Heuristics for Register-constrained Software Pipelining *

Josep Llosa, Mateo Valero and Eduard Ayguadé

Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya
Campus Nord, Mòdul D6, Gran Capità s/n
08071, Barcelona, SPAIN
{josepll,mateo,eduard}@ac.upc.es

Abstract

Software Pipelining is a loop scheduling technique that extracts parallelism from loops by overlapping the execution of several consecutive iterations. There has been a significant effort to produce throughput-optimal schedules under resource constraints, and more recently to produce throughput-optimal schedules with minimum register requirements. Unfortunately even a throughput-optimal schedule with minimum register requirements is useless if it requires more registers than those available in the target machine.

This paper evaluates several techniques for producing register-constrained modulo schedules: increasing the initiation interval (II) and adding spill code. We show that, in general, increasing the II performs poorly and might not converge for some loops. The paper also presents an iterative spilling mechanism that can be applied to any software pipelining technique and proposes several heuristics in order to speed-up the scheduling process.

1. Introduction

Software pipelining [10] is an instruction scheduling technique that exploits the ILP of loops by overlapping the execution of successive iterations of a loop. There are different approaches to generate a software pipelined schedule for a loop [1]. Modulo scheduling is a class of software pipelining algorithms that rely on generating a schedule for an iteration of the loop such that when this same schedule is repeated at regular intervals, no dependence is violated and no resource usage conflict arises. Modulo scheduling was proposed at the beginning of the 80s [25]. Since then, many research papers have appeared on the topic [19, 18, 17, 29, 27, 31, 22], and it has also been incorporated into some production compilers (e.g. [24, 12]).

Exploiting more ILP results in a significant increase in the register pressure [23, 21]. The register requirements of a schedule are of extreme importance for compilers since any valid schedule must fit in the available number of registers of the target machine. Some practical modulo scheduling approaches use heuristics to produce near-optimal schedules with reduced register requirements [17, 22]. Other approaches try to reduce the register requirements of the schedules by applying a post-pass process [13, 28]. However, even a throughput-optimal schedule with minimum register requirements is useless if it requires more registers than the target machine has. In addition, as shown in [21], loops with high register requirements represent a significant amount of the execution time of the programs. When there is a limited number of registers and the register allocator fails to find a solution with the number of registers available, some additional action must be taken. Different alternatives to fit the register requirements of a modulo scheduled loop in the available number of registers were outlined in [26].

One of the options, used by the Cydra 5 compiler [12], is to reschedule the loop with an increased *II*. If, after several trials, the compiler is unable to find a valid schedule requiring less registers than available, the compiler schedules the loop using local scheduling techniques, i.e. without performing modulo scheduling. Rescheduling the loop with a bigger initiation interval (see Figure 1a) usually leads to schedules with less iteration overlapping, and therefore with less register requirements. Unfortunately, the register reduction is at

*This work has been supported by the Ministry of Education of Spain under contract TIC 429/95, and by CEPBA (European Center for Parallelism of Barcelona).

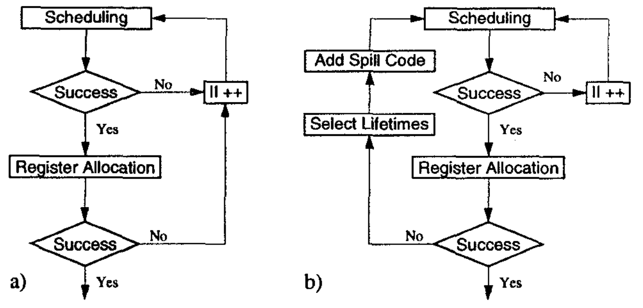

Figure 1. Flow diagram of the scheduling process with register reduction by: a) Increasing the II. b) Adding spill code.

the expense of a reduction in performance (less parallelism is exploited).

Another option is to spill some variables to memory, so that they don't waste registers for a certain number of cycles. For acyclic schedules, graph coloring [9] has been widely used to perform register allocation and adding spill code. Most subsequent research in the field of register allocation and spilling has mainly focused on improving heuristics for graph coloring [4, 6, 8, 7]. However, software pipelining has some constraints that hinder the use of traditional techniques for register allocation and spilling. For instance, lifetimes may cross the boundary of iterations and even can be longer than the II, interfering with themselves (in next iterations). Finally, modulo schedules are very compact —the goal is to saturate the most used resource— which complicates the addition of spill load/store operations without affecting the whole schedule.

Cyclic interval graphs [16] can deal with lifetimes that cross loop boundaries. Other register allocation heuristics have been proposed [26, 14], that deal with lifetimes larger than the II (in the presence of hardware support). However, none of these works deal with the addition of spill code (and its scheduling) for software pipelined loops. Software pipelining and spill code has been first considered (to the best of our knowledge) in [30], where spill load/store operations are added before scheduling the loop if, and only if, doing so does not increases the initiation interval.

In this paper, we treat the scheduling problem in a more general framework: generate modulo schedules with register constraints (as well as resource constraints). The above outlined alternatives (increasing the II and adding spill code) to schedule a loop with a limited number of registers are considered.

We show that increasing the II produces, in general, worse schedules (in terms of II) than adding spill code. In addition, we show that increasing the II might not converge to a solution (i.e. find a schedule requiring less than the available registers) for some loops. Also, it turns out that these loops represent a significant part of the execution time.

We propose an iterative algorithm for adding spill code to modulo schedules (see Figure 1b). If a loop requires more registers than available, we add spill code and reschedule the loop until a schedule requiring no more registers than available is found. Rescheduling is necessary, since the added load/stores might not fit in the schedule. The spilling method we propose, uses several heuristics in order to be a general method useful for any software pipelining mechanism (including methods that do not care about register requirements). In addition we also propose several heuristics to accelerate the iterative algorithm, so that the computational time required to produce the (register and resource constrained) schedules is small.

To evaluate the proposals of this paper, we use a register sensitive scheduling technique (HRMS [22]) as the core scheduler. In any case the techniques presented can also be used with other scheduling techniques. The evaluation has been performed using more than one thousand loops from the Perfect Club [5].

In Section 2 we make a brief overview of modulo scheduling, register allocation for modulo scheduling, and divide the lifetimes into two components that behave differently. Then we present the two approaches for register-constrained software pipelining: increase the II (Section 3) and add spill code (Section 4) introducing several heuristics for selecting the variables to spill, to schedule the spill operations and to accelerate the whole process. In Section 5 the different alternatives and heuristics are evaluated in terms of performance of the loops and scheduling time. Finally Section 6 states our conclusions.

2. Modulo Scheduling

2.1. Definitions

A loop is represented by a *dependence graph* $G = DG(V, E, \delta)$ where:

- V is the set of vertices of the graph G. A vertex (node) $v \in V$ represents an operation of the loop body.
- E is the dependence edge set. An edge $e = (u, v) \in E$ represents a dependence between two operations u, v.
- δ is a mapping $\delta : E \mapsto N$ that represents the *dependence distance* or *dependence weight*.

There is a dependence of distance δ_e associated to each edge $e = (u, v) \in E$ if the execution of operation v at iteration $i + \delta_e$ depends on the execution of operation u at iteration i.

The dependences can be of several types.

- *Control* dependences indicate that the target operation is executed (or not) depending on the outcome of the source operation.
- *Data dependences* between operations indicate accesses to the same storage location. For the purposes of this work, they have been further divided into *memory* ($MemE \subseteq E$) and *register* ($RegE \subseteq E$) data dependences. *Memory* data dependences are caused by accesses to the same memory location. *Register* data dependences are caused by accesses to the same registers. Since register allocation is performed after scheduling, only flow register data dependences are considered.

2.2. Overview of Modulo Scheduling

In a software pipelined loop, the schedule for an iteration is divided into stages so that the execution of consecutive iterations that are in distinct stages is overlapped. The number of stages in one iteration is termed **stage count**(*SC*). The number of cycles per stage is the initiation interval (*II*).

The execution of a loop can be divided into three phases: a ramp up phase that fills the software pipeline, a steady state phase where the software pipeline achieves maximum overlap of iterations, and a ramp down phase that drains the software pipeline. During the steady state phase of the execution, the same pattern of operations is executed in each stage. This is achieved by iterating on a piece of code, termed the **kernel**, that corresponds to one stage of the steady state phase.

The initiation interval *II* between two successive iterations is bounded either by loop-carried dependences in the graph (*RecMII*) or by resource constraints of the architecture (*ResMII*). This lower bound on the *II* is termed the **Minimum Initiation Interval** (*MII*= max (*RecMII*, *ResMII*)). The reader is referred to [12, 27] for an extensive dissertation on how to calculate *ResMII* and *RecMII*.

As an example consider the loop body of Figure 2a, whose optimized data dependence graph is shown in Figure 2b. For simplicity we assume (in this example) that all operations have a latency of two cycles, there are 4 general purpose functional units, and that each operation is fully pipelined and executes in one functional unit. Figure 2c shows a schedule for an iteration

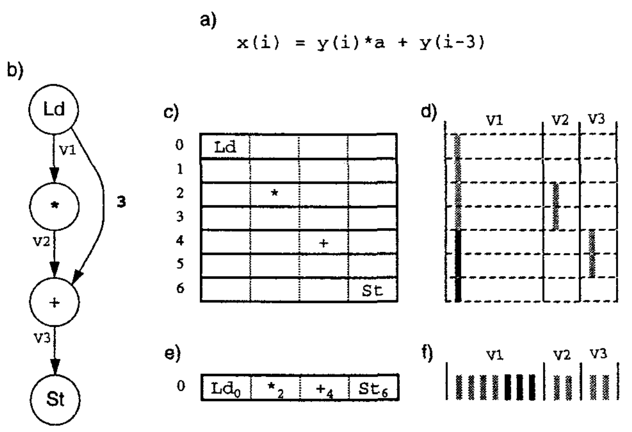

Figure 2. Example loop: a) Fortran code of the loop body. b) Optimized data dependence graph. c) Schedule of the loop. d) Lifetimes of one iteration. e) Kernel code. f) Register requirements (the shaded part is the scheduling component of the lifetimes).

with an initiation interval $II = 1$. In this case, the schedule is constrained only by resources (since it has no recurrences). We can initiate an iteration every cycle, since the loop requires 4 resources per iteration, and we have 4 available resources. Figure 2e shows the kernel code of the loop. In the kernel code, subindex of operations indicate the stage they belong to in the schedule of a single iteration of the original loop.

2.3. Register Requirements

Values used in a loop correspond either to loop-invariant variables or to loop-variant variables. Loop-invariants are repeatedly used but never defined during loop execution. Loop-invariants, have a single value for all the iterations of the loop and therefore they require one register each regardless of the scheduling and the machine configuration.

For loop-variants, a value is generated in each iteration of the loop and, therefore, there is a different value corresponding to each iteration. Because of the nature of software pipelining, lifetimes of values defined in an iteration can overlap with lifetimes of values defined in subsequent iterations. Figure 2d shows the lifetimes for the loop-variants corresponding to every iteration of the example loop. Lifetimes of loop-variants can be measured in different ways depending on the execution model of the machine. We assume (either in the examples and in the experiments) that a variable is alive from the beginning of the producer operation, until the start of the last consumer operation.

By overlapping the lifetimes of the different iterations, a pattern of length II cycles that is indefinitely repeated is obtained. This pattern (Figure 2f) indicates the number of values that are live at any given cycle (11 in this example).

As it is shown in [26], the maximum number of simultaneously live values (*MaxLive*) is an accurate approximation of the number of registers required by the schedule [1]. In all the examples of this paper, the register requirements of a given schedule are approximated by *MaxLive*, for simplicity. However, in Section 5 we measure the actual register requirements after register allocation.

Values with a lifetime greater than II pose an additional difficulty since new values are generated before previous ones are used. One approach to fix this problem is to provide some form of register renaming so that successive definitions of a value use distinct registers. Renaming can be performed at compile time by using modulo variable expansion (MVE) [20], i.e., unrolling the kernel and renaming at compile time the multiple definitions of each variable that exist in the unrolled kernel. A rotating register file can be used to solve this problem without replicating code by renaming different instantiations of a loop-variant at execution time [11]. In this paper we assume the presence of rotating register files.

2.4. Components of Lifetimes

Two components can be distinguished in the lifetime LT_u of a loop-variant u:

- A *scheduling component* $LTSch_u = t_v - t_u$ is caused by the scheduling distance in cycles between the producer u and the last consumer v.
- A *distance component* $LTDist_u = \delta_{(u,v)} \times II$ caused by the dependence distance $\delta_{(u,v)}$ between the producer u and the last consumer v. This component appears only in loop carried dependencess (i.e. when $\delta_{(u,v)} > 0$).

For instance in loop-variant V1 (Figure 2d) the shaded part corresponds to the scheduling component, $LTSch_{V1} = t_+ - t_{Ld} = 4$, of the lifetime and the dark part corresponds to the distance component $LTDist_{V1} = \delta_{(Ld,+)} \times II = 3$. Notice that not all the parts must be present in the lifetime, for instance, the lifetime of loop-variants V2 and V3 has only the scheduling component. These two components will have different effects when reducing the register requirements of loops.

[1]For an extensive discussion on the problem of allocating registers for software-pipelined loops refer to [26]. The strategies presented in that paper almost always achieve the *MaxLive* lower bound. In particular, the wands-only strategy using end-fit with adjacency ordering almost never required more than *MaxLive* +1 registers.

3. Increasing the Initiation Interval

One of the approaches considered in this paper, is to reschedule the loop with an increased II. Register pressure is, to some extent, proportional to the number of concurrently executed iterations. Increasing the II, in general, reduces the number of stages in which the schedule is divided, and therefore, the number of simultaneously overlapping iterations. In addition, the scheduling component of the lifetimes can be shorter because a larger II imposes less resource constraints than a smaller one. Unfortunately, increasing the II reduces the register requirements at the expense of reducing the throughput of the schedule.

As an example, consider the loop of Figure 2 that, when scheduled with $II = 1$, required 11 registers for loop-variants. Figure 3a shows the same loop scheduled (for the same architecture) with $II = 2$. Notice that the scheduling components of the lifetimes (Figure 3b) have the same length in cycles as in the schedule of Figure 2 but due to the smaller number of overlapping iterations (4 instead of 7) these components require less registers. Unlike the scheduling component, the distance component of the lifetimes increases with the II since it is proportional to it. For instance $LTDist_{V1}$ has increased from 3 cycles to 6 cycles. Figure 3d shows the register requirements of the schedule with $II = 2$. The new schedule requires 7 registers instead of 11. Notice that only the scheduling component of the lifetimes has reduced the register requirements, while the distance component requires the same number of registers.

3.1. Behavior of Loops when Increasing the Initiation Interval

Increasing the II might produce an extremely inefficient code, especially if the loops require many more registers than available, since the register requirements are reduced at the expense of increasing the number of cycles per iteration. Figure 4a shows the register requirements of a loop from the Perfect Club when the II is increased. This loop requires 54 registers when scheduled with an optimal II of 7 cycles. To schedule the loop with 32 registers the II must be increased from 7 to 13 cycles, and performance is reduced to 53% of the original loop. If the loop is scheduled with 16 registers the initiation interval has to be increased up to

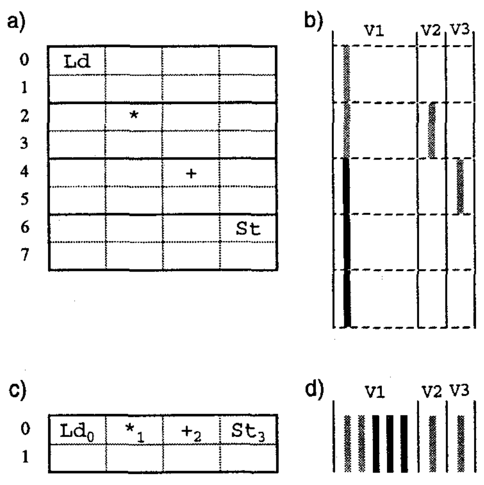

Figure 3. Example loop of Figure 2a scheduled with $II = 2$ to reduce register pressure: a) Schedule of the loop. b) Lifetimes of one iteration. c) Kernel code. d) Register requirements.

31 cycles, reducing performance to 22% of the original loop.

An additional problem with this technique is that, for some loops, it might not converge to the available number of registers. Figure 4b shows the behavior of a loop that, even though requiring only 55 registers (one more than the previous loop), it can not be scheduled with 32 registers. The loop achieves a permanent situation requiring 41 registers.

There are several reasons, mainly related with the "topology" of the DG, why a loop might not decrease the register requirements when the initiation interval is increased:

- The distance component of the loop-variants, since their lifetime increases with the initiation interval and the registers required never decrease however much the II is increased. For instance in loop 50 of APSI the distance component of loop variants acounts for 22 registers.
- The lifetime of loop-invariants is always II cycles. Therefore, they require one register each, independently of the schedule.
- The dependence graph might require more registers than available even using acyclic scheduling techniques (i.e. without overlapping iterations).

[2]The first loop of subroutine CPADE of the program APSI (ADM).

[3]The second loop of subroutine PADEC of the program APSI (ADM).

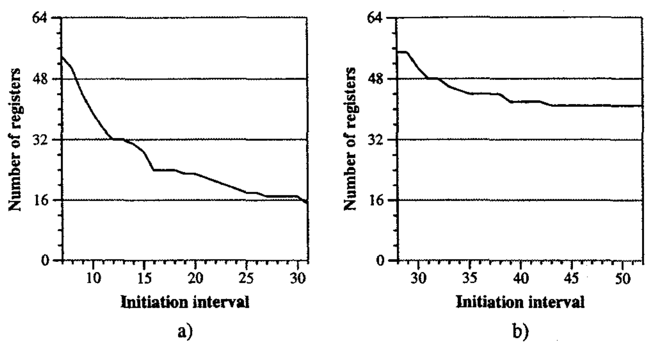

Figure 4. Behavior of the register requirements of two real loops when increasing the II: a) A loop that converges (loop 47 [2] of APSI). b) A loop that does not converge (loop 50 [3] of APSI).

4. Adding Spill Code

The other approach considered in this paper consists of spilling some lifetimes to reduce the register requirements. Due to some characteristics of software pipelining, traditional spilling techniques are difficult to apply. For instance, interference graphs cannot be used if the lifetimes are larger than the II (unless MVE is performed). Also, the addition of the spill code is difficult, since the schedules are very compact, requiring a heavy rescheduling of the loop. The option we investigated consists of the following steps:

- Select the appropriate lifetimes to spill.
- Add the required spill code for them.
- Perform modulo scheduling again.
- Repeat the register allocation.

Figure 1b shows the flow diagram of the scheduling process, adding spill code to reduce register pressure when required.

Rescheduling is necessary because the addition of loads and stores produces a different dependence graph requiring a different schedule. Unfortunately, it is possible in the new schedule that a different set of lifetimes might need to be spilled than those that have already been spilled.

4.1. Selecting Lifetimes to Spill

We have investigated two heuristics for selecting the lifetimes to be spilled.

Spill the longest lifetime regardless of the cost (in terms of additional operations required to add the associated spill code). The intuition is that large lifetimes free more registers. This criterion for selecting lifetimes is dubbed $Max(LT)$. In general, the largest lifetime will be larger than the II, so spilling the largest lifetime will free several registers at every cycle including the cycle with maximum register pressure.

Spill the largest $Lifetime/Cost$ is similar to the previous one, but with the improvement that the cost (in terms of load and store instructions) of adding spill code is contemplated when selecting the lifetime to be spilled out. The value selected is the one with the highest relation $\frac{Lifetime}{Cost}$ where $Cost$ is determined by the number of additional memory operations. This criterion for selecting lifetimes is dubbed $Max(LT/Traf)$.

The number of additional memory operations depends on the number of consumers the value has, on whether the value is a loop-variant or a loop-invariant, and if some of the additional memory operations are redundant and can be removed from the dependence graph.

4.2. Adding Instructions for Spilling Lifetimes

Once a lifetime has been selected for being spilled, it is required to add the appropriate load and store instructions in the dependence graph, so that the selected lifetime is stored in memory after being produced, and reloaded before being consumed.

For each lifetime being spilled the following actions are performed:

- The set of edges $\subseteq RegE$ akin to the spilled lifetime is removed.

- One store is added just after the producing operations and a new edge $e \in RegE$ is added from the producer to the new store.

- One load is added just before each use of the spilled result and a new edge $e \in RegE$ is added between each one of the loads and the related consumer.

- A new edge $\in MemE$ is added between the store and each of the loads.

- The weights δ_e of each of the edges $e \in RegE$ added between the producer operation or between the loads and their respective consumer operations are set to zero. Likewise, the weights of the added edges from the store to the loads is set to the original weights that had the associated removed edges.

As an example, Figure 5b shows the resulting graph when spilling out value V1. Notice that the original edges have been removed. A new store, 'Ss', spills the value to memory and two new loads, 'Ls1' and 'Ls2', reload the value when required. Observe that by assigning the distance of the original edge (Ld,+) to the new memory edge (Ss,Ls2) the new values V11, V12, and V13 do not have a distance component in their lifetimes.

Up to now we have outlined the general process of adding spill operations, but there are cases where it is not required to add all the operations. Several particular cases have been taken into account:

- If the producer operation is a load (as in the example of Figure 5), adding spill code in the former way yields to suboptimal code with redundant operations. It is not necessary to store the result of the load since the corresponding value is already in a memory location and it can be loaded when required. In our example the resulting graph is depicted in Figure 5c.

- If at least one of the successors of the producer is a store, it is not necessary to add a new store to spill out the result.

- Finally, loop-invariants can also be selected for being spilled. In this case it is assumed that the store to spill out the value is executed before entering the loop.

4.3. Scheduling Spill Operations

Once the spill code has been added (including optimizations) the loop has to be re-scheduled. The objective of the spill code is to decrease the register requirements regardless of the scheduling method.

A situation that must be resolved is the likelihood of deadlocks. For instance, consider the Figure 5c. In a register insensitive scheduler, loop-variant V13 might have the largest lifetime and be selected for being spilled out. Notice that if spill code is added for this loop-variant, and the graph is optimized it results in a graph equivalent to the one of Figure 5c, which still requires spill code, leading to a deadlock situation.

To avoid this kind of deadlocks, the new loop-variants that appear because of spill operations are marked as *non-spillable.* For instance loop-variants V12 and V13 in the example of Figure 5c will be *non-spillable.* Therefore if the register requirements must be further reduced, a different lifetime must be selected.

Another situation that can appear is that, for a particular scheduling method, the register requirements

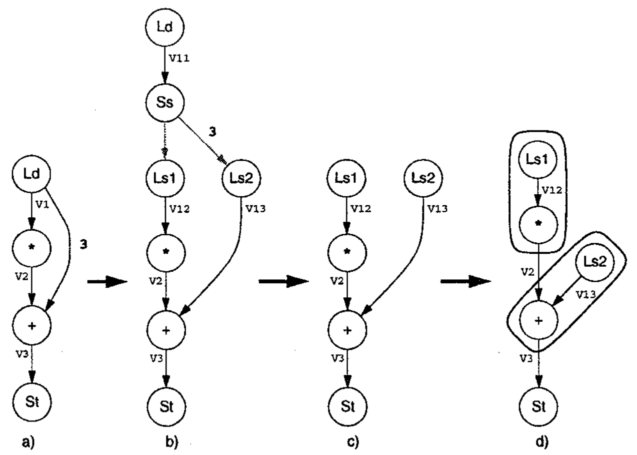

Figure 5. Adding spill code to the example loop of Figure 2a to reduce register pressure: a) DG of the original loop. b) DG after spilling lifetime V1. c) DG after optimizing redundant loads and stores. d) Operations scheduled closely to guarantee convergence.

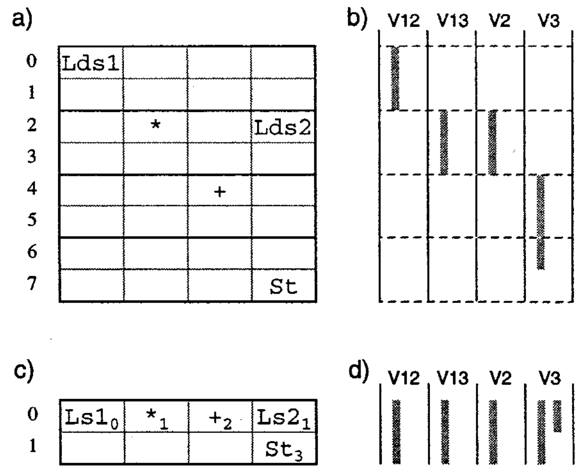

Figure 6. Schedule of loop of Figure 2a after adding spill code (Figure 5d): a) Schedule of the loop. b) Lifetimes of one iteration. c) Kernel code. d) Register requirements.

are not decreased, but increased. For instance, in the resulting graph of Figure 5c the scheduling step can schedule nodes 'Ls1' and 'Ls2' too far away from their respective successors, increasing the lifetime of loop-variants V12 and V13. This situation can result in a combined register pressure of loop-variants V12 and V13 bigger than the original loop-variant V1.

To prevent this case, operations connected by a *non-spillable* edge are forced to be simultaneously scheduled as a single "complex operation". Figure 5d shows the operations that must be scheduled as a "complex operation". According to this, operation $*$ must be scheduled exactly 2 cycles after operation 'Ls1' and operation '+' 2 cycles after operation 'Ls2'.

Figure 6a shows the resulting schedule. Notice that, even though operation 'Ls2' alone can be scheduled at cycle 0, operation '+' cannot be scheduled at cycle 2 —where it is forced to be scheduled if operation 'Ls2' is scheduled at cycle 0. An analogous situation occurs if 'Ls2' is scheduled at cycle 1. Consequently, 'Ls2' is forced to be scheduled at cycle 2 because operation '+' cannot be scheduled before cycle 4.

Figure 6b shows the lifetimes of loop-variants and Figure 6d shows the register requirements of this schedule. Notice that only 5 registers are required in contrast to 7 registers required if the *II* is increased by one. The lower register requirements are mainly due to the fact that the spill code has eliminated the distance component of the lifetime of loop-variant V1. It is also remarkable that the additional operations provoke an increase of the initiation interval (in this example the *II* of the spilled loop is also 2 cycles), which is a supplementary contribution to the reduction of the register requirements.

4.4. Behavior of Loops when Spilling Values

The addition of spill code to a dependence graph along with a reduction of the register requirements has the following negative effects:

- The memory traffic grows due to the additional memory operations.
- The *MII* can also augment if memory busses are saturated —or roughly saturated.
- Even for an optimal technique, it is difficult —and sometimes impossible— that the initiation interval reaches *MII*. The way in which spill operations are scheduled —i.e. like complex operations— prevents the scheduler from obtaining better schedules.

Figure 7 shows the evolution of the register requirements, *II*, *MII* and % of memory traffic as loop-variants are spilled, for the loops APSI 47 and APSI 50. For this example lifetimes have been selected using the $Max(LT)$ criterion.

For instance, notice in Figure 7a, that to schedule the loop APSI 47 with 32 registers, four lifetimes must be spilled out, and the *II* increases up to 14 cycles.

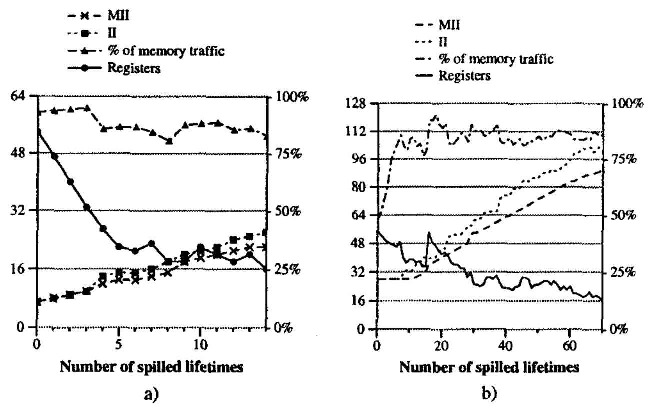

Figure 7. Behavior of the register requirements, *MII*, *II* and memory traffic of real loops when adding spill code: a) Loop 47 of APSI. b) Loop 50 of APSI.

Notice that, in this case, merely increasing the *II* allows this loop to be scheduled with 32 registers and an *II* of 13 cycles. Despite that in some particular cases increasing the *II* can be as good, or even better, as adding spill code, in general, adding spill code produces better schedules. For instance, by adding spill code, the loop APSI 47 can be scheduled with 16 registers and $II = 26$ cycles. In contrast, merely increasing the *II*, requires 31 cycles to schedule the loop with the same number of registers. Moreover, spill allows the loop APSI 50 to be scheduled with 32 and even 16 registers, while the technique of increasing the *II* failed to obtain a schedule even with 32 registers.

There are some results that need an additional comment. For instance, notice that the busses are never used 100%, and that in some cases spilling additional lifetimes decreases the bus usage. This is because, due to the "complex operations" added, the *II* increases faster than the *MII*, leaving some free bus slots. Another effect is that in some cases spilling additional lifetimes increases the register requirements. This is because the resulting graph is scheduled in a way that it requires slightly more registers. In addition in some cases the new graph might be scheduled with an smaller *II* (as in Figure 7b with 16 lifetimes spilled) leading to a noticeable increase in the register requirements (and also in memory traffic).

4.5. Increasing Efficiency

The main disadvantage of our approach to add spill code is that, for each variable spilled, the loop has to be rescheduled. This results in very low scheduling performance (i.e. the time to produce the schedules). In this section, we propose some simple techniques for boosting the performance of the scheduler to add spill code.

Spilling several lifetimes at once

The most obvious shortcut is to spill several lifetimes before rescheduling the loop. For this purpose, the decrease in register requirements of the new dependence graph must be estimated in order not to add an excessive amount of spill.

The register requirements are estimated by subtracting the lifetime selected from the schedule-dependent lower bound (*MaxLive*). Lifetimes will be selected until the lower bound is below the available number of registers.

Notice that this is an optimistic estimation since a lower bound is used instead of the actual register requirements. Furthermore, the decrease in register requirements caused by spilling a variable is not proportional to its lifetime because new —shorter— lifetimes are added to communicate the results to/from the added loads and stores.

Being so optimistic ensures that spill code is not added in excess. Loops with high register requirements will need several iterations to achieve the desired register requirements. Nevertheless, the number of times loops will be re-scheduled is extremely low compared with re-scheduling for each variable spilled out. For instance, if 32 registers are available, loops 47 and 50 from APSI are re-scheduled only once.

Pruning the search

Another area for performance improvement appears from the observation of the behavior of the *MII* and *II* in Figures 7a and 7b. Notice that when several variables have been spilled the *II* tends to be higher than the *MII*. This effect is caused by the *complex* operations that appear when spill code is added. So, in practice, for each time the loop is rescheduled, several schedules are tried with *II*'s ranging from *MII* to the final *II*.

It can also be observed that most of the time the *MII* is smaller than the *II* for which a valid schedule has been found in the previous re-scheduling iteration. It is also interesting to notice that the *II* almost never decreases from one iteration to the following.

Most time spent on unsuccessful scheduling attempts can be saved if, instead of exploring all the initiation intervals from *MII*, we explore from the maximum of the current *MII* and the previous *II*.

5. Experimental Evaluation

The effectiveness of the mechanisms to decrease the register requirements —in order to obtain valid schedules with a fixed number of registers— has been evaluated using HRMS [22] as the base scheduling technique. HRMS has been used because it is a fast, register-sensitive, software pipelining method. Nevertheless the techniques and heuristics proposed in this paper can be applied to any software pipelining method.

The techniques have been evaluated for a benchmark suite composed of a large number of innermost DO loops from the Perfect Club [5]. We have selected all loops composed of a single basic block. Loops with conditionals in their body have been previously converted to single basic block loops using IF-conversion [2]. We have not included loops with subroutine calls or with conditional exits. The dependence graphs have been obtained using the experimental ICTINEO compiler [3]. A total of 1258 loops, which account for 78% of the total execution time[4] of the Perfect Club, have been used.

We have used three functional unit configurations in order to provide a wider evaluation of the distinct techniques to reduce register pressure.

- Configuration P1L4 has 1 load/store unit, 1 Div/Sqrt unit, 1 adder and 1 multiplier. The adder and the multiplier have a latency of 4 cycles.
- Configuration P2L4 has 2 functional units of each kind with exactly the same latencies as P1L4.
- Finally, configuration P2L6 is like P2L4, but the adders and the multipliers have a latency of 6 cycles.

All configurations have in common a unit latency for store instructions, a latency of 2 for loads, a latency of 17 for divisions and a latency of 30 for square roots. For all configurations, all units are fully pipelined except the Div/Sqrt units which are not pipelined at all.

Functional unit configurations have been tested with 32 and 64 register files. This results in a set of experiments that ranges from moderate architectures (in terms of exploitable functional unit parallelism) with a large register supply to very aggressive architectures with a relatively limited register supply.

In Section 3 we have seen that increasing the *II* until a valid schedule (requiring no more than the available number of registers) is found, might not converge for some loops. Table 1 shows for all three configurations,

[4]Executed on an HP 9000/735 workstation.

Config.	Number of loops		% of dynamic cycles	
	64 regs.	32 regs.	64 regs.	32 regs.
P1L4	4	29	22%	35%
P2L4	4	30	20%	31%
P2L6	4	30	19%	29%

Table 1. Loops that never converge to a given number of registers, and % of cycles they represent.

how many loops never converge to 32 and to 64 registers. We have observed that the loops are the same (except in one case), independently of the configuration. In [21], using a register insensitive scheduler, the same loops (in fact one more loop) never converged to the desired number of registers for configuration P2L4. This behavior shows that, as suggested in Section 3, the main factor that determines the convergence of a loop is its topology.

Notice that only a few loops never converge to a solution. Unfortunately, the loops that cannot be scheduled with 64 and 32 registers represent about 20% and 30%, respectively, of the cycles of all 1258 loops when executed for the corresponding configurations assuming an infinite number of registers. From these results we conclude that increasing the *II* cannot be considered as a general purpose technique for register-constrained software pipelining.

The alternative technique we investigated is to add spill code. In Section 4 we have introduced two heuristics for selecting the lifetimes to spill. Figure 8a shows the overall performance of all the Perfect loops when using the two proposed spilling heuristics: $Max(LT)$ and $Max(LT/Traf)$. The heuristic that takes into account the cost of adding the spill code leads to schedules that have (for all configurations) better performance.

Notice that, when 64 registers are available there is almost no performance degradation due to the addition of spill code (even for the most aggressive configuration). Nevertheless, for all the 64-register configurations, the $Max(LT/Traf)$ heuristic produces schedules that (as expected) generate noticeable less memory traffic. Figure 8b shows the total memory traffic required by the loops when executed.

In Section 4 we also proposed two techniques for speeding-up the scheduling process (i.e. to reduce the compilation time). Figure 8c shows the time required to construct all the schedules. The scheduling time depends mainly on the quantity of spill to add (which is bigger for small register files). This is because for each variable to spill the loop is rescheduled. For the configurations with 64 registers, the scheduling time is about 15 minutes. Unfortunately, when only 32 registers are available (as in many of the existing microprocessors)

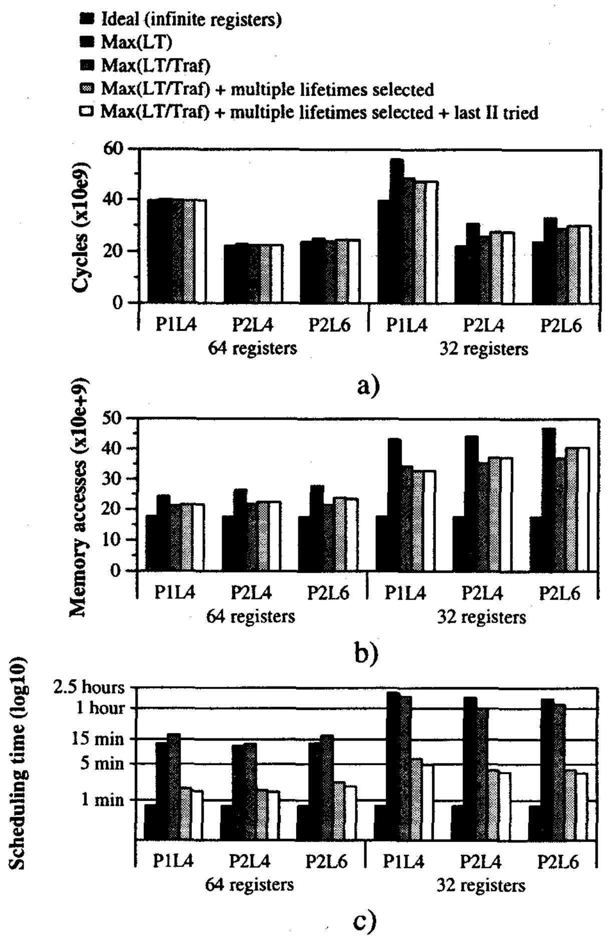

Figure 8. Evaluation of the spilling heuristics for several machine configurations: a) Cycles required to execute all the loops (in units of 10^9 cycles) b) Number of dynamic memory references (in units of 10^9) c) Time to schedule all the loops ($\log_{10}$ scale)

the scheduling time grows to more than 1 hour (1:40 for the worst case evaluated).

A technique for reducing the scheduling time is to spill several variables before re-scheduling the loop. As can be seen in Figure 8 the "$Max(LT/Traf)$+ multiple lifetimes selected" bar shows a small performance degradation (both in terms of execution time and memory traffic) in most of the cases. Nevertheless, for configuration P1L4 the heuristic produces a performance improvement. The small difference in performance is because a different set of variables is spilled if they are selected all together, or in separate steps (the loop has been rescheduled and the length of lifetimes varies). Since the selection is based on simple heuristics, selecting a slightly different set of variables can produce

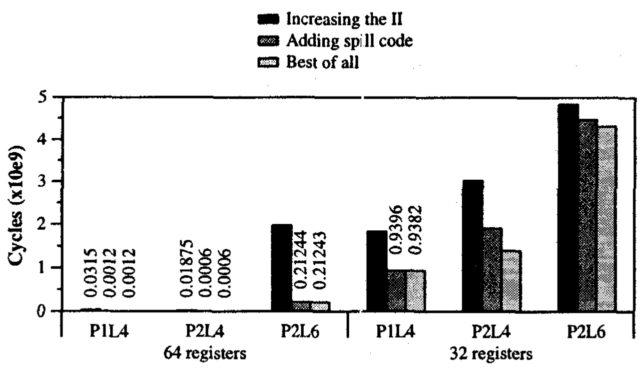

Figure 9. Increasing the II versus adding spill code: total execution time (in units of 10^9 cycles) of the loops that (1) require a reduction of the register requirements to fit in the available number of registers, and (2) converge to a solution by increasing the II.

either slightly better results or slightly worse results. Notice that this improvement reduces the scheduling time of the loops (for the configurations with 32 registers) from more than 1 hour to about 5 minutes.

In Figure 7 it can be observed that the II of loops with a high amount of spill code can be several cycles bigger than the MII because of the scheduling complexity introduced by the "complex" operations generated. As a consequence, the scheduler has to explore several schedules (from MII to the final II) before producing a valid schedule. If the loop must be rescheduled (with additional spills) and the new MII is lower than the previous II some time can be saved by skipping the exploration of the schedules from the current MII to the previous II. It can be noted that this optimization (bar "$Max(LT/Traf)$+ multiple lifetimes selected + last II tried" in Figure 8) reduces the scheduling time without any noticeable performance degradation. We consider that this combination of heuristics has the best behavior in terms of both execution time of the loops and compilation time.

Since increasing the II does not converge for some loops it is difficult to compare the spilling heuristics versus increasing the II. Therefore we performed a limited comparison using only the subset of loops that require more registers than available and that converges to a solution when the II is increased. Figure 9 compares the execution time of this subset of loops for the best set of heuristics ("$Max(LT/Traf)$+ multiple lifetimes selected + last II tried" bar in Figure 8) versus increasing the II. It must be said that the subset of loops compared varies depending on the machine configuration. Notice that, in all configurations, spilling produces schedules that, on average, have better performance than increasing the *II*. Even for some configurations (P2L6 with 64 registers) the advantage of spilling over increasing the *II* is astonishing.

Despite of this fact, we have observed that for a few loops spilling performs worse than increasing the *II*. Figure 9 also shows the performance of a combination of both techniques. The bar labeled "best of all" shows the total execution time of the loops when each loop is scheduled using the best technique for it. Notice that in some cases it produces a small reduction on the execution time of the loops. We have not implemented this improvement, but it would require a minor computational cost, if implemented as follows:

- If the loop requires additional registers, schedule it by adding spill code until a valid schedule is found.
- Once a valid schedule is found, schedule the loop with the same *II* but without the added spill code. If the schedule is valid for the available number of registers, then a better or equal schedule can be found by increasing the *II*
- Instead of exploring all the possible *II* configurations from *MII* until a valid schedule is found we can perform a binary search of the schedules between *MII* (lower bound) and the *II* obtained by adding spill code (upper bound).

With this implementation we only require to perform an additional schedule for the loops that require spill code. In addition this schedule will be performed fast, since it will be performed with a bigger *II* than required, having more available slots for the operations. Some loops will require additional scheduling steps, but we have observed that in most of the configurations this situation only arises for less than 10 loops (out of 1258). The worst case appears for configuration P2L6 with 32 registers, where 30 loops have better performance increasing the *II* than adding spill code.

6. Conclusions

In this paper, we have investigated several options to perform software pipelining with register constraints. Two options have been investigated in order to decrease register requirements when required: increasing the initiation interval (*II*), and adding spill code.

We have demonstrated that merely increasing the *II* does not converge to a solution for all loops. Even though it happens for a few loops, we have observed that they represent a significant amount of the execution time. For instance, if only 32 registers are available, the loops that never converge represent about 30% of the execution time.

When adding spill code, we have introduced heuristics to guarantee convergence independently of the scheduling technique: scheduling the spill loads/stores grouped with the predecessor/successor as a single "complex" operation and marking all variables whose producers/consumers are a spill load/store. We have also studied two heuristics for selecting the appropriate lifetimes: Selecting the largest lifetime and selecting the largest lifetime divided by the number of additional memory operations required. The last option proved to be more effective both, in terms of memory traffic and execution time (in cycles) of the resulting schedules.

We proposed two complementary heuristics to speed up the spill process. These heuristics caused a small performance degradation, but dramatically reduced the time required to produce a valid schedule with register constraints (from more than 1 hour to less than 5 minutes for the 1258 loops we used).

Finally, for the loops that achieve a valid schedule by increasing the *II*, we have compared the performance of these schedules with the schedules obtained by adding spill code. Adding spill code has proven, in general, to be more effective. Nevertheless, for a few loops, a better schedule was obtained by increasing the *II*. From this results we proposed a combination of both techniques that can be implemented without a noticeable increase in the compilation time. We have shown that this combination can produce better schedules than any of the techniques alone.

As future work, it remains to be studied other heuristics to select the variables to be spilled. For instance, if no hardware support (rotating register files) is provided for lifetimes larger than the *II*, cyclic interval graphs [16] can be used to perform the register allocation and to select the lifetimes to be spilled. Also in [14], even though they didn't dealt with the addition of spill code, there where heuristics proposed to select the variables to spill. Another possibility is to spill uses instead of variables, in any case, we do not expect a significant improvement, since most of the variables are used only once [15].

References

[1] V. Allan, R. Jones, R. Lee, and S. Allan. Software pipelining. *ACM Computing Surveys*, 27(3):367–432, September 1995.

[2] J. Allen, K. Kennedy, and J. Warren. Conversion of control dependence to data dependence. In *Proc. 10th annual Symposium on Principles of Programming Languages*, January 1983.

[3] E. Ayguadé, C. Barrado, J. Labarta, D. López, S. Moreno, D. Padua, and M. Valero. A uniform representation for high-level and instruction-level transformations. Technical Report UPC-CEPBA 95-01, Universitat Politècnica de Catalunya, January 1995.

[4] D. Bernstein, D. Goldin, M. Golumbic, H. Krawczyk, Y. Mansour, I. Nahshon, and R. Pinter. Spill code minimization techniques for optimizing compilers. In *Proc. of the ACM SIGPLAN'89 Conf. on Programming Languages Design and Implementation*, pages 258–263, July 1989.

[5] M. Berry, D. Chen, P. Koss, and D. Kuck. The Perfect Club benchmarks: Effective performance evaluation of supercomputers. Technical Report 827, Center for Supercomputing Research and Development, November 1988.

[6] P. Briggs, K. Cooper, K. Kennedy, and L. Torczon. Coloring heuristics for register allocation. In *Proc. of the ACM SIGPLAN'89 Conf. on Programming Language Design and Implementation*, pages 275–284, June 1989.

[7] P. Briggs, K. Cooper, and L. Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems*, 16(3):428–455, May 1994.

[8] D. Callahan and B. Koblenz. Register allocation via hierarchical graph coloring. In *Proc. of the ACM SIGPLAN'91 Conf. on Programming Language Design and Implementation*, pages 192–203, June 1991.

[9] G. Chaitin. Register allocation and spilling via graph coloring. In *Proc., ACM SIGPLAN Symp. on Compiler Construction*, pages 98–105, June 1982.

[10] A. Charlesworth. An approach to scientific array processing: The architectural design of the AP120B/FPS-164 family. *Computer*, 14(9):18–27, 1981.

[11] J. Dehnert, P. Hsu, and J. Bratt. Overlapped loop support in the Cydra 5. In *Proc. of the 3rd Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS-III)*, pages 26–38, April 1989.

[12] J. Dehnert and R. Towle. Compiling for the Cydra 5. *The Journal of Supercomputing*, 7(1/2):181–228, May 1993.

[13] A. Eichenberger and E. Davidson. Stage scheduling: A technique to reduce the register requirements of a modulo schedule. In *Proc. of the 28th Annual Int. Symp. on Microarchitecture (MICRO-28)*, pages 338–349, November 1995.

[14] C. Eisenbeis, S. Lelait, and B. Marmol. The meeting graph: a new model for loop cyclic register allocation. In *Proc., Fifth Workshop on Compilers for Parallel Computers (CPC95)*, pages 503–516, June 1995.

[15] M. Franklin and G. Sohi. Register traffic analysis for streamlining inter-operation communication in fine-grain parallel processors. In *Proc., 25th Annual Internat. Symp. on Microarchitecture (MICRO-25)*, pages 236–245, December 1992.

[16] L. Hendren, G. Gao, E. Altman, and C. Mukerji. Register allocation using cyclic interval graphs: A new approach to an old problem. ACAPS Tech. Memo 33, Advanced Computer Architecture and Program Structures Group, McGill University, 1992.

[17] R. Huff. Lifetime-sensitive modulo scheduling. In *6th Conference on Programming Language, Design and Implementation*, pages 258–267, 1993.

[18] S. Jain. Circular scheduling: A new technique to perform software pipelining. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 219–228, June 1991.

[19] M. Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, pages 318–328, June 1988.

[20] M. Lam. *A Systolic Array Optimizing Compiler.* Kluwer Academic Publishers, 1989.

[21] J. Llosa. *Reducing the Impact of Register Pressure on Software Pipelining.* PhD thesis, UPC. Universitat Politècnica de Catalunya, January 1996.

[22] J. LLosa, M. Valero, E. Ayguadé, and A. Gonzalez. Hipernode reduction modulo scheduling. In *Proc. of the 28th Annual Int. Symp. on Microarchitecture (MICRO-28)*, pages 350–360, November 1995.

[23] W. Mangione-Smith, S. Abraham, and E. Davidson. Register requirements of pipelined processors. In *Int. Conference on Supercomputing*, pages 260–246, July 1992.

[24] S. Ramakrishnan. Software pipelining in PA-RISC compilers. *Hewlett-Packard Journal*, pages 39–45, July 1992.

[25] B. Rau and C. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *Proceedings of the 14th Annual Microprogramming Workshop*, pages 183–197, October 1981.

[26] B. Rau, M. Lee, P. Tirumalai, and P. Schlansker. Register allocation for software pipelined loops. In *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation*, pages 283–299, June 1992.

[27] 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.

[28] F. Sánchez. *Loop Pipelining with Resource and Timing Constraints.* PhD thesis, UPC Universitat Politecnica de Catalunya, October 1995.

[29] P. Tirumalai, M. Lee, and M. Schlansker. Parallelisation of loops with exits on pipelined architectures. In *Proc., Supercomputing '90*, pages 200–212, November 1990.

[30] J. Wang, A. Krall, M. A. Ertl, and C. Eisenbeis. Software pipelining with register allocation and spilling. In *Proc. of the 27th Annual Int. Symp. on Microarchitecture*, pages 95–99, November 1994.

[31] N. Warter and N. Partamian. Modulo scheduling with multiple initiation intervals. In *Proc. of the 28th Internat. Symp. on Microarchitecture*, pages 111–118, November 1995.