# Demystifying On-the-Fly Spill Code

Alex Aletà
UPC, Barcelona
aaleta@ac.upc.edu

Josep M. Codina
Antonio Gonzàlez
UPC - Intel Labs, Barcelona
josex.m.codina,antonio.gonzalez@intel.com

David Kaeli
Northeastern Univ., Boston, MA
kaeli@ece.neu.edu

## Abstract

Modulo scheduling is an effective code generation technique that exploits the parallelism in program loops by overlapping iterations. One drawback of this optimization is that register requirements increase significantly because values across different loop iterations can be live concurrently. One possible solution to reduce register pressure is to insert spill code to release registers. Spill code stores values to memory between the producer and consumer instructions.

Spilling heuristics can be divided into two classes: 1) *a posteriori* approaches (spill code is inserted after scheduling the loop) or 2) *on-the-fly* approaches (spill code is inserted during loop scheduling). Recent studies have reported obtaining better results for spilling *on-the-fly*. In this work, we study both approaches and propose two new techniques, one for each approach. Our new algorithms try to address the drawbacks observed in previous proposals. We show that the new algorithms outperform previous techniques and, at the same time, reduce compilation time. We also show that, much to our surprise, *a posteriori* spilling can be in fact slitghtly more effective than *on-the-fly* spilling.

***Categories and Subject Descriptors*** D.3.4 [*Processors*]: Code generation/Compilers

***General Terms*** Algorithms, Languages

***Keywords*** Modulo scheduling, register allocation, spill code

## 1. Introduction

As the complexity of microprocessors continues to increase with each new generation, power consumption becomes a growing issue. One possible solution is to use a simpler processor, such as a *very long instruction word* (*VLIW*) architecture. These architectures have been shown to provide a good compromise between performance and power consumption. VLIW processors execute instructions in order. Therefore, performance relies heavily on the compiler, and in particular, efficient code generation.

In recent years, VLIW architectures have become very popular in the embedded DSP domain. In this context, multimedia applications and numerical code are the most commonly executed programs. For these applications, loops represent the majority of execution time. Hence, code generation for cyclic code becomes a key issue when considering compilers for VLIW-based systems. In this area, software pipelining techniques have been shown to be very useful to boost the execution performance of loops. To effectively schedule loop bodies, *modulo scheduling* (*MS*) [15] has been shown to achieve very good results.

MS increases parallelism by overlapping the execution of multiple iterations of a loop. New iterations can start without waiting for the previous iterations to finish. A new iteration can start after a constant number of cycles. This constant number of cycles is known as the *initiation interval*, or *II*. Since many iterations are executed concurrently, register pressure increases because many instances of the same instruction from different loop iterations may be live concurrently. In this work we have assumed a file that uses rotating registers [6] to deal with these lifetimes.

The use of cluster architectures is spreading to overcome delays that arise due to the transmission of signals with respect to the clock frequency [1]. For clustered processors, register pressure is even higher because of several issues. First, some values may be live in various clusters concurrently, therefore consuming one register in each cluster. Moreover, computation may not be balanced across clusters, thus some of the clusters will be responsible for larger portion of the computation. This imbalance may result in increased register pressure. Finally, inter-cluster communications may increase the lifetime of some values.

Spill code reduces register requirements at the expense of increasing memory traffic. Spilling a value means storing that value to memory. Then, the register where the value was being maintain can be released between consecutive uses. Before the next use, the value is re-loaded from memory.

Based on the phase of instruction scheduling where spill code is inserted, spill algorithms can be divided into two main categories: spill code *a posteriori* and spill code *on-the-fly*. The former inserts spill instructions after scheduling the loop. The later adds spill code during loop scheduling. *On-the-fly* spilling has more freedom to spill. Besides, scheduling decisions can also take advantage of register spilling. Therefore, it seems to have a bigger potential. In fact, current state-of-the-art modulo scheduling techniques insert *on-the-fly* spill code [5, 20]. We performed a thorough study of previous *on-the-fly* spilling schemes. This study has motivated the design of the new scheme described in this paper that overcomes many of the drawbacks we found. At the same time, we have studied previous *a posteriori* spill algorithms and identified their main limitations. Based on this study, we then developed an alternative spilling scheme for *a posteriori* also.

In this paper we present an evaluation of both techniques and compare them against state-of-the-art spilling algorithms. Results show that both schemes outperform previous approaches and, at the same time, they reduce compile time. However, one surprising result was that *a posterori* and *on-the-fly* techniques turned out



*PLDI'05* June 12-15, 2005, Chicago, Illinois, USA

to achieve very similar results. In fact, *a posteriori* achieves even slightly better performance.

In this work, we describe in detail both algorithms and report the results obtained. We also analyze the the cause of why spill *a posteriori* performs better than *on-the-fly*, and present an explanation.

The rest of this paper is organized as follows: In section 2 we give some back-ground on spill code, we define some basic concepts and relate them with previous work. In sections 3 and 4 we propose two new spilling techniques, one *a posteriori* and one *on-the-fly* based on our study of previous schemes. In section 5 we evaluate the new algorithms and compare them with stata-of-the-art heuristics. Finally, in section 6 we present the conclusions of this work.

## 2. Spill code basics

For a given modulo schedule *S* and a cycle *c*, we will let *alive(c)* denote the number of values that are alive in cycle *c*. We then define:

$$maxLive = \max_{0 \leq c \leq II-1} \{alive(c)\}$$

and the cycle *c* where $alive(c) = maxLive$ is defined as the *critical cycle*, denoted *CC*. As was shown in [14], *maxLive* is an accurate approximation of the number of registers required by the schedule. Hence, if the value for *maxLive* is greater than the number of registers in the microarchitecture (referred as *n_of_reg* hereafter), then the schedule is not valid. In order to reduce register pressure we can add spill code, which involves storing values to memory in order to release a register for a number of cycles. These values are reloaded when a consumer instruction needs them. We will refer to the store instructions inserted for spill purposes as *spill-stores* or *s-stores* and to new loads as *spill-loads* or *s-loads*.

There have been a number of approaches proposed to perform register allocation and spilling for acyclic code. Heuristics based on graph coloring [4] have been shown to be very effective. Unfortunately, these schemes cannot be applied to modulo-scheduled loops because values produced by multiple instances of the same instruction in a loop may be live concurrently. There have been some solutions proposed to overcome these limitations [14], though they do not allow for the insertion of spill code. In this paper, we focus on spill code generation for modulo-scheduled code.

An alternative scheme to register spilling for modulo schedules is to increase the II. In this case, the amount of overlap between iterations decreases and therefore fewer values are live concurrently. However, reducing overlap also reduces the amount of parallelism. In fact, the goal of modulo scheduling is to try to extract parallelism by overlapping loop iterations. In addition, increasing the II does not guarantee that we can produce a valid schedule. This is especially true for graphs with recurrences. On other hand, if we can produce a valid schedule by inserting spill code, we can maintain the II, but at the expense of increasing memory traffic. In general, adding spill code is a better solution as has been shown in [12].

We can choose to generate spill code for *variables* or for *uses*. Spilling for variables means storing a value to memory just after producing the variable value and reloading it to a register for each consumer instruction (see figure 1 *c*). Alternatively, spilling for uses means that the value is only loaded for a subset of its consumers (see figure 1 *b*). Therefore, the value has to be kept in a register for some number of cycles. Both spill options are compared in [19] where it is shown that spilling of uses outperforms spilling of variables. Moreover, spilling of variables increases memory traffic more than spilling of uses.

We define the time between two subsequent uses of a variable as the *inter-use time*. This quantity will be used to help decide which uses we will generate spill code for. Figure 1 shows more clearly how we define inter-use times.

Some other important characteristics of a spill heuristic are:

- how to select the values to spill,
- how many values to spill, and
- how to schedule spill instructions.

All these features are heavily dependent on when spilling is performed. We can generate spill code after scheduling (*i.e.*, spill *a posteriori*), or during scheduling (*i.e.*, spill *on-the-fly*). We describe these two different approaches, as well as the other characteristics, next.

### 2.1 Spill code a posteriori

Spill code *a posteriori* is the name of a family of spill heuristics. The main feature of these schemes is that spilling is considered only after a loop is scheduled. These approaches can take full advantage of the information provided by the schedule in order to identify for which *uses* spills will be inserted for.

On the other hand, modulo scheduling produces very compact code, so it may be difficult to fit the new spill instructions within the computed schedule. To address this issue, previous *a posteriori* approaches compute a new schedule once spill code has been added.

Usually, spill code is inserted for the consumer instruction possessing the longest inter-use lifetime (based on the schedule before adding spill code). We can further tune the accuracy of this heuristic if we weight the inter-use lifetime by dividing it by the number of memory instructions that had to be added for the spill (denoted as *newMemTraf*). In [19], it is also required that the selected inter-use spans the *CC*.

Another important feature of an *a posteriori* spill code algorithm is when to stop adding spills (*i.e.*, how many spill candidates are selected at a time). Since the new spill instructions have not been scheduled yet, we do not know the actual register pressure. Therefore, we have to estimate how many values need to be spilled. One option is to add only one spill, then re-schedule the graph and repeat the process until no further spills can be added or until a valid schedule is found [12]. This approach prevents over-spilling at the expense of more computation time. The opposite option is to add as many spill instructions as possible, that is, until we saturate memory bandwidth [18]. This scheme is faster because re-scheduling is performed only once. However, it over-spills (that is, it introduces more spill code than it is really requiered) so memory traffic increases significantly. For that reason, the resulting schedules are not always beneficial. An hybrid option is to spill $2^k$ candidates at a time (where *k* is the number of times that we have re-scheduled the graph) [16]. This approach is faster than spilling a candidate at a time and does not introduce as many spill instructions as saturating memory bandwidth. Finally, a different approach is to assume that the spill instructions will free a register during all the inter-use times. Using this assumption, a new approximation of *maxLive* is achieved. Then spilling will stop when the estimated *maxLive* is low enough (though when re-scheduling, the real *maxLive* may turn out to be bigger) [19].

As discussed above, previous *a posteriori* spill techniques re-schedule the loop after inserting spill code. When performing re-scheduling, register pressure can be further reduced if we schedule the s-stores as close as possible to the value-producer instruction and s-loads as close as possible to the consumers. To achieve this goal, previous *a posteriori* schemes scheduled spill instructions and their associated producers/consumers as a single instruction-pair. These instruction-pairs are more difficult to schedule, which can lead to a longer schedule. In some cases, these instructions cannot

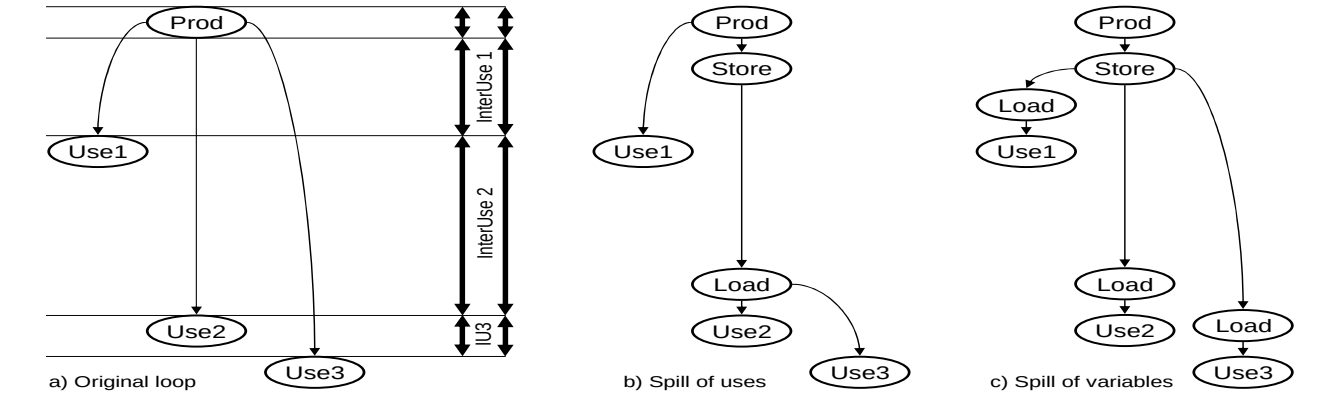


**Figure 1.** Example: spilling for uses and spilling for variables.

be scheduled due to resource conflicts and then the II must be increased.

### 2.2 On-the-fly spill code

On-the-fly spill code heuristics take into account register requirements during loop scheduling. If, at a particular point of the parital schedule, it is detected that register pressure is high, these schemes can insert and schedule spill instructions *on-the-fly*, that is, at the same time that they schedule the original loop instructions. The main advantage to using this approach is that we may have more flexibility scheduling spill instructions. A positive side-effect will be that instructions in the original loop can be scheduled while taking into account the already scheduled spill instructions. Therefore, instruction scheduling and register spilling can interact to produce a better schedule.

To the best of our knowledge, there are only two techniques that insert spill code on the fly for modulo scheduled loops, *URACAM* [5] and *MIRS* [20].

- URACAM is an approach to performing instruction scheduling, cluster assignment and register allocation in a single phase. The main obective of this approach is to maintain a balance with critical resources (i.e., communications, register and memory) at each scheduling pass. In order to achieve this balance, a figure of merit is defined which consists on a set of percentages. Each percentage in the figure of merit is associated with a resource and is assigned based on the pressure placed on that resource during each scheduling step. This figure of merit is used to compare partial schedules, selecting the most beneficial.

  After an instruction has been scheduled, *URACAM* studies the benefits of applying spill code for each inter-use already in the partial schedule by evaluating the figure of merit. The amount of spill added at each step is only limited by the benefit that is obtained (i.e., any inter-use lifetime that produces benefits is spilled). Unlike traditional approaches, *URACAM* allows spill instructions to be scheduled at any distance from their producer/consumers.
- MIRS is a modulo scheduling algorithm with integrated on-the-fly spill code generation and back-tracking. On-the-fly spill code is inserted whenever the partial schedule reaches a point where $maxLive > 2 \cdot n\_of\_reg$. Regarding spill candidates, they are selected following the heuristics described in [19]. However, unlike in [19], the new spill instructions are not scheduled together with their producers/consumers as instruction-pairs. They have some freedom to be scheduled further apart. In particular, a spill instruction can be scheduled up to 4 cycles away from its producer/consumer. If spill code does not reduce register pressure sufficiently, then back-tracking is used and selected instructions are un-scheduled in order to reduce register pressure. In particular, the cycles in which the un-scheduled instruction is executed have to span *CC*. Thus, *maxLive* can be reduced by moving instructions, instead of by spilling a value to memory.

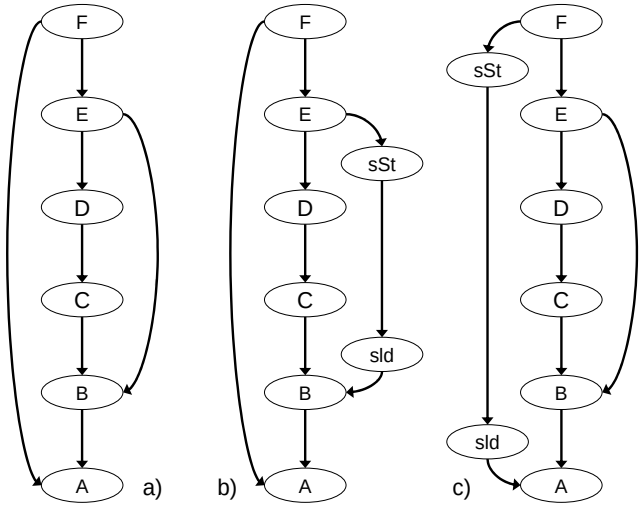


**Figure 2.** Problems with uracam

## 3. On-the-fly spill code generation

In this section we describe a new technique to generate spill code *on-the-fly* based on our study of previous approaches. In this paper, we will use *URACAM* as our reference technique. For this reason, we first discuss some areas for improvement in *URACAM*.

The reason why we chose to start with *URACAM* as our base case is because its *on-the-fly* spilling strategy is less restrictive, and thus allows us to apply optimizations such as inserting spill code, even before we detect an actual need for it.

### 3.1 Areas for improving URACAM

When inserting spill code *on-the-fly* with *URACAM* we try to anticipate register constraints and insert spill operations in advance. Thus, spill instructions can be added even if $maxLive < n\_of\_reg$. But anticipating a spill can have some drawbacks. Since the final schedule has not been computed yet, spilling is performed with limited information which can lead to making uninformed

decisions. We can see an example of a poor decision being made in figure 2. The graph on the left represents the *DDG* of a loop. According to [11], nodes in this graph will be scheduled in alphabetical order. When node *E* is scheduled, the figure of merit of *URACAM* may suggest that introducing spill code for the edge $E \rightarrow B$ (graph in the center) leads to a more balanced schedule. This may saturate memory ports. Then, when instruction *F* is scheduled, it is not possible to introduce a spill for edge $F \rightarrow A$. This may result in an invalid schedule because edge $F \rightarrow A$ has the longest inter-use lifetime.

In fact, this may be a critical drawback. In loops with high register requirements, there are often instructions at the end of the loop that depend on values generated at the beginning of the loop. However, these dependencies are not visible in a partial schedule until both the producer and the consumer have been scheduled. Since all intermediate instructions would have been scheduled before, this long dependency is not visible until a major portion of the loop has been scheduled. Thus, the memory slots may have already been used to schedule spill for uses that possess shorter inter-use lifetimes.

### 3.2 New on-the-fly spill algorithm

In this section we present a new on-the-fly spill approach that overcomes the problems we have highlighted above. The proposed scheme makes two scheduling passes. The first pass tries to schedule the loop without adding spill code. If $maxLive \leq n\_of\_reg$, then the schedule is already valid and we save compile time. If instead $maxLive > n\_of\_reg$, we re-schedule the loop with on-the-fly spill code generation. During this second scheduling pass, the spill heuristics are guided by information obtained from the first computed schedule. For that reason, instructions are scheduled following the same strategy used in the first pass so that both schedules are as similar as possible. The difference with the first pass is that after scheduling an instruction, adding spill code is considered. More specifically, spill instructions can be inserted in two cases:

1. $maxLive > n\_of\_reg$:

   If, during a partial schedule, we detect that $maxLive > n\_of\_reg$, we look for spill candidates within the current partial schedule. The fact is that the to-be scheduled instructions can only increase register pressure, so sooner or later we will need to spill some of the already scheduled values to memory.

   To schedule these spills, we use the *a posteriori* spilling scheme described in section 4 (without re-scheduling), but apply it to the already-scheduled subgraph instead of to the whole graph. Our objective here is to try to strike a compromise between inserting spill instructions late during scheduling (so that the s-loads and s-stores do not interfere with the original memory instructions). We can then insert spill instructions early during scheduling (so that spill instructions can be scheduled in the most suitable slots and then the rest of the instructions to be scheduled can take advantage of this knowledge).

   Note that if register pressure cannot be decreased enough such that $maxLive \leq n\_of\_reg$, then we can stop trying to schedule. We then will increase the *II* and re-start the scheduling process. This helps to reduce compile time.

2. Estimation of register requirements

   An important feature of on-the-fly spill code generation is that it allows for the insertion of spill code even before detecting an actual need for it so that instruction scheduling can adapt to the new situation. For that purpose, we need to anticipate that the final schedule will be register constrained. Otherwise, we could be over-spilling.

   Next, we will define some concepts that we use to anticipate that the final schedule will be register constrained. We define the lifetime of a schedule *S*, ***lftms(S)***, as the sum of the individual lifetimes for each value:

$$lftms(S) = \sum_{v \in S} lftm(v)$$

   Note that if

$$lftms(S) > II \cdot n\_of\_regs$$

   then

$$maxLive > n\_of\_regs$$

   In our spilling heuristic, we estimate ***lftms(S)*** for the final schedule. Whenever this estimation satisfies the previously mentioned relation: $lftms(S) > II \cdot n\_of\_regs$ we will anticipate that the final schedule will be register constrained and we will insert spill code.

   Hence, we need to estimate ***lftms(S)*** for the final schedule. However, some instructions have not been scheduled yet. For these instructions, we will assume that they will be scheduled in the same cycle where they were placed in the first pass schedule. We will refer to this cycle as the *estimated cycle*. Now, we can estimate the lifetimes of all instructions and therefore we can also estimate ***lftms(S)*** for the final schedule.

   Then, if $lftms(S) > II \cdot n\_of\_regs$ we will anticipate spill code for the consumers with the longest inter-use lifetimes. If both the producer and the consumer of the value are scheduled, we can schedule the spill instructions and reduce the estimation of ***lftms(S)***. If any of them is not scheduled yet, we will only reserve the memory slots required to introduce the spill instructions. In that case, ***lftms(S)*** is reduced assuming that the inter-use will be spilled for its whole estimated lifetime. When both the producer and the consumer are scheduled, we will schedule the spill instructions (if still required).

   We iterate through this process until the estimation

$$lftms(S) \leq II \cdot n\_of\_regs$$

   or until no further spill can be introduced.

   Note that this algorithm will tend to insert less spill code than may be needed since it only estimates ***lftms(S)***. Since it is only an estimate, we are conservative in order to prevent over-spilling.

After inserting spill code on-the-fly we proceed with the loop schedule. The next scheduled instructions will have more freedom in order to take advantage of the new spill code operations. More specifically, when scheduling a node we consider holding off scheduling the instruction so that it can obtain its operands from an s-load instead of from the producer operation. Then, we select the cycle that further reduces register requirements. If some node is then scheduled far from the estimated cycle, the estimation produced for the remaining nodes can also be imprecise. To prevent this from happening , if:

$$|real\_cycle(ins) - estimated\_cycle(ins)| > II$$

we re-compute the estimated cycle for the instructions that have not yet been scheduled. However, this occurs for less than 10% of the graphs that need spill code.

## 4. NoRPS

As was discussed in section 2.1, previously proposed *a posteriori* spilling schemes re-schedule a whole loop after inserting spill instructions. The reason for re-scheduling is that modulo schedules tend to be compact and so it may be difficult to find candidate slots for the insertion of new memory instructions.

In this work we propose a new scheme that adds spill code *a posteriori* without re-scheduling the loop. Spill instructions are

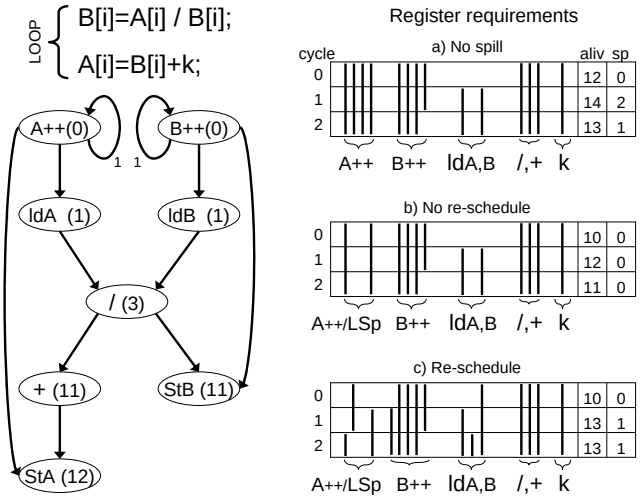


**Figure 3.** Example for spill code a posteriori.

scheduled at the point in time that they are added to the schedule (NoRPS stands for *No Re-scheduling Posteriori Spill*). In order to reduce register requirements, spill code is usually inserted for consumers whose inter-use lifetime is long (in terms of the number of cycles). Since there is considerable space between two consumers, it is almost always possible to schedule the spill instructions without re-scheduling the loop. In fact, scheduling spills without re-scheduling the loop has some advantages, as we will see. Next, we will describe an example that illustrates some of these benefits.

### 4.1 Motivating example

Assume we have an architecture with two memory ports and 12 registers (for illustration purposes only). Assume also that instructions in our example system incur the following latencies: one cycle for each add, two cycles for each memory instruction and eight cycles for each division. At the top left corner of figure 3 we show the code for a simple loop. Beneath it we present the associated data dependence graph (*DDG*). Each node in the *DDG* represents one instruction in the loop, and each edge represents a dependency between two instructions. Inside each node we describe the instruction executed, and in parentheses, the cycle where it has been scheduled (assuming an $II = 3$).

In table *a* in figure 3, we show the lifetimes of all of the values for the schedule, before considering spill code. The second column titled *aliv* represents the number of values that are alive during each cycle. For this schedule, $maxLive = 14$ (which is greater than the number of registers available), so we need to add spill code. We will select the edge $A++ \rightarrow StA$ because it has the longest inter-use lifetime. We denote loads and stores associated with spills as *sld* and *sSt*. Next, we schedule the spill instructions without re-scheduling the whole loop. We want to schedule the *sld* as close as possible to *StA*, which in this example is cycle 10 ($\equiv 1 \bmod 3$). However, both memory ports are used at modulo cycle 1 by the two loads of the original loop code iteration. Therefore, we have to move the load one cycle back in the schedule to cycle 9. For the associated *sSt*, we want to schedule it as close as possible to *A*++. Again, the first candidate is cycle 1, but we have to move *sSt* one cycle forward to cycle 2 due to resource conflicts.

The resulting register requirements of each instruction after inserting the spill instructions (without re-scheduling the loop) are shown in table *b* of figure 3. Since we have not re-scheduled the loop, the only differences correspond to the value spilled, *A*++. In cycle 2, we store *A*++ to memory. However, we still need to keep the value in a register an additional cycle longer due to the self-dependency. Therefore, *A*++ is live from cycle 0 to cycle 3. Note that moving the *sSt* from cycle 1 (where we would have liked to have placed it) to cycle 2 (where we finally placed it) has no effect on register pressure. We have also scheduled the *sld* in cycle 9. Therefore, the value produced by *A*++ does not need to be in a register from cycle 3 to cycle 9, saving 6 lifetimes, 2 per cycle of the modulo schedule. Hence, we obtain $maxLive = n\_of\_reg$. Note that for *sld*, a register is required from cycle 9 (where it is scheduled), to cycle 12 where *StA* is. For this instruction, scheduling it one cycle earlier (cycle 9) the ideal cycle (cycle 10) increases its lifetime by one cycle.

Next we will see what would happen if we re-schedule the whole loop with the new spill instructions. According to the ordering proposed in [11], priority is based on path length. The scheduling order works in a bottom-up fashion. Therefore, the ordering for the original instructions would be *StA*, +, /, *ldA*, *A*++, *ldB*, *B*++, *StB*. As we mentioned in section 2.1, when re-scheduling the loop, spill instructions are scheduled together with consumer/producer instructions as instruction-pairs. Thus, the *StA* would be scheduled again in cycle 12. The *sld* would take advantage of the re-scheduling and it would be placed closer to its consumer in cycle 10, reducing its lifetime one cycle with respect to the non-re-scheduling approach.

The / would be scheduled again in cycle 3, and the *ldA* in cycle 1. The *A*++ could be scheduled in cycle 0, but it has to be scheduled together with its store. Since both memory ports are already occupied in cycle 1, we have to move *A*++ one cycle back to cycle -1 [1], such that the *sSt* can be scheduled in cycle 0. Then the *ldB* has to be scheduled in cycle -1 because memory ports are saturated in cycles 1 and 0. Therefore, *B*++ also needs to be moved earlier in the schedule to cycle -2. Finally, the *StB* would be placed in cycle 11.

We present the register requirements of the new schedule in table *c*. As we can see, $maxLive > n\_of\_reg$. Since memory ports would be saturated, no further spilling would be possible. As a result, the II would have to be increased.

### 4.2 Analysis of the motivating example

Next, we will look in more detail at the differences between the two different *a posteriori* approaches used in the example: the re-scheduling and the non-re-scheduling approaches.

1. If the loop is re-scheduled, instruction *sld* is scheduled together with its consumer *StA*, which reduces *sld*'s lifetime by one cycle with respect to the non-re-scheduling approach. This is one of the positive consequences of re-scheduling. Spill loads are placed closer to its consumers.
2. When instruction *A*++ is re-scheduled, it is moved one cycle earlier in the schedule because it has to be scheduled together with *sSt* as an instruction pair. However, placing *sSt* closer to *A*++ does not produce any benefit on register pressure because *A*++ is alive for 3 cycles anyway due to its self-dependency. In this case, re-scheduling has not reduce register pressure. On the other hand, moving *A*++ one cycle earlier increases the length of the schedule. Hence, in this case re-scheduling has a negative effect.
3. During re-scheduling, instruction *ldB* has to be moved 2 cycles earlier from its original scheduling position because there are other memory instructions (the spill instructions) already scheduled. This increases *ldB*'s lifetime by 2 cycles. In addition, instruction *B*++ has to be moved 2 cycles later in the schedule

[1] The cycle can be negative because we are doing a modulo schedule.

too, which in turn increases *B++*'s lifetime (and the length of the schedule) by an additional two cycles.

These 3 points have illustrated the main differences between re-scheduling and not re-scheduling. First, when a re-scheduling step is performed, spill instructions are scheduled closer to their producers/consumers. In the case of an s-load, this often helps reduce their lifetimes. In the case of a store, it does not always reduce the producer-instruction lifetime because there may be another consumer scheduled between the store and the consumer for which we have spilled. Second, when re-scheduling the original memory instructions of the graph, there may already be some spill instructions scheduled, which can cause the original memory instructions to be scheduled further from their producers/consumers. This policy may be worse than moving spill instructions because it also affects other original loop instructions, and can increase the length of the schedule. The last important difference between both schemes is related to the strategy used to schedule the spill instructions during the re-scheduling step. These instructions are scheduled as instruction-pairs with its associated producers/consumers. These instruction-pairs are more complex to schedule. Therefore, re-scheduling can increase the length of the schedule and also the II.

In addition, by not re-scheduling the loop, we can also enjoy other advantages. For instance, since all of the original loop instructions are already scheduled and they will not be moved, we can compute the exact benefit of spilling a certain candidate. Therefore, we can more precisely select the instances of where to insert spill code. We define our selection heuristic in the next subsection.

### 4.3 Selecting spilling candidates

Before describing our selection policy, we first introduce some new terms to help quantify spill characteristics.

Given a particular cycle *c* of the schedule, we defined *alive(c)* as the number of live values in cycle *c*. In our previous example in figure 3, we show the value for *alive(c)* in the second column from the right in tables *a,b,c*. If $alive(c) > n\_of\_regs$ then:

$$n\_of\_spills(c) = alive(c) - n\_of\_regs$$

which indicates the number of values that need to be spilled. In the example we show the *n_of_spills* in the right-most column of tables *a,b,c*. Given a schedule *S*, *n_of_spills(S)* will be sum of the number spills found in each cycle of *S*:

$$n\_of\_spills(S) = \sum_{c=0}^{II-1} spills(c)$$

Therefore, for the example shown in figure 3, we have 3 spills for the original schedule, 0 spills after inserting spill code without re-scheduling, and 2 spills for the re-schedule.

Now we can describe the metric that will be used to determine the spill priority. First, we compute the total number of spills in the schedule $S$. Then, for each register dependency, we compute the cycle where a s-load could be scheduled and the cycle where a s-store could be scheduled. Then, for each register dependency we assume that the corresponding spill store and load have been scheduled in the previously mentioned cycles and we compute the new number of spills that would remain in the resulting schedule $S'$. We select the candidate that further reduces the number of spills, divided by the number of new memory instructions added:

$$max \left\{ \frac{n\_of\_spills(S) - n\_of\_spills(S')}{newMemTraffic} \right\}$$

Let us illustrate this metric with the example presented in figure 3. As mentioned above, the *sld* can be placed in cycle 9 and the *sSt* placed in cycle 2. Thus, we can save the lifetime between cycles 3 and 9, that is, we can reduce *alive(c)* by two in each cycle. Thus, the resulting schedule would have 10, 12, 11 live values in each cycle (table *b*), which means 0 spills. The reduction in the number of spills is therefore 3. Since we have to add both sSt and sld instructions, we assign the *value* of the metric to be 3/2. In case of a tie in this first metric (as would be the case between dependencies $A++ \rightarrow StA$ and $B++ \rightarrow StB$, we select the use with the longer inter-use lifetime.

Once the spill instructions are scheduled, we update the cycles where s-stores and s-loads can be scheduled and the metric for all uses. We iterate over this process until $maxLive \leq n\_of\_reg$. If we find we cannot further improve the schedule during this process, we re-schedule the loop with the new spill instructions. During re-scheduling, we try to schedule the spill instructions as close as possible to their producer/consumer. To achieve this goal, spill instructions are scheduled just after their associated producer/consumer have been scheduled. However, they do not have to be scheduled together as an instruction-pair. Therefore, in some cases they are separated.

In the case where re-scheduling does not produce a valid schedule, we have to increase the II. Then, all the spill instructions added are removed in the hope that by using higher II, the loop can be scheduled without adding spill code.

## 5. Experimental Evaluation

In this section, we evaluate the spill techniques described in this paper: *MIRS*, *URACAM*, *NoRPS* and *newOF* (*newOF* denotes our new scheme that inserts spill code on-the-fly). In our evaluation, we use more than 4000 loops taken from the SPECFP2000 suite. In particular, we have selected only the loops present in the 10 fortran benchmarks in the suite. We chose not to evaluate the C programs because, due to C's complex memory disambiguation, there are a significant number of memory dependencies that cause recurrences. Hence, for these programs, recurrences limit the benefits of modulo scheduling, independent of the particular register allocation and spilling technique used. To generate the *DDG*s for these loops, we have used the ORC compiler [10], with unrolling disabled, and level *O3* optimization. We have also used ORC to obtain loop execution frequencies and the average number of iterations executed for each loop.

Clustered architectures have been used in some commercial systems [17, 8, 13, 7, 9]. In this paper we consider three different clustered architecture configurations (other configurations that we tested exhibited similar trends): two that can issue up to 6 instructions per cycle and two that can issue up to 12 instructions per cycle. We also study a non-clustered (i.e., unified) configuration. In the clustered configurations, each cluster is supplied with 1 integer functional unit, 1 floating point unit, 1 memory port and 16 registers. In table 1, we present the latencies assumed for the different instructions. Next we describe the four configurations tested:

1. unified32r: A unified configuration with 2 integer functional units, 2 floating point units, 2 memory ports and 32 registers. (Issue 6).
2. 2c1b1l32r: A 2-cluster configuration with one 1-cycle latency bus for inter-cluster communication and 32 registers. (Issue 6).
3. 4c1b1l64r: A 4-cluster configuration with one 1-cycle latency bus for inter-cluster communication and 64 registers. (Issue 12).
4. 4c2b1l64r: A 4-cluster configuration with two 1-cycle latency buses for inter-cluster communication and 64 registers. (Issue 12).

All spill algorithms are evaluated using the same instruction scheduler. More specifically, we utilize the *swing modulo scheduler* [11], though other approaches could also be used. For the clustered architectures, before scheduling the loop, the *DDG* is partitioned using

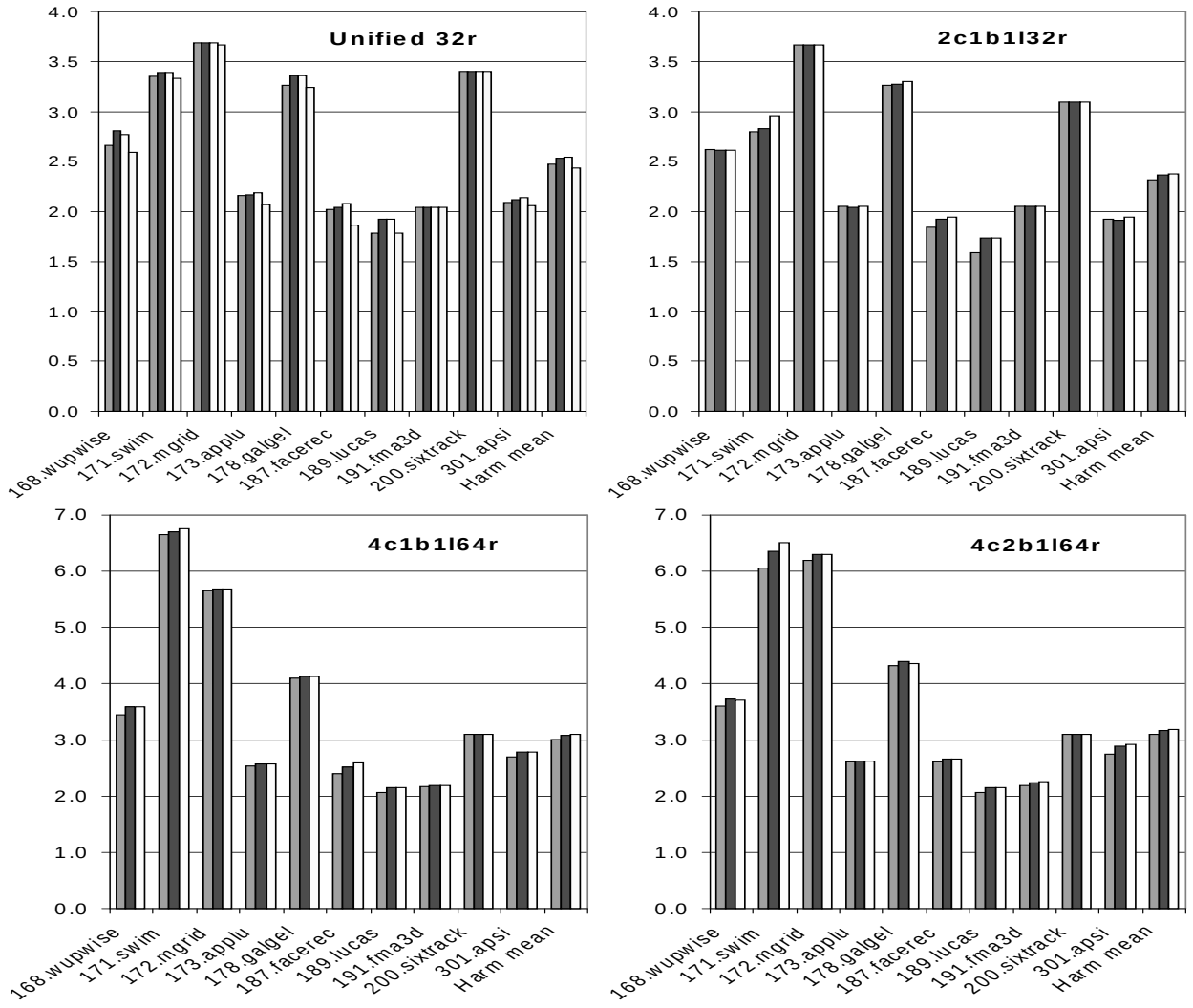


**Figure 4.** IPC for the different spill techniques and different configurations.

| *II sum* | *integer* | *fp* |
|---|---|---|
| arith | 1 | 3 |
| mem | 2 | 2 |
| mul/abs | 2 | 6 |
| div/sqrt | 6 | 18 |

**Table 1.** Latencies assumed for the instructions.

a multi-level strategy oriented towards modulo scheduling [2, 3]. For implementing *MIRS*, there are some user-defined parameters specified. For these parameters, we have used the values reported by Zalamea *et al.* [20].

### 5.1 Performance Evaluation

In figure 4, we show the IPC (average number of instructions committed per cycle) obtained for the 10 fortran SPECFP2000 benchmarks using the different spill techniques. The first important conclusion is that the two heuristics presented in this paper outperform previously proposed spilling schemes for all configurations and for all programs (except for 200.sixtrack, which is recurrence constrained and obtains almost the same performance for all spilling techniques).

With respect to *URACAM*, we only report results for the unified architecture with 32 registers. The reason why we choose not present results for the clustered configurations is that *URACAM* applies additional transformations (besides spilling) for clustered architectures that may lead to very different schedules. Hence, the differences reported may not necessarily be related to more efficient spill code generation. Therefore, the comparisons would be unfair and inconsistent. For *URACAM* equipped with 32 registers, the average speed-up of the proposed schemes (with respect to *URACAM*) is close to 5%. As we anticipated in section 2.2, the spilling algorithm used by *URACAM* often fails to spill the longest inter-uses because it has already filled all memory slots with other spill instructions. Moreover, this approach sometimes inserts spill

| *II sum* | *unified32r* | *2c1b1l32r* | *4c1b1l64r* | *4c2b1l64r* |
|---|---|---|---|---|
| mirs | 23786 | 25380 | 18749 | 17915 |
| newOF | 23695 | 25282 | 18494 | 17737 |
| norps | 23548 | 25093 | 18503 | 17732 |

**Table 2.** Sum of the II of all the loops for the different spill techniques and the different configurations.

| *SL sum* | *unified32r* | *2c1b1l32r* | *4c1b1l64r* | *4c2b1l64r* |
|---|---|---|---|---|
| mirs | 81653 | 87110 | 77005 | 76876 |
| newOF | 77196 | 82824 | 75675 | 75722 |
| norps | 77523 | 82358 | 75857 | 75618 |

**Table 3.** Sum of the length of the schedules of all the loops for the different spill techniques and the different configurations.

instructions before they are actually needed. For this technique, aggressive spill insertion sometimes results in *over-spilling*, and significantly increasing memory traffic.

With respect to *MIRS*, the differences are smaller. The average speedup of the proposed techniques is 4% for the 6-issue configurations, and 3% for the 12-issue configurations. For the 4-cluster configurations, the speed-up is slightly lower for two reasons. First, register pressure is not as critical because we use 64 registers. Second, the 4-cluster configurations also suffer from communication constraints. Since we use the same graph partition algorithm for all schedules, the differences are reduced. Another important issue to consider is that *MIRS* uses back-tracking to produce the schedule, while the two proposed techniques and *URACAM* do not. This fact improves *MIRS* schedules, at the expense of a higher compilation time. In particular, we have measured that compilation time for *MIRS* and find that it approximately doubles compilation time versus the other two schemes. The main reason why the two new schemes outperform *MIRS* is due to the metric used to select the spill candidates. In *MIRS*, spill candidates are selected according to the number of cycles between two successive consumers of the same value (without taking into account the cycle in which the spill instructions can be scheduled). Hence, this approach does not take into account the memory conflicts that will arise when scheduling the new spill instructions and relies on back-tracking to schedule them in an appropriate cycle. On the other hand, *NoRPS* and *newOF* insert spill code while taking into account the impact of the scheduling of the new instructions. Another difference is that *MIRS* only requires the spilled value to be spilled to span *CC*, whereas *NoRPS* and *newOF* use a more precise estimate, based on the impact that spilling a particular candidate will have.

Finally, when we compare the two proposed spill heuristics (*newOF* and *NoRPS*), we see that the performance differences are very small. The IPC obtained by the approaches is almost the same. For these two approaches, the spill candidates are chosen based on the same metric. Therefore, the spill instructions inserted are often the same. The main difference is that *NoRPS* inserts spill instructions a posteriori, whereas *newOF* inserts spill instructions on-the-fly. Hence, in most cases the differences between the two schedules are tied to the cycles where particular instructions are scheduled. This results in very small schedule differences. In some cases (where *maxLive* is only slightly bigger than *n_of_regs*), this small difference may lead to a better schedule for one of the techniques. In general, *NoRPS* is slightly better because, as we explained in section 4, it assigns higher priority to original memory instructions than to memory spill instructions. Nevertheless, the final IPC differs less than 1% on average. Comparing the sum of the *II's* and

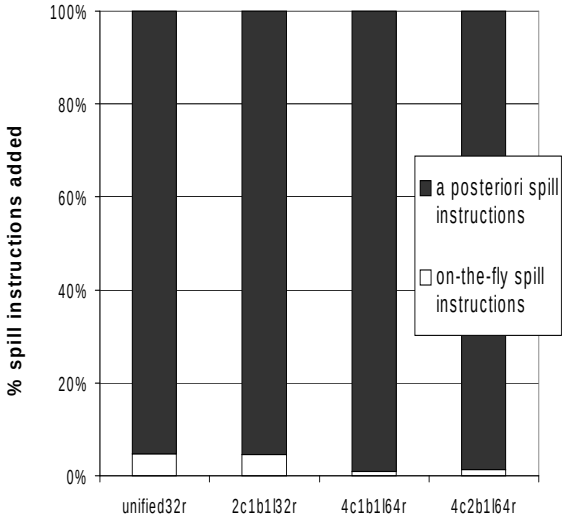


**Figure 5.** Percentage of spill instructions added on-the-fly and a posteriori with *MIRS* for the different configurations.

the sum of the lengths of the schedules obtained with *newOF* and *NoRPS* (tables 2 and 3, respectively), we can see that the differences are small. In fact, for the 4c1b1l64r configuration, *newOF* obtains better results. However, *NoRPS* performs better in some loops that have a bigger impact on IPC and so performance is higher for *NoRPS*.

The main conclusion here is that *a posteriori* spilling can provide a spill code picture as accurate as on-the-fly spilling. This result was a bit unexpected.

If we carefully study the *MIRS* approach, we will see that it is very similar to an *a posteriori* spill technique. To further support this conclusion, we present figure 5. We will refer to spill instructions inserted during loop scheduling as *on-the-fly spill instructions*, whereas the spill instructions inserted after scheduling the original loop instructions will be referred to as *a posteriori spill instructions*. (Since *MIRS* includes back-tracking, some original loop instructions may later be un-scheduled. However, we consider a spill instruction to be an *a posteriori* spill instruction if it is inserted after having produced a schedule where all the original instructions of the loop are scheduled). As we can see in figure 5, the percentage of on-the-fly spill instructions is low; approximately 90% of the spill instructions are inserted *a posteriori*. In order to introduce on-the-fly spill instructions, *MIRS* requires that $maxLive > 2 \cdot n\_of\_reg$ for a given partial schedule. This means that when spill instructions are added on-the-fly, register pressure is very high for the partial schedule. With added register pressure, it may be more difficult to produce a valid schedule, even if we insert spill code. In fact, when we reach $maxLive > 2 \cdot n\_of\_reg$, we often have to increase the II. Therefore, most of the spill instructions added by *MIRS* are added once a first complete schedule of the original loop has been produced.

Previous *a posteriori* approaches did not perform as well as *on-the-fly* schemes. We will further explore the causes for this in the next sub-section.

### 5.2 Further analysis of a posteriori spilling

As previously discussed in section 4, the key novelty of *NoRPS* with respect to other *a posteriori* approaches is that a loop is not rescheduled after inserting spill code instructions. We maintain that

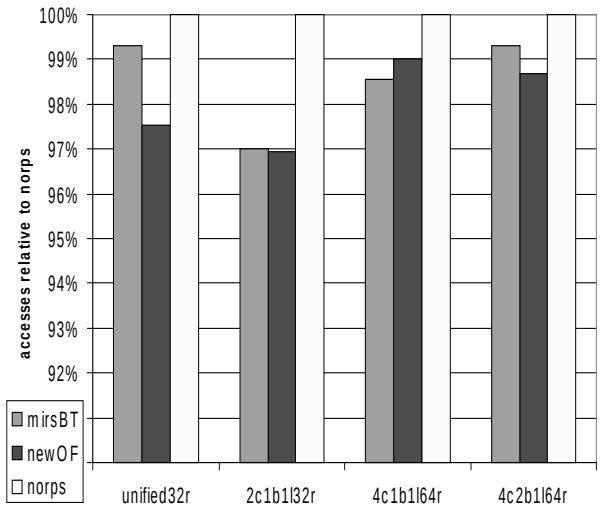


**Figure 6.** Average for all the programs of the ratio of dynamic memory accesses with respect to *NoRPS* for the different spill schemes and the different configurations.

most of the time it is possible to schedule spill instructions without re-scheduling the whole loop because the spill instructions are usually inserted for long lifetimes. To check whether this intuition holds, we computed the percentage of loops for which a valid schedule was obtained without re-scheduling over the total number of loops that needed spill code. We have observed that for 90% of the instances where spill code was inserted, the loop did not need to be re-scheduled. Therefore, re-scheduling is not essential.

We obtain a few advantages by not re-scheduling. First, the resulting schedules are shorter because the instructions have been scheduled with fewer conflicts. However, for modulo scheduling, the length of the schedule has a limited impact on performance. Another positive consequence of not re-scheduling was in the example shown in figure 3 in section 4. If the insertion of spill instructions cause non-spill memory instructions present in the loop to be scheduled later, this can increase lifetimes for the original memory instructions, and may also increase the lifetimes of other instructions in the loop. Finally, the metrics used to insert spills are more precise when no re-scheduling is performed because they can exactly measure the impact of a particular spill insertion.

On the other hand, if we do not re-schedule, we may find it more difficult to find a suitable slot to schedule a spill instruction. For that reason, in case a valid schedule has not been found, *NoRPS* allows for re-scheduling to be performed. However, the re-scheduling algorithm is different from previous *a posteriori* approaches. In previous schemes, spill instructions were scheduled together with their producers/consumers as instruction-pairs. This makes finding an appropriate slot more difficult and the II may have to be increased due to resource conflicts. Using our modified re-scheduling algorithm, we try to place spill instructions as close as possible to their producers/consumers, but not necessarily back-to-back.

### 5.3 Memory traffic

In figure 6, we compare the memory traffic of the schedules generated with the different spill schemes for each configuration. In particular, we have measured the ratio of dynamic memory accesses with respect to the number of dynamic memory accesses produced with the *NoRPS* approach. In the figure we present the averaged ratios of the programs evaluated. The main conclusion is that the differences are small (in most of the cases around 1% and for all the cases under 3%).

The spill technique that produces the greatest number of memory accesses is *NoRPS*. This is due to the fact that, in general, this spill heuristic achieves a lower *II*. This comes at the expense of inserting more spills. However, for the 4c1b1l64r configuration, the sum of the *II* of *newOF* was lower than for *NoRPS*, and the number of memory accesses is still smaller for *newOF*. The *newOF* technique produces fewer memory accesses because when spills are inserted on-the-fly, the spill instructions can be scheduled closer to their producer/consumers. Therefore, shorter lifetimes are produced and fewer spills are required. In fact for 3 of the 4 configurations reported, *newOF* generates the least memory traffic.

## 6. Conclusions

In this work, we have described two new spill code generation schemes. Our new proposals are based on a thorough study of the benefits and drawbacks of previous techniques for both *on-the-fly* and *a posteriori* spill approaches. Our new schemes outperform previous state-of-the-art techniques for register spilling. However, one unexpected result in this work is that we found that our scheme that inserts spill code a posteriori and our scheme that inserts spill on-the-fly obtain almost the same performance. In recent work, on-the-fly techniques were reported to obtain better performance. We have studied the reasons why the proposed *a posteriori* scheme achieve similar schedules than on-the-fly, while previous proposals did not. The key difference is that a loop is not re-scheduled after spill code is inserted. We have shown that this has some advantages. The most important benefit is that it allows for a very precise estimation of the effects that spilling a selected use can have. Hence, the metric used provides more accurate spilling. The proposed on-the-fly algorithm uses the same metric. However its performance is slightly lower than the *a posteriori* scheme, due mainly to the benefits obtained by scheduling spill instructions at the end.

## 7. Acknowledgements

This work was partially supported by the Spanish Ministry of Education and Science under contract TIN2004-03072 and Feder funds, by Intel, by CenSSIS the Center for Subsurface Sensing and Imaging Systems, under the Engineering Research Centers Program of the NSF (Award Number EEC-9986821), and by the Institute for Complex Scientific Software at Northeastern University.

## References

[1] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger. Clock rate versus ipc: the end of the road for conventional microarchitectures. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 248–259, June 2000.

[2] A. Aletà, J. M. Codina, J. Sánchez, and A. González. Graph partitioning based instruction scheduling for clustered processors. In *Proceedings of the 34th International Symposium on Microarchitecture*, pages 150–159, December 2001.

[3] A. Aletà, J. M. Codina, J. Sánchez, A. González, and D. Kaeli. Exploiting pseudo-schedules to guide data dependence graph partitioning. In *Proceedings of the International Conference on Parallel Architectures and Compiler Techniques*, pages 281–290, September 2002.

[4] G. Chaitin. Register allocation and spilling via graph coloring. *ACM SIGPLAN Notices*, 39(4):66–74, April 2004.

[5] J. M. Codina, J. Sánchez, and A. González. A unified modulo scheduling and register allocation technique for clustered processors. In *Proceedings of the International Conference on Parallel Architectures and Compiler Techniques*, pages 175–184, September 2001.

[6] J. Dehnert, P. Hsu, and J. Bratt. Overlapped loop support in the cydra 5. In *Proceedings of the 3rd International conference on architectural support for programming languages and operating systems*, pages 26–38, April 1989.

[7] P. Faraboschi, G. Brown, J. Fisher, G. Desoli, and F. Homewood. Lx: a technology platform for customizable vliw embedded processubg. In *Proceedings of the 27th international symposium on computer architecutre*, June 2000.

[8] J. Fridman and Z. Greenfield. The tigersharc dsp architecture. *IEEE Micro*, pages 66–76, January-february 2000.

[9] P. Glaskowsky. Map1000 unfolds at equator. *Microprocessor report*, 12(16), December 1998.

[10] R. Ju, S. Chan, T.-F. Ngai, C. Wu, Y. Lu, and J. Zhang. Open research compiler (orc) 2.0 and tuning performance on itanium. *Presented at the 35th International Symposium on Microarchitecture*, December 2002.

[11] J. Llosa, E. Ayguadé, A. González, and M. Valero. Swing modulo scheduling. In *Proceedings of the International Conference on Parallel Architectures and Compiler Techniques*, September 1996.

[12] J. Llosa, M. Valero, and E. Ayguadé. Heuristics for register-constrained software pipelining. In *Proceedings of the 29th International Symposium on Microarchitecture*, pages 250–261, December 1996.

[13] G. G. Pechanek and S. Vassiliadis. The manarray embedded processor architecture. In *Proceedings of the 26th Euromicro Conference: Informatics: inventing the future*, September 2000.

[14] B. Rau, M. Lee, P. P. Tirumalai, and M. S. Schlansker. Register allocation for software pipelined loops. In *Proceedings of the conference on Programming language design and implementation*, pages 283–299, June 1992.

[15] B. R. Rau. Iterative modulo scheduling: an algorithm for software pipelining loops. In *Proceedings of the 27th annual international symposium on Microarchitecture*, pages 63 – 74, November 1994.

[16] J. Ruttenberg, G. R. Gao, A. Stoutchinin, and W. Lichtenstein. Software pipelining showdown: optimal vs. heuristic methods in a production compiler. In *Proceedings of the conference on programming language design and implementation*, pages 1–11, May 1996.

[17] Texas Instruments Inc. *TMS320C62x/67x CPU and instruction set reference guide*, 1998.

[18] J. Wand, A. Krall, M. A. Ertl, and C. Eisenbeis. Software pipelining with register allocation and spilling. In *Proceedings of the 27th International Symposium on Microarchitecture*, pages 95–99, November 1994.

[19] J. Zalamea, J. Llosa, E. Ayguadé, and M. Valero. Improved spill code generation for software pipelined loops. In *Proceedings of the conference on programming language design and implementation*, pages 134–144, June 2000.

[20] J. Zalamea, J. Llosa, E. Ayguadé, and M. Valero. Mirs: Modulo scheduling with integrated register spilling. In *Proceedings of the 14th workshop on languages and compilers for parallel computing*, august 2001.