing the number of concurrent activities taking place in a processor: first increase the number of independent functional units,as exemplified by *superscalar architectures*, second increase the depth of the pipelines functional units as exemplified by *superpipelined architectures*. The responsibility of managing the fine grain parallelism is either supported by the hardware (i.e. at run-time) or by the compiler or by a combination of the two. In order to simplify the control logic and thus reduce the clock cycle, VLIW and the original RISC architectures rely entirely on the compiler to manage the functional units. The current trend in high performance microprocessors is to introduce some hardware support for dynamically scheduling the instructions in order to reduce the burden of the compiler and maintain upward compatibility when faster implementations are made possible by the progress of technology. This trend does not mean that compile-time code optimizations (instruction selection and scheduling, register allocation) become useless: as a matter of fact compile-time optimization potentially leads to much better speedup than dynamic scheduling because it benefits from a global vision of the code.

Generating code for optimal performance is known to be a difficult problem.

Consider the loop of figure 1.

```
For i = 1 to N do
      r1 <-- A(i)
      r2 <-- B(i)
      r3 <-- r1+r2
      C(i) <-- r3
Endfor
```

Figure 1: Original Loop

We assume that the memory access unit and the adder are pipelined, with, say, m stages and a stages respectively. Throughout the paper, we will use the framework of reservation table both for specifying the resource constraints associated to each type of instructions and for representing the execution of the whole loop. A reservation table represents the occupancy of the machine resources versus the execution time in a two dimensional table, where the horizontal coordinate is time counted in clock cycles and the vertical coordinate describes the functional units. Within this framework, the execution of the previous loop is shown on figure 2. It can easily be seen that the total execution time of the loop is equal to $N(a+m+2)+m-1$ clock cycles. The corresponding throughput is thus asymptotically equal to 1 iteration every $a+m+2$ cycles.

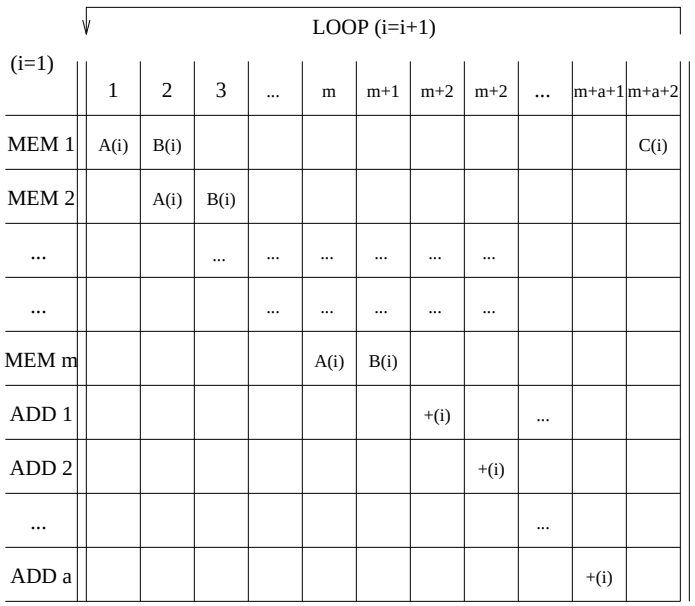

Figure 2: Simple loop execution

As the clock cycle tends to shrink with the advances of technology, the number of stages in the pipelines tends to become larger and larger, which results in the performance of a simple loop schedule (like the one shown above) to decrease in proportion. As a consequence, new algorithms for loop optimization have been developed. Those algorithms can be classified into two large classes:

1. **(Finite) unrolling algorithms.**
   Several iterations are grouped together so as to enlarge the loop body and to exploit the parallelism between instructions belonging to different iterations. Unfortunately with this kind of methods, the degree of unrolling is difficult to determine beforehand because it depends on many parameters: the structure of the loop computation and the target architecture of course, but mainly on the local optimization algorithm that is performed. However those algorithms are easy to implement since they only require an algorithm for unrolling and an algorithm for optimizing the code locally. For instance, if we unroll the previous loop 4 times, we obtain the reservation table in figure 3 which results in an asymptotic throughput equal to four iterations every $m + a + 8$ clock cycles (provided that $m + a + 1$ is greater than 8, otherwise the 8 memory loads could not have been compacted together).

2. **Software Pipelining algorithms.**
   These algorithms completely modify the structure of the loop in order to unveil the maximum amount of parallelism between instructions belonging to different iterations.

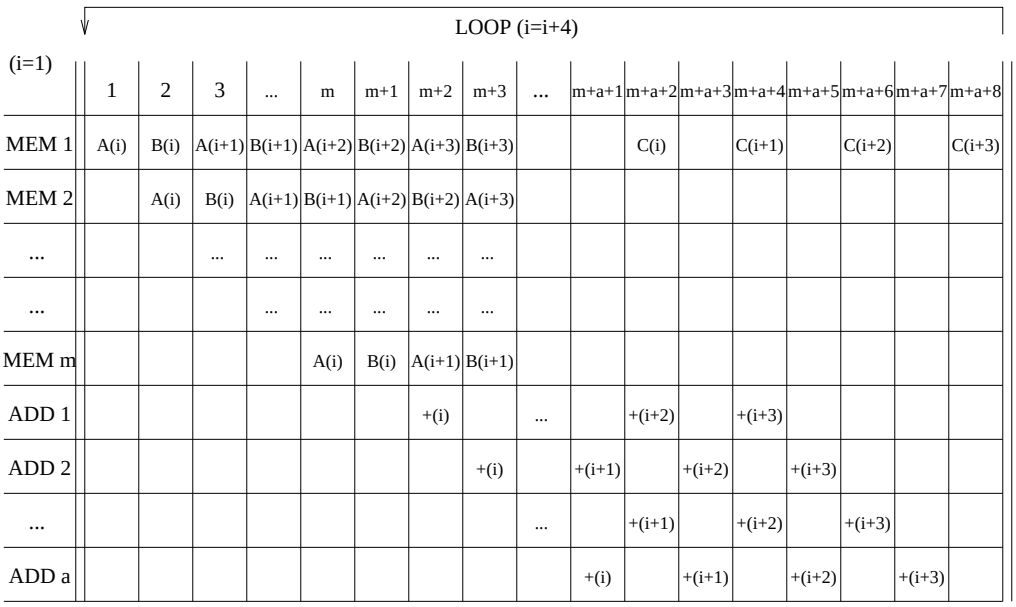

Figure 3: 4-times unrolled loop

The resulting code typically consists of a prelude, a loop mixing different original operations and a postlude, similar to the principle of hardware pipeline, hence the denomination. Software pipelining methods can be further divided into methods emulating "infinite unrolling" and methods based on "modulo resource constraints". Infinite unrolling methods unroll and compact the loop until a repetitive pattern is found ([AN88, Aik88] for the case when there are no resource constraints and [Bod89] for the case with resource constraints).

On figure 4, m is set to 5 and a to 4; previous loop has to be unrolled 20 times before discovering a repetitive pattern (from clock cycle 21 to clock cycle 38 in the reservation table). This pattern includes 6 iterations and the resulting throughput is equal to 6 iterations every 18 clock cycles. This throughput is optimal since memory access is saturated in the steady state.

"Modulo resource constraints" software pipelining (also called Cyclic Scheduling)[Cha81, RG81, Ebc87, Lam87, Eis88, BCW89] consists in building directly the pattern of the software pipelined loop by enforcing "modulo" resource constraints (modulo the size of the pattern, called the *initiation interval*). For instance, the optimal throughput for the previous loop is one iteration every 3 clock cycles, since there are 3 memory access per iteration and only one memory port. Therefore, the initiation interval is set to 3 and a reservation table is sought such that any instruction can be initiated every 3 clock cycles. We obtain the reservation table shown in figure 5. These methods are

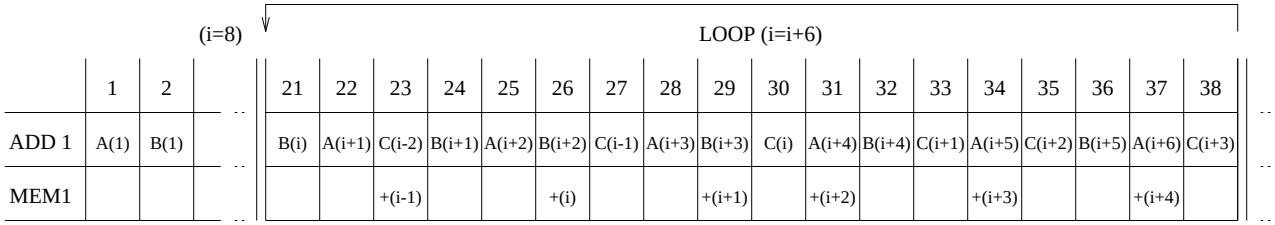

Figure 4: "Infinite-times" unrolled loop and resulting repetitive pattern

also called "modulo scheduling", "cyclic scheduling" or "polycyclic algorithms".

Figure 6 shows the performance of the codes generated by the four strategies described above. It should be noted that these performances have been computed directly from the reservation tables, by assuming a synchronous model and taking into account "nop" operations. However, almost all existing processors provide a powerful hardware interlock mechanism, that allows not to specify "nop" operations in the instruction stream. Therefore, in a real execution, the code corresponding to the cyclic schedule would have a far better performance. Moreover, this code is much simpler than the one obtained with the finite or infinite unrolling strategy. As a matter of fact, with the cyclic schedule fewer special cases need to be considered than for unrolled schedule. Finally, the size of the loop body is usually much larger when using the unrolling strategy. This may result in a high instruction cache miss rate when the code of the loop body does not fit in the instruction cache. As far as the complexity of the optimal strategy is concerned, both methods based on infinite unrolling and on cyclic scheduling lead to exponential algorithms, although experiments show that there are tractable in practice.

The two classes of algorithms have a major drawback in that nothing can be planned beforehand (let us try that and look whether it works). In the case of unrolling algorithms, the unknown is the unrolling degree, whereas for cyclic scheduling, it is the initiation interval. Moreover, to our knowledge, an optimality result could be obtained only for the case of time-unit operations [Hsu86, RG81].

The purpose of this paper is to show that in some cases, a new technique of software pipelining can lead to optimal results in more general cases. This new technique applies to loops with an acyclic dependence graph and consists in decoupling the problem of loop scheduling into two phases: the first step consists in building the loop pattern by "forgetting" about the tasks dependencies. The second step consists in attaching to each task an iteration number called the iteration index so that all dependencies are satisfied. Other recent new loop scheduling algorithms are also based on this idea [Mun91, GS91, WE93]

The interest of our method lies in the fact that it allows us to generate throughput optimal code in some important cases. Moreover it also allows us to take into account

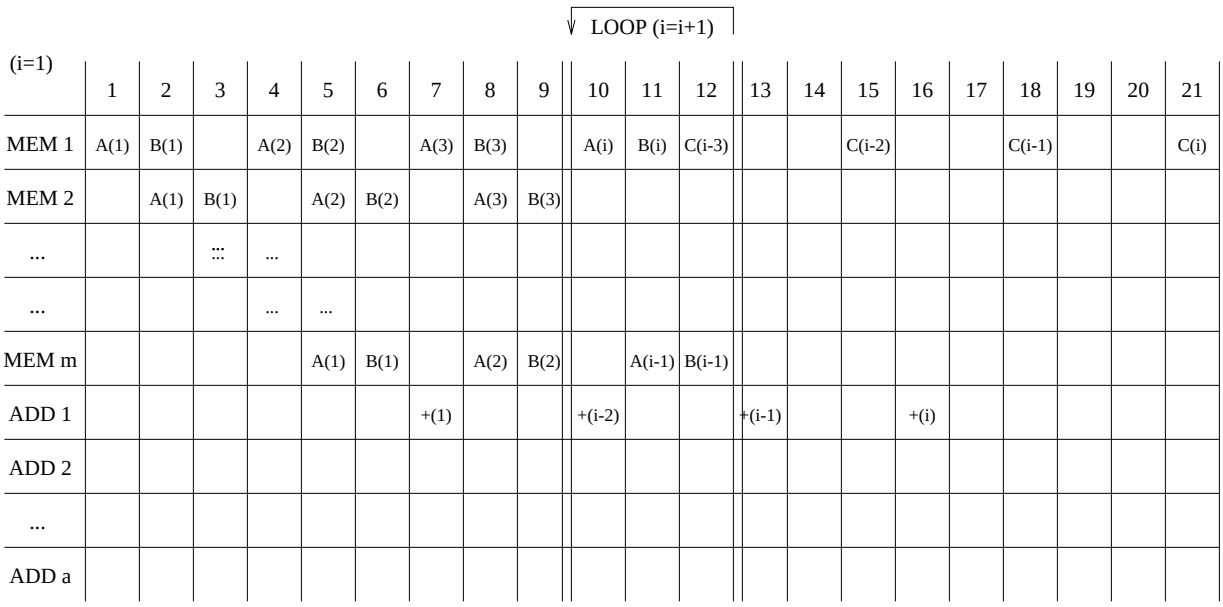

Figure 5: Cyclic scheduling or "Modulo resource contraints" software pipelining

another optimization criteria: **the loop unwinding degree necessary for achieving throughput optimal code**. Indeed, it may prove necessary to unwind the loop in order to get throughput optimal code, whatever the scheduling method. Let us make this clear on an exemple. We consider the previous loop again, but we now assume that two memory ports are available. Now although memory access is still the resource bottleneck, the optimal throughput achievable is one iteration every "3/2" clock cycles, since we can execute two memory accesses per clock cycle. If the loop is not unrolled (i.e. each occurence of any given task executes on the same functional unit whatever the iteration), then the best throughput achievable is one iteration every $\lceil 3/2 \rceil$ clock cycles. However by unwinding the loop a priori two times, the best throughput achievable is 2 iterations every $\lceil 2 \times 3/2 \rceil$ clock cycles, i.e one iteration every "3/2" clock cycles in average, which is the optimal throughput.

Unwinding may be necessary for other reasons than resource allocation: it is well known that performing register allocation after loop scheduling may require loop unwinding [Lam87, EJL90]. If we assume that the previous loop must be unrolled 3 times for register allocation, then, without any further attention, the overall unwinding degree would be $lcm(2,3) = 6$[1] for explicitely describing all the possible cases of imbrications between the 2-periodic schedule and the 3-periodic register allocation. If register allocation could also be carried on the loop unwound 4 times, the overall degree would become $lcm(2,4) = 4$. Or, conversely, if an optimal schedule could be found with the loop unrolled 3 times (this is not the case here),

[1] *lcm* stands for "least common multiple"

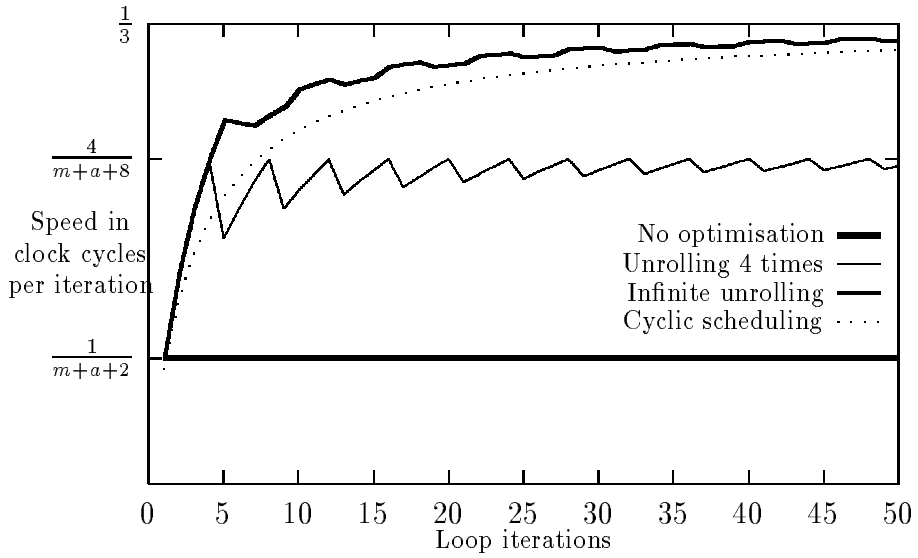

Figure 6: Performance of different scheduling strategies

the overall unwinding degree would be equal to $lcm(3,3) = 3$. Therefore in order to find the smallest unwinding degree, one should not simply minimize the unwinding degree for the resource allocation and register allocation. A better strategy is to be able to characterize the valid unwinding degrees in every pass. This is the object of the section 3, as far as static resource allocation is concerned. In section 3, we consider the optimistic case where all the tasks in the loop body are independent, while in section 4, it is shown how to alleviate this restriction.

The paper is organized as follows. In the first section, the problem is stated and we show why cyclic scheduling methods not always succed in generating throughput optimal schedules, even in simple cases where optimal schedules are trivial. Therefore we introduce a new class of software pipelining algorithms. Next, we consider the case of independent tasks and derive criteria for characterizing unwinding degrees leading to optimal resource allocation. The third section shows how to deal with tasks which are not independent: given a resource allocation planning, we show how to attach an iteration index to each task, so that the semantic data dependencies are satisfied. We give the conditions under which an iteration indexing function exists and the criteria which should be used for choosing a solution. In the last section, examples of applications are given. In particular, we show that it is important to have a good machine model for being able to optimize critical resource, more than not critical ones.

2 Problem statement and Cyclic scheduling (modulo resource constraints)

In this section, the loop scheduling problem and the target architecture model are presented. Then we explain why a simple cyclic scheduling strategy may not lead to optimal results as soon as sophisticated architectural features are considered (actually different task durations on the same resource).

2.1 Semantical scheduling constraints

Throughout this paper, we consider a set of generic tasks T_1, T_2, ..., T_n representing the instructions of one iteration of the loop. The occurrence of task T_p in the i^{th} loop iteration will be noted T_p^i.

For ensuring the loop is executed in a semantically correct way, the tasks are constrained to respect some execution order. This is represented through a **dependency graph**. To be exact, this graph should be composed of every task T_p^i and every dependency concerning two of these tasks. Since this could result in a quite large graph (of order $m \times N$, where m is the number of generic tasks and N is the length of the loop) or even a graph with unknown size if N is unknown, one is used to sum it up into a graph G consisting of the m generic tasks and where one dependency between task T_p and T_q is assumed as soon that there are (eventually) two iterations i and j such that T_q^j depends on T_p^i. To be more precise, edges in the dependency graph are given two valuations, v_{pq} and h_{pq}, which means that $T_q^{i+h_{pq}}$ must be issued at least v_{pq} time units after T_p^i for each iteration i.

Recent work has shown that general loop scheduling problem is NP-hard [Han90], also in the very simple case when the graph is one cycle, with one processor, and the tasks are time-unit. Therefore, an important hypothesis that we do throughout this paper is that the dependency graph G is **acyclic** (this does not exclude loops with dependencies between different iterations).

2.2 Architectural scheduling constraints

In our model, we assume that each task uses the machine resources according to a fixed predetermined plan, specified by a reservation table. The reservation table describes the resource usage as well as the timing. This allows us to account for a large variety of architectures ranging from uniprocessors with several functional units to tightly-coupled multiprocessors.

Homogeneous tightly-coupled multiprocessors are modeled as a set of identical processors, each task has a predetermined execution time but may execute on any processor belonging to this set. In the case of heterogeneous multiprocessors, there are several types of processors which corresponds to different sets.

Concerning uniprocessors, the resources consist of the different functional units (ALU, issue, floating-point units) available in the processor. In the case of pipelined functional units, each stage of the pipeline is a resource by itself since each stage is likely to incur a resource conflict. It should be noted that in some cases only the first stage is required to find out the conflicts. In that case, the duration of occupancy of first stage will be noted as

$d(T)$ for a task T, whereas effective execution time will be modeled through the way of the valuation v defined just above for verifying semantical constraints.

A major feature in high-performance processor with respect to scheduling is its issue capability, i.e. how many and which type of instructions can be simultaneously issued at each cycle. This can be modeled by using dummy resources (i.e. with no physical existence) to account for the constraints on the issue.

2.3 Optimization problem

For being able to represent a schedule by a program, we are looking for periodic schedules, i.e. integer functions that verify:

$$\Omega(T_q^i) = \Omega(T_q^{i-1}) + L$$

The integer L is the *latency*, sometimes referred as II for "initiation interval" in the literature. The resulting average throughput of that schedule is equal to $1/L$ (one iteration every L time units), therefore the least L, the best the performance. A periodic schedule is entirely determined by its value on each task of one iteration (the first for instance) and the latency L.

In the following, we will be conducted to unroll the loop for obtaining better performances. This amounts to extend the previous notion of periodicity to "u-periodicity": a schedule Ω is said "u-periodic" if:

$$\Omega(T_q^i) = \Omega(T_q^{i-u}) + L_u$$

The resulting average throughput is equal to u/L_u. A u-periodic schedule is entirely determined by its value on each task of u consecutive iterations and the latency L_u.

2.4 Principle of Modulo Resource Constraints algorithm

Modulo Resource Constraints algorithm is a general and practical method to perform software pipelining. A latency L is given and one tries to find a schedule for one iteration such that using the same pattern for every iteration and initiating every iteration in turn every L time units does not cause any resource conflicts, and preserve semantic precedence constraints. An example has been given in figure 5.

The method consists in using the well-known **list scheduling** algorithm (see for instance [LDS80] for an application of list scheduling for microprogram compaction) for scheduling the tasks of one iteration, and preventing resource conflicts due to iterating every L time units by introducing new resource constraints: two different generic tasks can not use the same resource at clocks distant by a multiple of L time units (hence the term "modulo resource constraints").

When one fails to schedule code for this given latency L, then one starts again with a greater value for L, and so on, until a schedule is found or L reaches the length of the code for one iteration (this is computed as a first pass in the algorithm).

A great advantage of this method is that it can a priori be applied to any kind of code, since it relies simply on one local scheduling algorithm. Actually, Lam in [Lam87]

was able to obtain software pipelining also for loops with cyclic graphs and for loops with if statement or even with branches outside the loop, by the mean of a so-called "hierarchical reduction". This technique was originally used for microprogramming on array-processors [Cha81, Tou84, Eis88, BCW89]. It is the basic idea of polycyclic architecture [RG81]. It was further developed and adapted for vector architecture [EJL88, TDT88] and also for more general programs [Ebc87, Lam87].

The large complexity of this algorithm comes from the great uncertainty on the value L for which a schedule will be found. (This is not monotonic, in the sense that it can happen that a schedule is found for L and not for $L+1$). This makes the algorithm costly since every successive value for L must be tried.

Another inconvenient is closely related to the underlying local scheduling algorithm (list scheduling), that tends to schedule the tasks as soon as possible. For real loop scheduling, this promote to charge the registers from memory very early and increases the register pressure: when the tasks to be scheduled come from one arithmetical expression, the graph has a structure of tree and it is useless that the task near the root are scheduled too early. This can be alleviated by performing the list scheduling in the inverse order (scheduling the tasks as later as possible).

Another difficulty of that method is related to code generation and results from the restructuring of inner loop (this is not specific to Modulo Resource Constraints algorithm but to software pipelining methods in general). Since generated inner loop is no more composed of tasks of the same iteration, like in original code, but of tasks of different iterations, branch code put be added and special cases on the number of iterations must be treated apart. For avoiding additional code, a special hardware mechanism has been proposed by [RG81].

2.5 Limitations

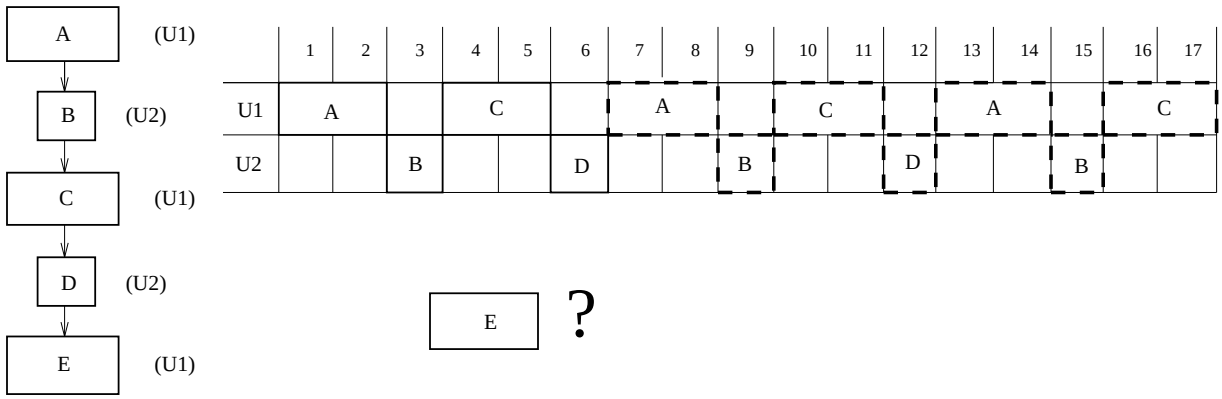

Figure 7: Example

A counterpart of the generality of this algorithm is that even when the machine archi-

tecture is very simple, it can fail to find an obvious schedule for a given L. Let us explain this on example of figure 7: when we apply the cyclic local scheduling algorithm to the given graph, with latency equal to 6, it appears that E-task cannot be scheduled, so that optimal performance is not achieved by this method (notice that the same phenomenon would still have appeared in the case of a reverse order scheduling). However, it is clear that there exists one 6-latency local schedule, drawn on diagram of figure 8.

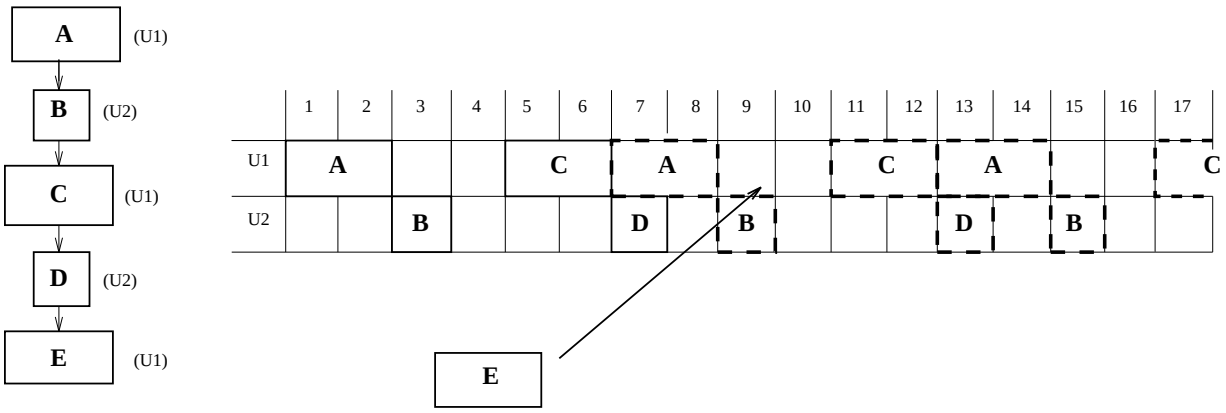

Figure 8: Example

2.6 Some solutions

On the previous example, one of the problems is that the tasks have different durations; for inserting the E-task, it is necessary to change the schedule of a previously scheduled task (C here) and forces the algorithm to backtrack.

In some cases, some tricks can be used to pass round this difficulty. A trivial case is when the tasks take the same time, say T clock cycles, to execute. Then the solution for finding the optimal schedule is to enforce the tasks to be issued only at clock cycles that are a multiple of T. This method can be extended to the case when all the tasks on a same functional unit have the same duration, but not all the tasks in the iteration, see figure 9.

The key point here is the following: the primary concern is to planify how the tasks of different iterations are mixed, **independently** of how they are related in the graph. On the basis of this idea, we split the problem into two parts. First we will forget about the dependencies between the tasks and consider them as independent. Second we assume we are given a pattern of tasks and examine what are the conditions for this pattern to correspond to one possible execution of the loop. Treating these two parts independently allows us a better understanding and control of some quality criteria of the schedule. In the first pass we will try to get a pattern as compact as possible. It may happen that the optimum can be obtained only by unwinding the loop; the criteria to minimize will the number of times

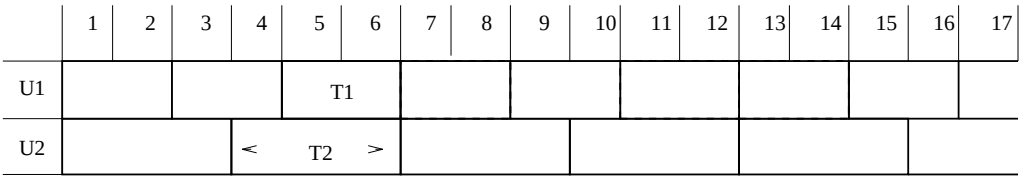

Figure 9: Tasks durations on U_1 (resp. U_2) are T_1 (resp. T_2)

the loop has to be unrolled. In the second pass we will try to minimize the makespan of one iteration (hence the register pressure since these two criteria are closely related).

3 Minimizing the loop unrolling when tasks are independent

In this section, we assume that the tasks are **independent**. Therefore, finding a schedule for the loop tasks amounts to finding a compact pattern of the tasks as short as possible. We will see in the first part of this section that unwinding may be necessary for obtaining a schedule saturating one resource when there is more than one exemplary of that resource (more than one memory port for instance) and a general solution will be presented. For this solution the unwinding degree may be quite large, resulting in a code that may not fit in the instruction buffer/cache and degrade the expected performance. Therefore in the rest of this section, the problem of minimizing (or at least understanding the role of) unwinding degree will be tackled. Many parameters will be taken into account for formalizing this problem. For the architecture counterpart: the number of different types of functional units, the number of available samples of each type of functional unit. For the tasks counterpart: the type of processor it must run on and the time it takes. We will first present simple cases and complicate it more and more to come up to the general case.

3.1 The general case and a simple solution

In the general case, we assume that the machine is composed of k_1 functional units of type P_1, k_2 of type P_2, ..., k_m of type P_m. If we do not allow the loop to be unwound, then optimization problem consists in finding a compact pattern so that two tasks sharing the same type of processor are allowed to overlap only is they are assigned to different functional units. Therefore, this problem looks like the *bin packing problem*, which is well-known to be NP-hard. Let us recall this optimization problem [GJ79]:

> Given a finite set $U = u_1, u_2, \ldots u_n$ of "items" and a rational "size" $s(u) \in [0, 1]$ for each item $u \in U$, find a partition of U into disjoint subsets $U_1, U_2, \ldots U_k$ such

that the sum of the sizes of the items in each U_i is no more than 1 and such that k is as small as possible (**minimize** k, figure 10).

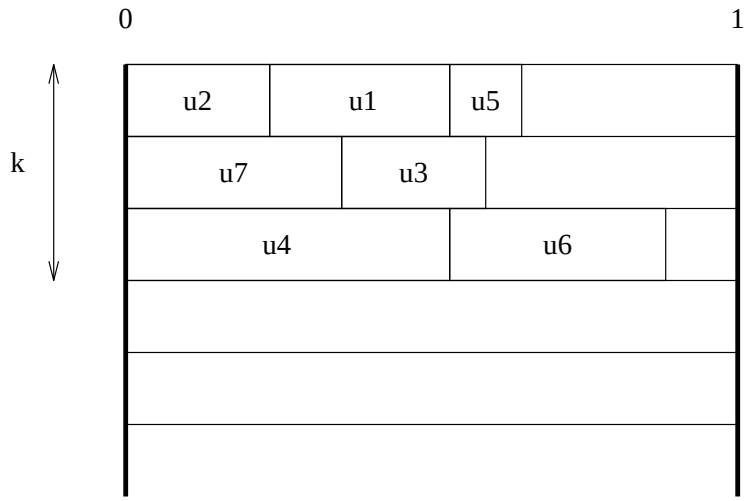

Figure 10: Minimizing the number k of boxes

Our problem is the following:

> Find the least integer L such that for each type of processor j, there exists a partition of tasks of type j into disjoint subsets $U_j^1, U_j^2, \ldots, U_j^q$ such that q is not greater than k_j and the sum of tasks in any set U_j^p is not greater than L (**minimize** L, figure 11).

Although bin-packing problem is NP-hard, simple heuristic algorithms are very efficient ("First Fit" or "Best Fit" decreasing algorithms), therefore slightly modified versions of these algorithms can be used for finding schedules without unwinding the loop. This is not our purpose here to develop that point further.

Instead, we allow loop unwinding and study for which unwinding degrees an optimal schedule can be found. When we unwind the loop u times, then we get u copies $T_j^1, T_j^2, \ldots T_j^u$ of each task T_j. Let us assume that we can compact these tasks within a pattern of length (latency) L^u, then the average latency we obtain is $L = L^u/u$. A well known lower bound for L is derived by writing that for any type j, the sum of tasks durations on the k_j functional unit of type P_j can not exceed $k_j \times L$ (L times the number of processors of type P_j.

$$L \geq L_{min}^{P_j} = \frac{\sum_{T \ of \ type \ P_j} d(T)}{k_j}$$

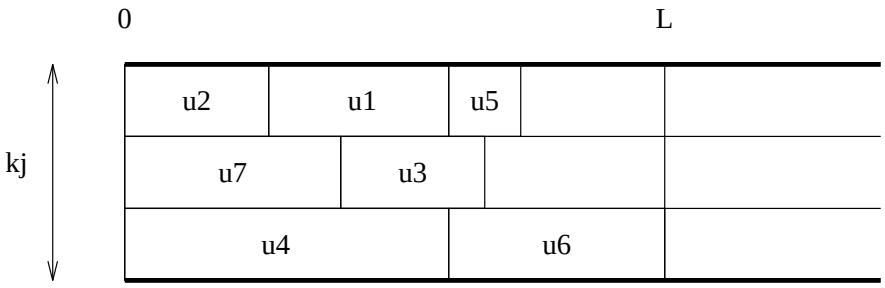

Figure 11: Minimizing the size L of boxes

From where we obtain the lower bound $L_{min} = max_j(L_{min}^{P_j})$. Now the question is: does there exists an unwinding degree u such that $L^u/u = L_{min}$. The answer is "yes" and a possible value for u is given in the following theorem:

Theorem 3.1 *Let $u = lcm(k_1, k_2, \ldots k_m)$, then there exists a schedule of the loop unwound u times with throughput equal to L_{min}.*

Proof 1 *Let us consider first only the functional units of type P_j and tasks of type P_j. We unroll the loop k_j times and on each functional unit, we compact all the tasks corresponding to the same iteration (figure 12) inside a pattern of (integer) length $k_j \times L_{min}^j$ (this is possible by definition of L_{min}^j). Then we repeat u/k_j times this schedule, by juxtaposing copies of that pattern. The length of whole pattern for tasks of type P_j is therefore $u/k_j \times k_j \times L_{min}^j = u \times L_{min}^j$; inside this pattern, the number of copies of original loop body is $k_j \times u/k_j = u$.*

The whole loop pattern is constructed by using this scheduling for each type of processor and embody it within a pattern of length $u \times L_{min}$. Since u copies of loop body are executed within each pattern, the resulting average throughput is $(u \times L_{min})/u = L_{min}$.

■

Therefore we know there always exists a solution to our throughput optimization problem, but at the price of an unwinding degree that may be large. This degree ($lcm(k_1, k_2, \ldots k_m)$) depends only on architectural features and not on the program. On current processors, the number of different functional units of the same type is usually not too large (1 or 2). But our motivation for investigating this study further is based on two points:

1. The number of functional units of the same type might be larger for future superscalar processors.
2. Cyclic register allocation is also likely to cause a loop unwinding. Interaction of both unwinding for scheduling and unwinding for register allocation can lead to a quite large unwinding degree (lcm of the both previous ones) if no precaution is taken.

The next sections are devoted to a precise study of available unwinding degrees for some simple cases.

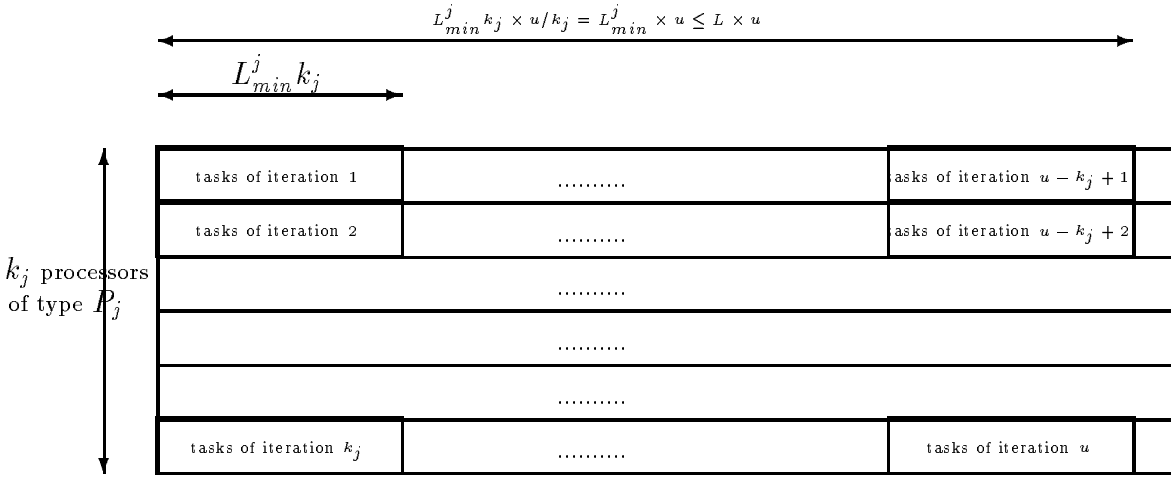

Figure 12: Pattern for a general solution

3.2 One type of F.U., one duration

In this section, all the tasks $T_1, T_2, \ldots T_n$ are to be run on the same type of functional units (P), and there are k different functional units of type P (this model also the case of a shared memory MIMD machine). All the tasks have the same duration d. Therefore L_{min} is here equal to nd/k.

In that case we have a precise result about unwinding degrees for which an optimal schedule exists:

Theorem 3.2 *The degrees u of unwinding for which the lower bound L_{min} is achieved are the integer multiples of*

$$\frac{k}{\gcd(nd, k)}$$

greater than k/n

Proof 2 *A number u is a valid unrolling degree iff it is possible to distribute equally the tasks among the processors. The total duration und must then be a multiple of the number of processors k:*

$$k | und^2$$

By dividing by $\gcd(nd, k)$, it follows that $k/\gcd(nd, k)$ must divide u. More, the resulting latency $L = uL_{min}$ must be large enough for containing at least one task: $L > d$. This implies that u must be greater than k/n.

■

[2] $x|y$ means that y is an integer multiple of x

In that case, we are therefore able to determine the minimum unrolling degree necessary for saturating the resources:

Corollary 3.3

$$u_{min} = \lfloor \frac{\gcd(nd,k)}{n} \rfloor \frac{k}{\gcd(nd,k)}$$

3.3 One type of F.U., two durations

Here we are in the case of one type of functional units P (in number k, P^1,P^2, ... P^k) and the tasks may have two different durations d_1 and d_2. We are given n_1 tasks of duration d_1 and n_2 tasks of duration d_2. Therefore the minimum achievable latency is

$$L_{min} = \frac{n_1 d_1 + n_2 d_2}{k}$$

In this case, we have still a simple characterization of valid unwinding degree:

Theorem 3.4 *A necessary and sufficient condition for the latency to achieve the lower bound L_{min} is that the unwinding degree u verifies:*

1. *uL_{min} is a multiple of d_{12}*
2. $$k\lceil -\frac{uaL_{min}}{d_2} \rceil \leq u(bn_1 - an_2) \leq k\lfloor \frac{ubL_{min}}{d_1} \rfloor \quad (1)$$

where a et b are integers verifying

$$ad_1 + bd_2 = d_{12} \quad (2)$$

($d_{12} = d_1 \wedge d_2$ stands for the gcd of d_1 and d_2)

Proof 3 *For a given unwinding degree u, we consider on each processor P^i ($i = 1, \ldots k$) the number a_i of tasks of duration d_1 and the number b_i of tasks of duration d_2. Since the processors are saturate, we obtain that:*

$$uL_{min} = a_i d_1 + b_i d_2 \quad (3)$$

This holds only if and only if there exists an integer λ_i such that:

$$a_i = \frac{uaL_{min}}{d_{12}} + \lambda_i \frac{d_2}{d_1 2} \quad (4)$$

$$b_i = \frac{ubL_{min}}{d_{12}} - \lambda_i \frac{d_1}{d_1 2} \quad (5)$$

Now we write that the whole number of tasks of duration d_1 (resp. d_2) is un_1 (resp. un_2):

$$un_1 = a_1 + a_2 + \ldots + a_k \quad (6)$$

$$un_2 = b_1 + b_2 + \ldots + b_k \quad (7)$$

From (4) and (6), we obtain:

$$\sum_{i=1}^{k} \lambda_i = u(bn_1 - an_2) \tag{8}$$

It remains to state that the a_i and b_i must be non negative, therefore:

$$-\frac{uaL_{min}}{d_2} \leq \lambda_i \leq \frac{ubL_{min}}{d_1} \tag{9}$$

Thus the conditions for unwinding degree u to be a valid one is that (8) and (9) hold for some integers λ_i.

It is straightforward to see that (8) and (9) imply (1).

Now let us assume that (1) holds. We perform the integer division of $u(bn_1 - an_2)$ by k:

$$u(bn_1 - an_2) = Qk + R$$

Q and R are integers, and R is comprised within 0 and $k-1$. If $R = 0$ then we take $\lambda_i = Q$ for every i and (8) and (9) are verified. Else, we take $\lambda_1 = \lambda_2 = \ldots = \lambda_R = Q + 1$ and $\lambda_{R+1} = \lambda_{R+2} = \ldots = \lambda_k = Q$. Then (8) is trivially verified. For (9), since $1 \leq R$,

$$\lambda_i \leq Q+1 \leq \frac{1}{k}(u(bn_1-an_2)-R)+1 \leq \frac{1}{k}(k\lfloor\frac{ubL_{min}}{d_1}\rfloor - R)+1 \leq \lfloor\frac{ubL_{min}}{d_1}\rfloor - \frac{R}{k}+1 \leq \frac{ubL_{min}}{d_1}$$

The last inequality is due to the fact that $-\frac{R}{k} + 1 < 1$.

The lower bound of λ_i is deduced in the same way, from $R \leq k-1$.

■

Therefore determining whether a given integer u is a valid degree or not can be done in polynomial time. For determining the least available value, it is sufficient to verify this condition for any integer least than the upper bound given by theorem 3.1, that depends only on the architecture. Thus, the problem of determining the minimal unwinding degree necessary for optimal throughput on a given architecture is polynomial. This result is very interesting in practical cases, because it happens often that there are only two different durations on a given type of processor (for instance one duration for floating operation in simple precision, another duration for double precision).

Example 1 *Consider $k = 7$ identical processors and a loop with $n_1 = 6$ tasks of duration $d_1 = 3$ and $n_2 = 2$ tasks of duration $d_2 = 5$. Then we obtain $L = 4$. By taking $a = 2$ and $b = -1$, the table 1 gives the values for the lower and upper bounds of previous inequalities.*

Therefore, the least valuable value for L is 5 for which we obtain the pattern drawn on figure 13.

The first condition can not be deduced from the second one. This can be seen on the following example:

Example 2 *Consider $k = 33$ processors and a loop with $n_1 = 3$ tasks of duration $d_1 = 18$ and $n_2 = 7$ tasks of duration $d_2 = 30$. Then we obtain $L = 8$. By taking $a = -3$ and $b = 2$, we obtain the values given in table 2. The first value for which the inequality (1) holds is $u = 11$, but since $uL = 88$ is not a multiple of $d_{12} = 6$, 11 is not valuable. The least valuable degree is $u = 15$ (the next others are 21, 27, 30, 33, ...).*

u	$k\lceil -\frac{uaL_{min}}{d_2} \rceil$	$u(bn_1 - an_2)$	$k\lfloor \frac{ubL_{min}}{d_1} \rfloor$	
1	-7	-10	-14	
2	-21	-20	-21	
3	-28	-30	-28	
4	-42	-40	-42	
5	-56	-50	-49	$\Leftarrow$
6	-63	-60	-56	$\Leftarrow$
7	-77	-70	-70	$\Leftarrow$

Table 1: Values for example 1

3.4 One type of F.U., more than two durations

When there are more than two different durations on a given processor, equations like those of previous section can be also stated, and the problem of determining whether an unrolling degree is valid amounts to a integer programming satisfiability problem,. This problem is known to be NP-complete, but polynomial for a fixed number of variables or equations [Len83]. However the complexity grows exponentially with the numbers of variables. Since this number is here $k\times$ the number of different durations, the complexity makes that problem intractable as soon as the architecture becomes more complex.

To our knowledge, this is an open problem whether a simpler condition for an unwinding degree to be available exists. In that case, the easiest is to use a bin packing heuristic like the one mentioned at the head of this section.

3.5 The general case

Until now, we have made the assumption that only one type P of F.U. was under consideration and that it was the (or one) bottleneck of the scheduling problem. In other words, $L_{min} = L^P_{min}$.

The first step towards the generalization is to consider a non bottleneck functional unit, i.e. $L_{min} > L^P_{min}$. The principle is to add fictitious tasks with duration 1 to come back to the case of a bottleneck functional unit.

The second step is to take into account all the F.U. types in the machine, whether it is a bottleneck or not.

3.5.1 One non bottleneck F.U.

In that case, we have $L_{min} > L^P_{min} = \frac{\sum_i n_i d_i}{k_p}$. We introduce fictitious tasks with durations 1, that we add to the system (if there already exists tasks of duration 1 (say n_1), then we increase n_1 by a number n_0, else we introduce $d_0 = 1$ and create n_0 tasks. The number of new tasks is set to $L_{min}k_p - \sum_i n_i d_i$. The new resulting latency L'_P is $\frac{\sum_i n_i d_i + n_0}{k_P} = L_{min}$ and the F.U. is considered as a bottleneck and we turn up to the previous case.

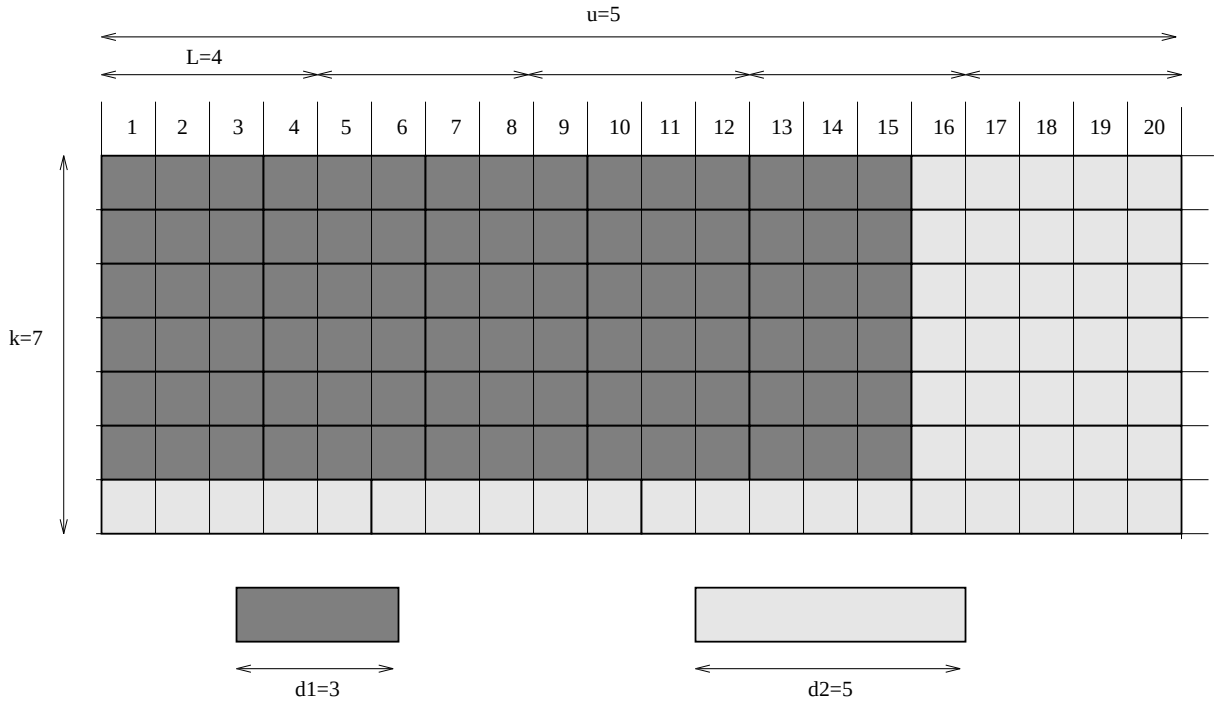

Figure 13: A solution to example 1

3.5.2 More than one F.U. type

Let us sum up what we have found until now:

- In any case, the $u = lcm(k_1, k_2, \ldots k_m)$ is a valid unwinding degree.
- When the F.U. type is a program bottleneck and there are no more than two different durations for tasks on that functional unit, then we have a very simple condition for verifying whether an unwinding degree is available or not.
- If the F.U. type is not a program bottleneck, then we must add fictitious tasks of duration 1 to come back to the previous case.
- When there are more than 2 different durations on the F.U., then we don't have any simple condition for verifying that an unwinding degree is valid, but we can try to schedule the tasks within the generated pattern with the "bin packing" based heuristic.

The algorithm we propose is to verify for each integer value from 1 to $lcm(k_1, k_2, \ldots k_m)$ if there exists a valid loop schedule for that value. Since this upper bound depends only on the architecture and each step can be performed in polynomial time, we conclude that the problem of minimizing the unwinding degree is polynomial for a given architecture, but we

don't know whether the degree of the polynom mentioned in section3.4 can be reduced, like in the case of 2 different durations.

4 Minimizing the span for one given schedule

In the previous sections, we showed how to build a compact reservation table with an optimal initiation interval with respect to the resource constraints. This was done by ignoring the dependencies between tasks. In this section we show how to derive the schedule for a given iteration from the reservation table and the dependency graph. For the schedule to be legal, it must satisfy all dependencies between tasks. Moreover, it is important that the makespan be minimized. As a matter of fact, the larger the makespan, the more iterations are simultaneously alive and the more registers are required.

Let L be the initiation interval associated to the reservation table and ω_i ($\omega_i \in [0, L-1]$) be the location of the template of task T_i in the reservation table.

In the schedule of a given iteration, the corresponding occurence of task T_i can only be scheduled at dates which are equal to ω_i modulo L so that the execution of the loop in the steady state conforms to the reservation table.

Therefore, the issue date of the k^{th} occurence of task T_i can be written as

$$\Omega(T_i^k) = \omega_i + (k + p_i) * L,$$

for some integer p_i. Intuitively, a given iteration extends over several consecutive blocks of length L and p_i indicates in which block task T_i should execute.

Now the problem consists in determining the values of p_i such that all dependencies appearing in the dependency graph are satisfied. Let $G = (V, U)$ be the dependency graph. Let us consider the following dependency $(T_i, T_j, v_{ij}, h_{ij})$. This dependency directly translates into the following constraint on the issue dates:

$$\forall n,\ \Omega(T_j^{n+h_{ij}}) \geq \Omega(T_i^n) + v_{ij}$$

and

$$\forall n,\ \omega_j + (n + p_j + h_{ij}) * L \geq \omega_i + (n + p_i) * L + v_{ij}$$

$$\forall n,\ p_j - p_i \geq \lceil (\omega_i - \omega_j + v_{ij})/L \rceil + h_{ij})$$

The dependencies are satisfied if and only if the values p_i satisfy the following constraints:

$$\forall (T_i, T_j, v_{ij}, h_{ij}) \in U, p_j - p_i \geq \lceil (\omega_i - \omega_j + v_{ij})/L \rceil + h_{ij}$$

As a consequence, $(p_i)_{i=1,\ldots,m}$ appears to be a potential on the graph Γ derived from G in the following way:

- Γ has the same set of vertices as G
- for each edge $(T_i, T_j, v_{ij}, h_{ij})$ in G, there is an edge from T_i to T_j with a weight equal to $\lceil (\omega_i - \omega_j + v_{ij})/L \rceil + h_{ij}$

The problem of finding p_i such that the previous constraints are satisfied is a well known operational research problem (longest path algorithm) which can be solved in polynomial time.

5 Application to loop scheduling for the Intel i860

The class of software pipelining algorithms which was introduced in the previous sections allowed us to generate throughput optimal loop schedules for a general class of resource constraints.

In this section we show how this approach can be used to generate code for explicitly advanced pipelines.

5.1 Explicitly advanced pipelines

The i860 provides a high-performance pipelined floating-point engine which can execute a floating-point addition and a floating-point multiply in the same cycle (this is called *dual operation mode*). The floating-point pipelines can be operated in either of two modes: scalar mode or pipelined mode. In scalar mode, a floating-point instruction is not allowed to issue before the previous instruction executing on the functional unit has completed. In the pipelined mode, instructions are allowed to overlap which results in a potentially higher throughput. However, in pipelined mode, the control of the pipelined functional units is the responsibility of the compiler or the assembly code writer. This type of pipelines is known as an *explicitly advanced pipelines*.

To make things clear we illustrate the working of an explicitly advanced pipeline of depth 3 for example. A pipelined instruction operating on this pipeline has the following effect:

- the result present in the third stage of the pipeline is written back in the register file in the destination register encoded in the current instruction
- the intermediate results are pushed from stage 2 to stage 3 and from stage 1 to stage 2
- the operands encoded in the current instruction are loaded in the first stage of the pipelined functional unit.

Note that for a pipeline of depth s, a pipelined instruction writes the result of the $s+1$ previous instruction operating on the pipelined functional unit. As matter of fact, a pipelined instruction takes one cycle, its reservation table is a single column. This is very different from the standard mode of control of pipelines, where an instruction operating on a pipelined functional unit triggers a sequence of operations at successive cycles and whose reservation table has a diagonal shape.

Explicitly advanced pipelines make code generation very difficult for the following reasons:

- the pipelined mode is only worth if one has to execute a sequence of independent operations on the pipeline, else the scalar mode is just as efficient and far simpler to manage
- when using the pipelined mode, one has to take care that dummy pipelined instructions are generated in order to push the last results out of the pipelined functional unit
- standard instruction scheduling algorithm are not adapted since they rely on the assumption that the availability date of the result of any instruction is known as soon

as the issue date of the instruction is known. This is unfortunately not the case for explicit advanced pipelines where the result of a computation becomes available only after a given number of instructions operating on the same pipeline has been executed.

In [BEH91], a technique inspired by microcode compaction is described to schedule pipelined instruction in a basic block. However this technique does not apply in the case of loops.

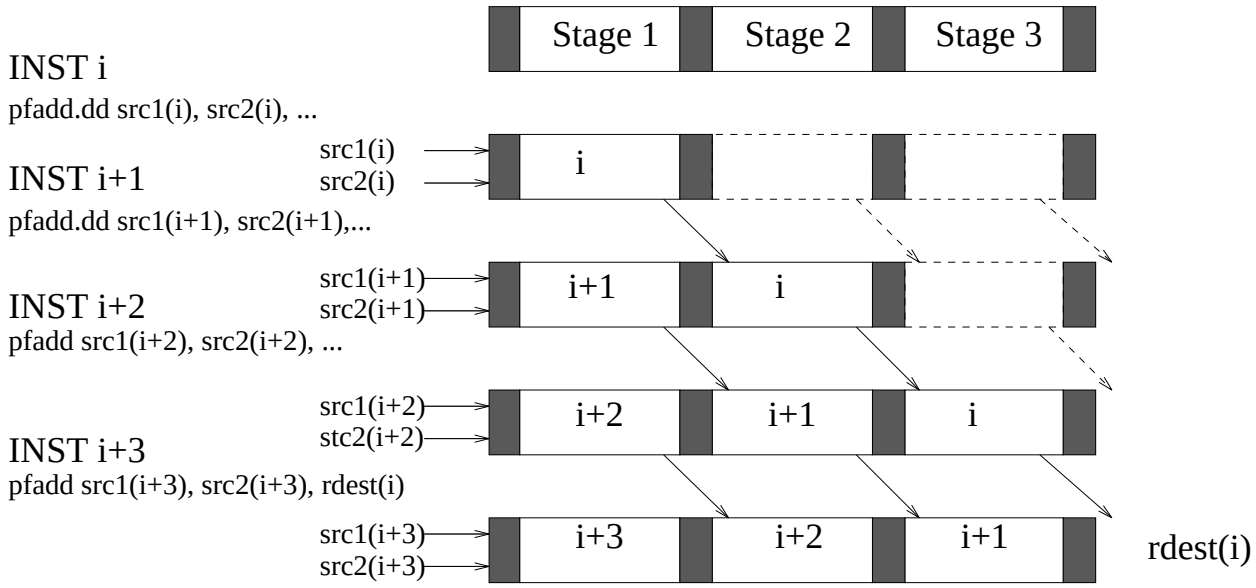

Figure 14: Explicitly Advanced Pipelines

5.2 The loop scheduling algorithm

The input of the loop scheduling algorithm is the cycle-free dependency graph whose nodes are the instructions of the intermediate code corresponding to the loop body. The instructions are denoted I_0 through I_{n-1} and the set of instructions is noted $\mathcal{I}$.

For the sake of simplicity in the presentation, we assume that we have a single explicit advanced pipeline with s stages. (This is no real restriction as long as the different pipelined functional units are operated by distinct assembly instructions.) Depending on its type, an instruction I in $\mathcal{I}$ is either carried out by a scalar assembly instruction or by a pipelined assembly instruction. This defines a partition of $\mathcal{I}$ into two subsets: the set of *scalar* instructions $\mathcal{S}$ and the set of *pipelined instructions* $\mathcal{P}$. Let p be the number of pipelined instructions. As in the previous chapters, each instruction, scalar or pipelined is associated with a reservation table. The reservation table of a pipelined instruction has a simple shape: the different stages of the pipeline are all used at the same cycle.

Due to the presence of pipelined instructions, we cannot straightforwardly derive the precedence constraint graph from the dependence graph. As a matter of fact the duration of a pipelined instruction, i.e. the delay between issue time and the time when the result is available, depends on the schedule of the next s pipelined instruction which push the result. Therefore a two step algorithm is required to build a loop schedule.

Step 1

The elementary reservation tables are compacted into the shortest possible schedule pattern with the assumption that no dependencies exist between the instructions. The corresponding initiation interval is called λ. Due to the simple shape of the elementary reservation tables (as long as one does not mix simple precision and double precision floating-point computations in a loop) an optimal schedule pattern can easily be built.

The schedule pattern is described by a mapping $\omega : \mathcal{I} \longrightarrow N$ which gives the starting position of the reservation table of each instruction in the schedule pattern. Now it is important to know in which order the pipelined instructions occur in the schedule pattern if one wants to determine precisely the duration of a pipelined instruction. This order is described by the mapping $\pi : [0, p-1] \longrightarrow [0, n-1]$:

$$\forall k \in [0, \ldots, p-1], \quad I_{\pi(k)} \in \mathcal{P}$$

$$0 \leq \omega(I_{\pi(0)}) < \ldots < \omega(I_{\pi(k-1)}) < \omega(I_{\pi(k)}) < \ldots < \omega(I_{\pi(p-1)}) < \lambda$$

Let $I \in \mathcal{I}$ be a pipelined instruction and let k be its position in the schedule pattern (i.e. $I = I_{\pi(k)}$).

- the pipelined instruction pushing the result of instruction I out of the pipeline is instruction $I_{\pi((k+s) \bmod p)}$
- the result is available at time:

$$\omega(I_{\pi((k+s) \bmod p)}) + \tau(I_{\pi((k+s) \bmod p)}) + \lambda \lfloor \frac{k+s}{p} \rfloor$$

Now assume that I_i and I_j are two instructions linked by a dependence in the dependence graph. This dependence induces a precedence constraint described by the tuple $(I_i, I_j, v_{i,j}, h_{i,j})$ where

- $h_{i,j}$ is entirely determined by the height of the dependence relating I_i and I_j.
- the duration $v_{i,j}$ is defined as:

$$v_{i,j} = \begin{cases} \tau(I_i) & \text{if } I_i \in \mathcal{S} \\ \omega(I_{\pi((k+s) \bmod p)}) + \tau(I_{\pi((k+s) \bmod p)}) - \omega(I_{\pi(k)}) + \lambda \lfloor \frac{k+s}{p} \rfloor & \text{if } I_i = I_{\pi(k)} \in \mathcal{P} \end{cases}$$

Step 2

Now it is possible to determine an iteration indexing function so that all data dependencies are satisfied.

The following remarks should be made concerning the scheduling algorithm described above:

- the proposed algorithm belongs to the class of software pipelining algorithms described above: this approach is perfectly suited for handling explicitly advanced pipelines.
- in the software pipelined loop body, each original floating point operation (add or multiply) corresponds to a single pipelined instruction, i.e. no dummy pipeline pushes are required. This allows us to save instruction issue bandwidth.
- the algorithm is relatively simple to implement: it has been implemented in our retargetable object code optimizer

5.3 A complete example

Let us consider the following Fortran DO loop:

Source code:

```
      DO 1 i = 1, N
            Y(i) = A * X(i) + B
1     CONTINUE
```

I860 Assembly Code:

```
      .xxx LOOP BEGIN
Loop:
      .xxx mem "READ_FROM" 1
      fld.d 8(r7)++, v90          // @X++;load X(i)
      fmul.dd v90, f8, v92        // A*X(i)
      fadd.dd f10, v92, v94       // A*X(i) + B
      bla r4, r5, Loop
      .xxx mem "WRITE_TO" 2
      fst.d v94, 8(r6)++          // @Y++;store Y(i)
      .xxx LOOP END
```

The schedule pattern generated by the algorithm described above is shown in figure 15. The initiation interval is equal to 3 and the throughput of the schedule is 2 floating point operations every 3 cycles. This is the best possible initiation interval since one of the resource is busy over 3 cycles. Therefore the theoretical performance is 26.6 MFlops (on a 40 Mhz i860 processor). The assembly code was generated automatically with our retargetable object code optimizer OCO.

Figure 16 shows the actual performance of our software pipelined code (curve i860-oco) as well as the performance of two different codes generated by the Portland Group Fortran

Compiler. The performance is plotted as a function of the number N of iterations of the loop in order to emphasize the effect of the cache on the performance.

Two set of optimizations were used for the Portland Group Fortran Compiler PGFTN:

1. Option: -O4
   software pipelined code is generated by the compiler

2. Option: -O4 -MVect=recog,unroll
   the compiler vectorizes the loop in order to exploit the pipelined memory access mode.

The code generated by our algorithm outperforms both codes generated by PGFTN in the cache region, however as the size of the data increases the vectorized code performs better. It should be noted that our software pipelining algorithm can be combined with vectorization techniques.

6 Conclusion

In this report we have presented a new approach for loop scheduling with resource constraints, based on a decoupling of the problem into two phases, namely tasks packing for optimizing resource usage, then iteration allocation for verifying semantical constraints. This approach has permitted to concentrate on two somewhat unusual optimization criteria:

1. determining the valid unwinding degrees that lead to optimal schedules,.

2. minimizing the makespan of one iteration among loop patterns.

A byproduct of that method is that it permits to handle the exotic pipeline execution mode of the famous i860 processor and generate optimal schedules for a large class of loops (among which vector loops) for that processor.

References

[Aik88] A. Aiken. *Compaction-Based Parallelization.* PhD thesis, Department of Computer Science, Cornell University, Technical Report No. 88-922, 1988.

[AN88] A. Aiken and A. Nicolau. Perfect pipelining: a new loop parallelization technique. *Lecture Notes in Computer Science*, (300):221–235, 1988.

[BCW89] F. Bodin, F. Charot, and C. Wagner. Overview of a high-performance programmable pipeline architecture. *ACM Supercomputing 89 (Crete)*, pages 398–409, 1989.

[BEH91] D. G. Bradlee, S. J. Eggers, and R. R. Henry. Integrated register allocation and instruction scheduling for RISCs. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 1991.

[Bod89] F. Bodin. *Optimisation de microcode pour une architecture horizontale et synchrone: Etude et mise en oeuvre d'un compilateur.* PhD thesis, Irisa, Université de Rennes 1, 1989.

[Cha81] A. E. Charlesworth. An approach to scientific array processing: The architectural design of the AP-120B/FPS-164 family. *Computer*, 14(9):18–27, 1981.

[Ebc87] K. Ebcioglu. A compilation technique for software pipelining of loops with conditional jumps. In *Proc. of the 20th Annual Workshop on Microprogramming*, pages 69–79, December 1987.

[Eis88] C. Eisenbeis. Optimization of horizontal microcode generation for loop structures. In *Proceedings of the International Conference on Supercomputing*, pages 453–465, Saint-Malo, France, July 1988.

[EJL88] C. Eisenbeis, W. Jalby, and A. Lichnewsky. Squeezing more CPU performance out of a Cray-2 by vector block scheduling. In *Proceedings of Supercomputing'88*, Kissimee, Florida, 1988.

[EJL90] C. Eisenbeis, W. Jalby, and A. Lichnewsky. Compiler techniques for optimizing memory and register usage on the Cray-2. *International Journal on High Speed Computing*, 2(2), June 1990.

[GJ79] M. R. Garey and D.S. Johnson. *Computers and Intractability, A Guide to the Theory of NP-Completeness.* W.H. Freeman and company, 1979.

[GS91] Franco Gasperoni and Uwe Schwiegelshohn. Efficient algorithms for cyclic scheduling. Technical Report RC 17068, IBM Research Division, July 1991.

[Han90] C. Hanen. Study of a NP-hard cyclic scheduling problem: The periodic recurrent job-shop. In UPMC Laboratoire MASI Ecole des Mines de Paris CAI, editor, *International Workshop on Compilers for Parallel Computers*, Paris, December 3-5 1990.

[Hsu86] P. Y.-T. Hsu. *Highly Concurrent Scalar Processing.* PhD thesis, University of Illinois at Urbana-Champaign, January 1986.

[Lam87] M. Lam. *A Systolic Array Optimizing Compiler.* PhD thesis, Carnegie Mellon University, May 1987.

[LDS80] D. Landskov, S. Davidson, and B. Shriver. Local microcode compaction techniques. *Computing Surveys*, 12(3):261–294, February 1980.

[Len83] H. W. Lenstra. Integer programming with a fixed number of variables. *Mathematics of Operations Research*, 8(4):538–548, 1983.

[Mun91] Alix Munier. Résolution d'un problème d'ordonnancement cyclique à itérations indépendantes et contraintes de ressources. *Recherche Opérationnelle / Operations Research*, 25(2):161–182, 1991.

[RG81] B.R. Rau and C.D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *Proceedings of the* 14^{th} *Conference on Microprogramming and Microarchitecture*, pages 183–198, Octobre 1981.

[TDT88] J.H. Tang, E.S. Davidson, and J. Tong. Polycyclic vector scheduling vs. chaining on 1-port vector supercomputers. *Proceedings of Supercomputing '88,*, November 1988.

[Tou84] R. F. Touzeau. A FORTRAN compiler for the FPS-164 scientific computer. In *Proc. of the ACM SIGPLAN'84 Symposium on Compiler Construction*, 1984.

[WE93] J. Wang and C. Eisenbeis. Decomposed software pipelining: a new approach to exploit instruction level parallelism for loop programs. In *Proceedings of IFIP WG 10.3 Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, Orlando, Florida, January 1993. North-Holland.

u	$k\lceil -\frac{uaL_{min}}{d_2} \rceil$	$u(bn_1 - an_2)$	$k\lfloor \frac{ubL_{min}}{d_1} \rfloor$	
1	33	27	0	
2	661	54	33	
3	99	81	66	
4	132	108	99	
5	132	135	132	
6	165	162	165	
7	198	189	198	
8	231	216	231	
9	264	243	264	
10	264	270	264	
11	297	297	297	$\leftarrow$
12	330	324	330	
13	363	351	363	
14	396	378	396	
15	396	405	429	$\Leftarrow$
16	429	432	462	$\leftarrow$
17	462	459	495	
18	495	486	528	
19	528	513	528	
20	528	540	561	$\leftarrow$
21	561	567	594	$\Leftarrow$
22	594	594	627	$\leftarrow$
23	627	621	660	
24	660	648	693	
25	660	675	726	$\leftarrow$
26	693	702	759	$\leftarrow$
27	726	729	792	$\Leftarrow$
28	759	756	792	
29	792	783	825	
30	792	810	858	$\Leftarrow$
31	825	837	891	$\leftarrow$
32	858	864	924	$\leftarrow$
33	891	891	957	$\Leftarrow$

Table 2: Values for example 2

Cycle	0	1	2
core_issue	L(i)	B	S(i-6)
bus			
fld_1			
fld_2			
fld_3			
fld_res			L(i)
fpu_issue	*(i-1)	+(i-3)	
fpu_mul_1		*(i-1)	*(i-1)
fpu_mul_2		*(i-2)	*(i-2)
fpu_mul_res		*(i-3)	
fpu_add_1			+(i-3)
fpu_add_2			+(i-4)
fpu_add_3			+(i-5)
fpu_add_res			+(i-6)

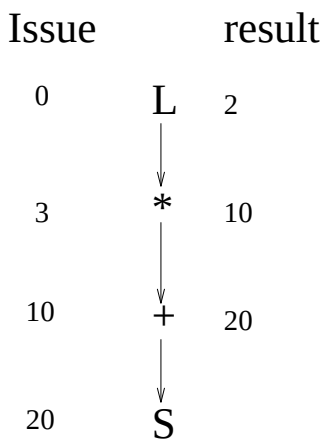

Figure 15: Schedule pattern

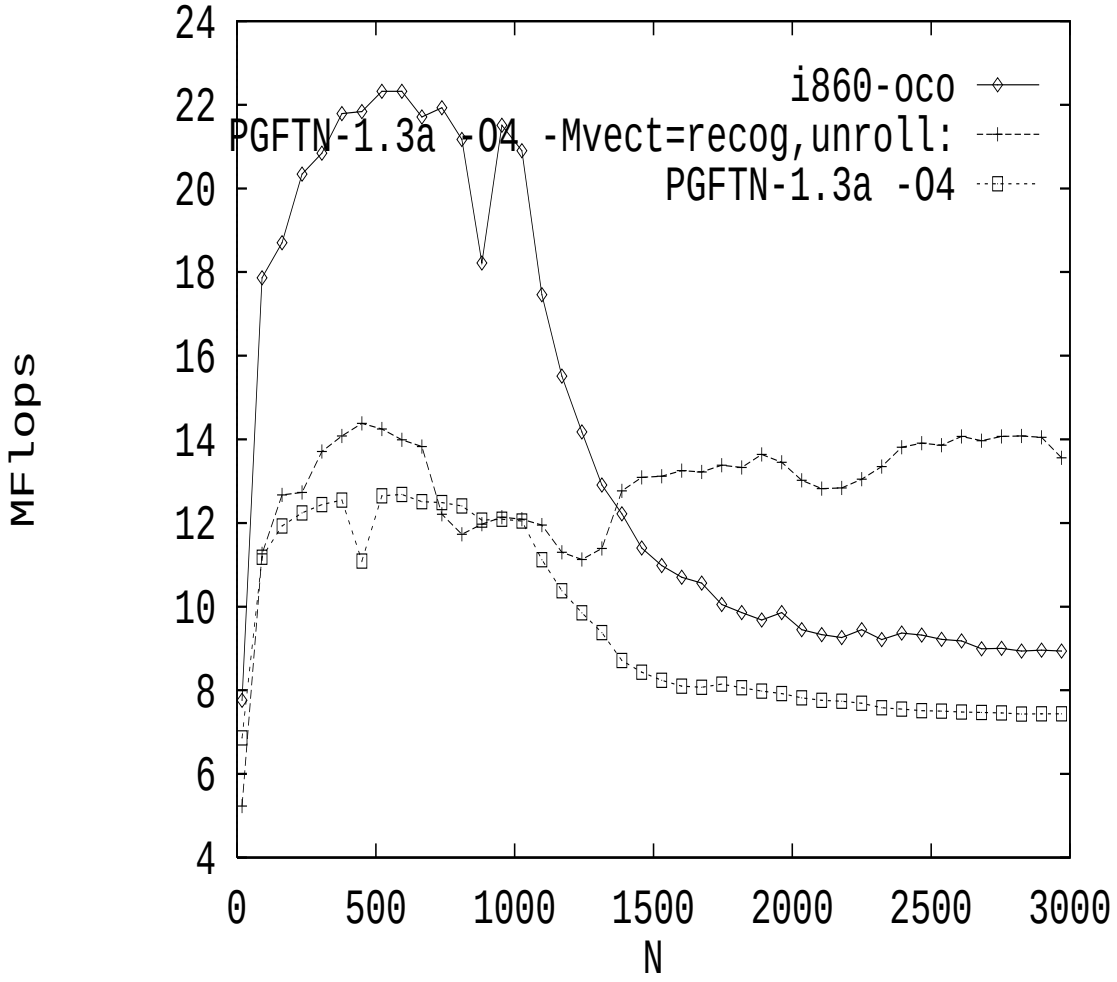

Figure 16: Performance of Kernel 1