Efficient Spilling Reduction for Software Pipelined Loops in Presence of Multiple Register Types in Embedded VLIW Processors

Sid-Ahmed-Ali TOUATI
University of Versailles Saint-Quentin-en-Yvelines
and
Frederic BRAULT
INRIA-Saclay
and
Karine DESCHINKEL
University of Versailles Saint-Quentin-en-Yvelines
and
Benoît DUPONT DE DINECHIN
STMicroelectronics

Integrating register allocation and software pipelining of loops is an active research area. We focus on techniques that pre-condition the dependence graph before software pipelining in order to ensure that no register spill instructions are inserted by the register allocator in the software pipelined loop. If spilling is not necessary for the input code, pre-conditioning techniques insert dependence arcs so that the maximum register pressure MAXLIVE achieved by any loop schedule is below the number of available registers, without hurting the initiation interval if possible. When a solution exists, a spill-free software pipeline is guaranteed to exist.

Existing pre-conditioning techniques consider one register type (register class) at a time [Karine Deschinkel and Sid-Ahmed-Ali Touati 2008]. In this paper, we extend pre-conditioning techniques so that multiple register types are considered simultaneously. First, we generalise the existing theory of register pressure minimisation for cyclic scheduling. Second, we implement our method inside the production compiler of the ST2xx VLIW family, and we demonstrate its efficiency on industry benchmarks (FFMPEG, MEDIABENCH, SPEC2000, SPEC2006). We demonstrate a high spill reduction rate without a significant initiation interval loss.

Categories and Subject Descriptors: D.3.4 [**Processor**]: Compilers, Code generation, Optimisation

1. INTRODUCTION

Media processing applications such as voice, audio, video, and image processing, spend most of their run-time in inner loops. Software pipelining is the key instruction scheduling technique used to improve performances, by converting loop-level parallelism into instruction-level parallelism (ILP) [Lam 1988; B. Ramakrishna Rau 1994]. However, on wide issue or deeply pipelined processors, the performance of software-pipelined loops is especially sensitive to the effects of register allocation [Lam 1988; Christine Eisenbeis and Sylvain Lelait and Bruno Marmol 1995; Joseph A. Fisher and Paolo Faraboschi and Clifford Young 2005], in particular the insertion of memory access instructions for spilling the live ranges.

Usually, loops are software pipelined assuming that no memory access miss the cache, and significant amount of research has been devoted to heuristics that produce near-optimal schedules under this assumption [B. Ramakrishna Rau and Michael S. Schlansker and P. P. Tirumalai 1992; John Ruttenberg and G. R. Gao and A. Stoutchinin and W. Lichtenstein 1996]. The code produced by software pipelining is then processed by the register allocation phase. However, a cache miss triggered by a spill instruction introduced by the register allocator has the potential to reduce the dynamic instruction level parallelism (ILP) below the level of the non software pipelined loop without the cache miss.

In addition to limiting the negative effects of cache misses on performances, reducing spill code has other advantages in embedded VLIW processors. For instance, energy consumption of the generated embedded VLIW code is reduced because memory requests need more power than regular functional units instructions. Also, reducing the amount of

spill code improves the accuracy of static program performance models: indeed, since memory operations have unknown static latencies (except if we use scratch-pad memories), the precision of WCET analysis and static compilation performance models is altered. When performance prediction models are inaccurate, static compiler transformation engines may be guided to bad optimisation decisions. Consequently, we believe that an important code quality criteria is to have a reduced amount of memory requests upon the condition of not altering ILP scheduling.

In a previous research achievement, we have proposed a theoretical framework called *SIRA* [Sid-Ahmed-Ali Touati and Christine Eisenbeis 2004] (*schedule-independent register allocation*) for the class of software pipelining techniques known as *modulo scheduling* [Lam 1988; B. Ramakrishna Rau 1994]. In particular, we use the SIRA framework to pre-condition the data dependence graph (DDG) before software pipelining in order to guarantee that the maximum register pressure MAXLIVE created by any instruction schedule does not exceed the number of available registers. In case of inner loops, this guarantees that a spill-free register allocation exists [Dominique de Werra and Christine Eisenbeis and Sylvain Lelait and Bruno Marmol 1999; Laurie J Hendren . and Guang R. Gao and Erik R Altman and Chandrika Mukerji 1992]. Given a number of available registers for each register type, SIRA add arcs to the DDG while trying to avoid increasing the critical circuit length if possible. This increase of the critical circuit length is the objective function to minimise.

In this paper, we augment the SIRA framework to address the problem of bounding register pressure in presence of multiple register types, without hurting the initiation interval (II) if possible. Optimising the register requirements of each register type separately cumulates the increases of the critical circuit length. As our experiments show, considering all the register types simultaneously when trying to minimise the increase of the critical cycle length gives good results. This is because loop statements are connected by complex data dependencies, and some statements may create multiple results with distinct register types.

This article is organised as follows. Section 2 presents relevant related work on periodic register allocation for innermost loops scheduled with software pipelining. Section 3 defines our loop model. Section 4 recalls the SIRA framework and the reuse graphs. It then proposes an efficient heuristic for controlling register pressure with multiple register types. Section 5 presents experimental results on well known benchmarks collections (MEDIABENCH, FFMPEG, SPEC2000, SPEC2006), showing that our method is effective in practice. Finally, we summarise our results and discuss some perspectives.

2. RELATED WORK IN PERIODIC REGISTER ALLOCATION

Classic register allocation involves three topics: which live ranges to evict from registers (register spilling); which register-register copy instructions to eliminate (register coalescing); and what architectural register to use for any live range (register assignment). The dominant framework for classic register allocation is the graph colouring approach pioneered by Chaintin et al. [Gregory Chaitin 2004] and refined by Briggs et al. [Preston Briggs and Keith D. Cooper and Linda Torczon 1994]. This framework relies on the vertex colouring of an *interference graph*, where vertices correspond to live ranges and edges to interferences. Two live ranges interfere if one is live at the definition point of the other and they carry different values.

In the area of software pipelining, live ranges may span multiple iteration, so the classic register allocation techniques are not directly applicable because of the self-interference of such live ranges. One solution is to unroll the software pipelined loop until no live range self-interferes, then apply classic register allocation. A better solution is to rely on techniques that understand the self-interferences created by loop iterations, also known as *periodic register allocation techniques*.

Because the restrictions on the inner loops that are candidate to software pipelining, the periodic register allocation techniques mostly focus on the issues related to register spilling and register coalescing. In particular, the register coalescing problem of a software pipeline can be solved by using modulo expansion and kernel unrolling [Dominique de Werra and Christine Eisenbeis and Sylvain Lelait and Bruno Marmol 1999; Laurie J Hendren . and Guang R. Gao and Erik R Altman and Chandrika Mukerji 1992; Lam 1988; B. Ramakrishna Rau and M. Lee and P. P. Tirumalaiand and Michael S. Schlansker 1992], or by exploiting hardware support known as rotating register files [B. Ramakrishna Rau and M. Lee and P. P. Tirumalaiand and Michael S. Schlansker 1992]. Without these techniques, register-register copy instructions may remain in the software pipelined loop [Alexandru Nicolau and Roni Potasman and Haigeng Wang 1992]. For the register spilling problems, one can either try to minimise the impact of spill code in the software pipeline [Santosh G. Nagarakatte and R. Govindarajan 2007], or pre-condition the scheduling problem so that spilling is avoided [Sid-Ahmed-Ali Touati and Christine Eisenbeis 2004].

The SIRA framework [Sid-Ahmed-Ali Touati and Christine Eisenbeis 2004] generalises previous research on pe-

riodic register allocation [Dominique de Werra and Christine Eisenbeis and Sylvain Lelait and Bruno Marmol 1999; Laurie J Hendren . and Guang R. Gao and Erik R Altman and Chandrika Mukerji 1992] by considering both scheduled and unscheduled loops. As a result, it can be used both for a pre-pass periodic register allocation to prevent live range spilling, or as a post-pass periodic register allocation to prevent live range splitting. SIRA also allows to handle rotating register files. It provides a framework that is general enough to model any cyclic register allocation heuristic: indeed the proposed reuse graphs (See Section 4) model any cyclic register allocation solution. SIRA was the first theoretical model that considered delays in accessing registers (important characteristics for VLIW and EPIC processors), with multiple register types.

The SIRA motivations for handling register constraints by pre-conditioning software pipelining are as follows:

(1) *Separating Register Pressure Control from Instruction Scheduling* With the increase of loop code size of media processing applications, methods that formulate software pipelining under both register pressure and resource constraints as integer linear programming problems [Alexandre E Eichenberger and Edward S. Davidson 1997; Santosh G. Nagarakatte and R. Govindarajan 2007; John Ruttenberg and G. R. Gao and A. Stoutchinin and W. Lichtenstein 1996] are not applicable in practice. Indeed, such exact methods are limited to loops with a few dozen instructions. In real media processing applications, it is not uncommon to schedule loops with hundreds of instructions. So, in order to reduce the difficulty of scheduling large loops, we satisfy the register constraints before the scheduled resource constraints (issue width, execution units).

(2) *Handling Registers Constraints before Scheduled Resource Constraints* This is because register constraints are more complex: given a bounded number of available registers, increasing the loop initiation interval (II) to reduce the register pressure does not necessarily provide a solution, even with optimal scheduling. Sometimes, spilling is mandatory to reduce register pressure. Spilling modifies the DDG, bringing an iterative problem of spilling followed by scheduling. By contrast, resource constraints are always solvable by increasing the II. For any DDG, there always exists at least one schedule under resource constraints, whatever these resource constraints are.

(3) *Avoiding Spilling instead of Scheduling Spill Code* This is because spilling introduces memory instructions whose exact latencies are unknown. Consequently, when the code is executed, any cache miss may have dramatic effects on performance, especially for VLIW processors. In other terms, even if we succeed to optimally schedule spill instructions as done in [Santosh G. Nagarakatte and R. Govindarajan 2007], actual performance does not necessarily follow the static schedule, because spill instructions may not hit the cache as assumed by the compiler.

3. LOOP MODEL

In a target architecture with multiple register types (for instance, $T = \{int, float, branch\}$), we consider an innermost loop (with possible recurrences). It is represented by a data dependence graph (DDG), such that :

(1) $V_{R,t}$ is the set of values to be stored in registers of type $t \in T$. Our theoretical model considers that each statement $u \in V$ may write to multiple registers, however there is at most one write per register type $t \in T$. We denote by u^t the value of type t defined by the statement u.

(2) $E_{R,t}$ is the set of flow dependence arcs through a register of type $t \in T$. Any arc e has the form $e = (u^t, v^t)$, where $\delta(e)$ is the latency of the arc e in terms of processor clock cycles and $\lambda(e)$ is the distance of the arc e in terms of number of iterations. This set also defines the set of consumers (readers) of each variable $u^t \in V_{R,t}$ as the sink of all flow dependence arcs starting from u:

$$Cons(u^t) = \{v \in V | (u, v) \in E_{R,t}\}$$

A software pipeline (SWP) is defined by a scheduling function σ that assigns to each statement $u \in V$ a scheduling date (in terms of processor clock cycles) that satisfies at least the data dependence constraints or other constraints (resources, registers, etc.). SWP is defined by an initiation interval (II), and the scheduling date σ_u for the operations of the first iteration. Operation u of iteration i (noted $u(i)$) is scheduled at time $\sigma_u + (i-1) \times II, \forall e = (u, v) \in E$. Such instruction schedule must satisfy the usual cyclic data dependencies: $\sigma_u + \delta(e) \leq \sigma_v + \lambda(e) \times II$.

By aggregating all these constraints on all the DDG circuits, we find that $II \geq MII_{dep} = \max_{\text{any circuit } C} \frac{\delta(C)}{\lambda(C)}$, where $\delta(C) = \sum_{e \in C} \delta(e)$ and $\lambda(C) = \sum_{e \in C} \lambda(e)$. Any circuit C of the DDG that maximises the fraction $\frac{\delta(C)}{\lambda(C)}$ is called a *critical circuit* since it imposes a lower limit on the value of the II. Since an efficient SWP schedule must minimise the value of II, we need to take care of not increasing the cost of the critical circuits.

In our SIRA model, we take into account specific delays for accessing registers. On some VLIW architectures (such as Philips Trimedia and the VelociTI / TMS320C6xxx), delays for accessing registers are architecturally defined. In our model, we define two integral delay functions δ_r and $\delta_{w,t}$. $\forall u \in V$, the operation u reads its source registers at date $\sigma_u + \delta_r(u)$. $\forall u \in V_{R,t}$, the operation u^t writes its result at date $\sigma_u + \delta_{w,t}(u)$.

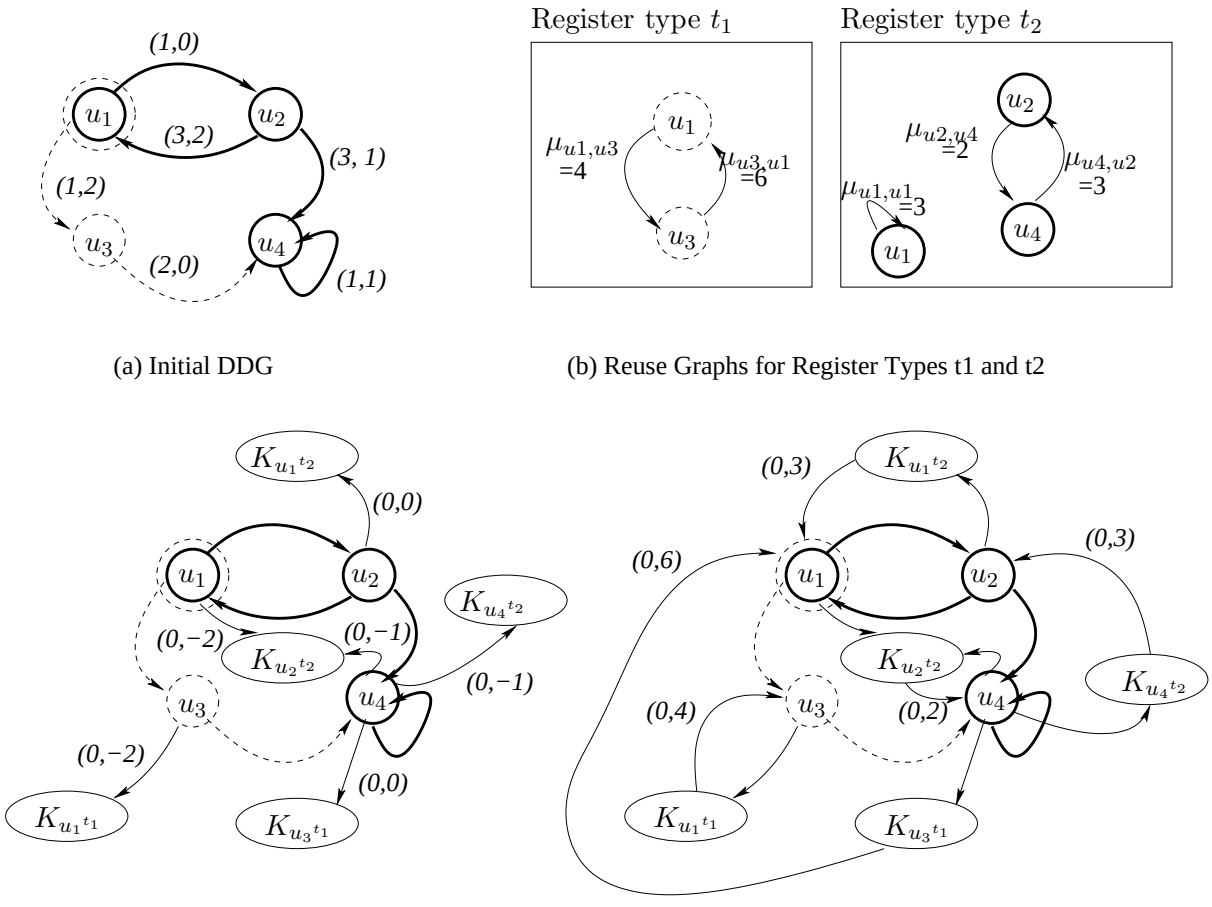
4. BOUNDING REGISTER PRESSURE IN PRESENCE OF MULTIPLE REGISTER TYPES

In the following section, we recall the notion of reuse graphs used inside SIRA. Then we provide an efficient formally defined heuristic for SIRA using a combination of linear programming and linear assignment algorithm.

4.1 SIRA and Reuse Graphs

A simple way to explain and recall the concept of SIRA is to provide an example. All the theory has already been presented in [Sid-Ahmed-Ali Touati and Christine Eisenbeis 2004]. Figure 1(a) provides an initial DDG with two register types t_1 and t_2. Statements producing results of type t_1 are in dashed circles, and those of type t_2 are in bold circles. Statement u_1 writes two results of distinct types. Flow dependence through registers of type t_1 are in dashed arcs, and those of type t_2 are in bold arcs.

As an example, $Cons(u_2^{t_2}) = \{u_1, u_4\}$ and $Cons(u_3^{t_1}) = \{u_4\}$. Each arc e in the DDG is labeled with the pair of values $(\delta(e), \lambda(e))$. In this simple example, we assume that the delay of accessing registers is zero ($\delta_{w,t} = \delta_r = 0$). Now, the question is how to compute a periodic register allocation for the loop in Figure 1(a) without increasing the critical circuit if possible.



(a) Initial DDG

(b) Reuse Graphs for Register Types t1 and t2

(c) DDG with Killing Nodes

(d) Preconditioned DDG after Applying SIRA

Fig. 1. Example for SIRA and Reuse Graphs

As formally studied in [Sid-Ahmed-Ali Touati and Christine Eisenbeis 2004], periodic register allocation is modeled thanks to *reuse graphs*. We associate a reuse graph G_t^r to each register type t, see Figure 1(b). The reuse graph has to be computed by the SIRA framework, Figure 1(b) is one of the examples that SIRA may produce. Note that the reuse graph is not unique, other valid reuse graphs may exist.

A reuse graph G_t^r contains $V_{R,t}$, *i.e.*, only the nodes writing inside registers of type t. These nodes are connected by *reuse arcs*. For instance, in $G_{t_2}^r$, the set of reuse arcs is $\{(u_2, u_4), (u_4, u_2), (u_1, u_1)\}$. Each reuse arc (u^t, v^t) is labeled by an integral distance $\mu_{u,v}^t$. The existence of a reuse arc (u^t, v^t) of distance $\mu_{u,v}^t$ means that the two operations $u^t(i)$ and $v^t(i + \mu_{u,v}^t)$ *share the same destination register*. Hence, reuse graphs allows to completely define a periodic register allocation for a given loop, either before SWP (unscheduled loop) or after SWP (already scheduled loop, as done with meeting graphs [Christine Eisenbeis and Sylvain Lelait and Bruno Marmol 1995]).

In order to be valid, reuse graphs should satisfy two main constraints [Sid-Ahmed-Ali Touati and Christine Eisenbeis 2004]: 1) They should describe a bijection between the nodes; that is, they must be composed of elementary and disjoint circuits. 2) The *associated DDG* should be schedulable, *i.e.*, it has at least one valid SWP.

Now, let us describe what we mean by the DDG *associated with* a reuse graph. Once a reuse graph is fixed before SWP, say the reuse graph of type t_2 in Figure 1(b), the periodic register allocation creates new scheduling constraints between statements. These scheduling constraints result from the anti-dependencies created by register reuse. Since each reuse arc (u^t, v^t) in the reuse graph G_t^r describes a register sharing between $u^t(i)$ and $v^t(i + \mu_{u,v}^t)$, we must guarantee that $v^t(i + \mu_{u,v}^t)$ writes inside the same register after the execution of all the consumers of $u^t(i)$. That is, we should guarantee that $v^t(i + \mu_{u,v}^t)$ writes its result after the killing date of $u^t(i)$. If the loop is already scheduled, the killing date is known. However, if the loop is not already scheduled, then the killing date is not known and hence we should be able to guarantee the validity of periodic register allocation for all possible SWP.

Guaranteeing precedence relationship between lifetime intervals for any subsequent SWP is done by creating *the associated DDG* with the reuse graph. This DDG is an extension of the initial one in two steps:

(1) First, we introduce dummy nodes representing the killing dates of all values. This idea was already present in [Benoît Dupont-de-Dinechin 1997]. For each value $u^t \in V_{R,t}$, we introduce a node K_{u^t} which represents its killing date. The killing node K_{u^t} must always be scheduled after all u^t's consumers. Consequently, we add the set of arcs $\{(v, K_{u^t}) | v \in Cons(u^t)\}$. Figure 1(c) illustrates the DDG after adding all the killing nodes for all register types. For each added arc $e = (v, K_{u^t})$, we set its latency to $\delta(e) = \delta_r(v)$ and its distance to $-\lambda$, where λ is the distance of the flow dependence arc $(u, v) \in E_{R,t}$. As explained in [Sid-Ahmed-Ali Touati and Christine Eisenbeis 2004], this negative distance is a mathematical convention, it simplifies our mathematical formula and does not influence the fundamental results of reuse graphs.

(2) Second, we introduce new anti-dependence arcs implied by periodic register allocation. For each reuse arc (u^t, v^t) in G_t^r, we add an arc $e' = (K_{u^t}, v^t)$ representing an *anti-dependence* in the associated DDG. We say that the anti-dependence $e' = (K_{u^t}, v^t)$ in the DDG G is associated to the reuse arc (u^t, v^t) in G_t^r. The added anti-dependence arc has a latency equal to $\delta(e') = -\delta_{w,t}(v)$ and has a distance equal to the reuse distance $\lambda(e') = \mu_{u,v}^t$. Figure 1(d) illustrates the DDG associated to the two reuse graphs of Figure 1(b). Periodic register allocation with multiple register types is done conjointly on the same DDG even if each register type has its own reuse graph. The reader may notice that the critical circuits of the DDG in Figure 1(a) and (c) are the same and equal to $MII_{dep} = \frac{4}{2} = 2$ (a critical circuit is (u_1, u_2)). The set of added anti-dependence arcs of type t is noted E_t^r (do not confuse with $E_{R,t}$)). In Figure 1(d), $E_{t_1}^r = \{(K_{u_1}^{t_1}, u_3), (K_{u_3}^{t_1}, u1)\}$ and $E_{t_2}^r = \{(K_{u_1}^{t_2}, u_1), (K_{u_2}^{t_2}, u_4), (K_{u_4}^{t_2}, u_2)\}$.

As can be seen, computing a reuse graph of a register type t implies the creation of new arcs with μ distances. We proved in [Sid-Ahmed-Ali Touati and Christine Eisenbeis 2004] that if a reuse graph G_t^r is valid, then it describes a periodic register allocation with exactly $\sum \mu_{u,v}^t$ registers of type t.

Now the SIRA problem is to compute a valid reuse graph with a minimised $\sum \mu_{u,v}^t$, without increasing the critical circuit if possible. Or, instead of minimising the register requirement, SIRA may simply look for a solution such that $\sum \mu_{u,v}^t \leq R_t$, where R_t is the number of available registers of type t. We may propose many exact method models (the problem has been proved NP-complete in [Sid-Ahmed-Ali Touati and Christine Eisenbeis 2004]) or heuristics based on the SIRA framework. The following section presents SIRALINA, an efficient two steps heuristic.

4.2 SIRALINA: A Two-Steps Polynomial Heuristic for Multiple Register Types

Our resolution strategy is based on the analysis of the exact integer linear model of SIRA published in [Sid-Ahmed-Ali Touati and Christine Eisenbeis 2004]. As the problem involves scheduling constraints and assignment constraints, and

the reuse distances are the link between these two sets of constraints, we attempt to decompose the problem into two sub-problems:

—A scheduling problem: to find a scheduling for which the potential reuse distances are as small as possible. This step essentially minimises the total sum of all lifetime intervals for all register types $t \in T$, *i.e.* the total sum of the times between the killing nodes schedules $\sigma_{k_{u^t}}$ and the nodes schedules σ_{u^t}. This first step is independent of the reuse graph. The next step creates a correct reuse graph based on the costs computed in this first step.

—An assignment problem: to select which pairs of statements will share the same register. Based on the schedule information of the first step, this second step builds reuse arcs (with their corresponding anti-dependences) and a correct valid reuse graph.

For the case of a unique register type, a similar two steps heuristics has been presented in [Karine Deschinkel and Sid-Ahmed-Ali Touati 2008] and demonstrated effective on some toy benchmarks. Here, we provide a generalisation of that heuristic in the case of multiple register types, with full industry-quality implementation and experimentation.

4.2.1 *Variables for the Linear Problem*

—An integer schedule variable $\sigma_u \in \mathbb{N}$ for each statement $u \in V$. We assume a finite upper bound L for such schedule variables (L sufficiently large, $L = \sum_{e \in E} \delta(e)$);

—$\forall t \in T, u^t \in V_{R,t}$ has a killing node K_{u^t}, thus a scheduling variable $\sigma_{K_{u^t}} \in \mathbb{N}$.

—A reuse distance $\mu^t_{u,v} \in \mathbb{N}, \forall (u,v) \in V^2_{R,t}, \forall t \in T$.

—A binary variables $\theta^t_{u,v}$ for each $(u,v) \in V^2_{R,t}, \forall t \in T$. It is set to 1 iff (K_{u^t}, v) is an anti-dependence arc ((u,v) is a reuse arc); That is, $\theta^t_{u,v} = 1$ iff the operations $u(i)$ and $v(i + \mu^t_{u,v})$ share the same destination register.

When we have multiple register types, we are faced to optimise multiple objectives. Ideally, given a number R_t of available registers of type t, we seek for a solution such as $\forall t \in T, \sum_{(u,v) \in V^2_{R,t}} \mu^t_{u,v} \leq R_t$. Let note $z^t = \sum_{(u,v) \in V^2_{R,t}} \mu^t_{u,v}$. We combine all these objective functions into a single linear objective function by introducing general weights between register types:

$$\begin{aligned} \text{Minimise} \quad & \sum_{t \in T} \alpha_t z^t \\ = & \sum_{t \in T} \alpha_t \sum_{(u,v) \in V^2_{R,t}} \mu^t_{u,v} \end{aligned}$$

where α_t defines a weight associated to the register type t. For instance, the branch register type on a VLIW processor such as ST231 may be more critical than the general purpose register type: this is because there are few branch registers, and they are single bits so not easily spillable. Consequently, we may be asked to give higher weights for a register type against another if needed. In our context, a unit weight ($\alpha_t = 1, \forall t$) is sufficient to have satisfactory results as will be shown later in the experiments. However, other contexts may require distinct weights that the user is free to fix depending on the priority between the registers types.

4.2.2 *Step 1: The Scheduling Problem.*

This scheduling problem is built for a fixed II which indeed describes the desired critical circuit of the DDG when SIRA is performed before SWP. We first solve a periodic scheduling problem for the DDG described in Figure 1(c), independently of a chosen reuse graph. That is, we handle the DDG with killing nodes only without any anti-dependences. The goal of this first step of SIRALINA is to compute the potential values of all $\mu^t_{u,v}$ variables for all pairs $(u,v) \in V^2_{R,t}$, independently of the reuse graph that will be constructed in the second step.

If $e = (K_{u^t}, v)$ is an anti-dependence arc associated to a reuse arc (u^t, v^t) (this will be decided in the second step of SIRALINA, *i.e.* to decide if $\theta^t_{u,v} = 1$), then its reuse distance must satisfy the following inequality [Sid-Ahmed-Ali Touati and Christine Eisenbeis 2004]:

$$\forall (K_{u^t}, v) \in E^r_t : \mu^t_{u,v} \geq \frac{1}{II}(\sigma_{K_{u^t}} - \delta_{w,t}(v) - \sigma_v) \tag{1}$$

This inequality gives a lower bound for each reuse distance of anti-dependence arc; We recall that E^r_t denotes the set of anti-dependence arcs of type t.

If (K_{u^t}, v) is not an anti-dependence arc then $\theta^t_{u,v} = 0$. In this case, according to [Sid-Ahmed-Ali Touati and Christine Eisenbeis 2004], $\mu^t_{u,v}$ is equal to zero:

$$\forall (K_{u^t}, v) \notin E^r_t : \mu^t_{u,v} = 0 \tag{2}$$

Now we can write:

$$z^t = \sum_{(u,v)\in V^2_{R,t}} \mu^t_{u,v} = \sum_{(K_{u^t},v)\in E^r_t} \mu^t_{u,v} + \sum_{(K_{u^t},v)\notin E^r_t} \mu^t_{u,v}$$

From Equation. 2, we know that $\sum_{(K_{u^t},v)\notin E^r_t} \mu^t_{u,v} = 0$. Consequently, by considering Inequality 1:

$$z^t \geq \frac{1}{II} \sum_{(K_{u^t},v)\in E^r_t} \left(\sigma_{K_{u^t}} - \delta_{w,t}(v) - \sigma_v\right) \tag{3}$$

As the reuse relation is a bijection from $V_{R,t}$ to $V_{R,t}$, then E^r_t describes a bijection between the set of killing nodes of type t and $V_{R,t}$. This bijection implies that, in the right sum of Inequality 3, we can have one and only one $\sigma_{K_{u^t}}$ term. Also, we can have one and only one σ_v term. Inequality 3 can then be separated into two parts as follows:

$$\begin{aligned} \sum_{(K_{u^t},v)\in E^r_t} \left(\sigma_{K_{u^t}} - \delta_{w,t}(v) - \sigma_v\right) &= \sum_{u\in V_{R,t}} \sigma_{K_{u^t}} - \sum_{v\in V_{R,t}} (\delta_{w,t}(v) + \sigma_v) \\ &= \sum_{u\in V_{R,t}} \sigma_{K_{u^t}} - \sum_{v\in V_{R,t}} \sigma_v - \sum_{v\in V_{R,t}} \delta_{w,t}(v) \end{aligned} \tag{4}$$

We deduce from Equality 4 a lower bound for the number of required registers of type t:

$$z^t \geq \frac{1}{II} \left(\sum_{u\in V_{R,t}} \sigma_{K_{u^t}} - \sum_{v\in V_{R,t}} \sigma_v - \sum_{v\in V_{R,t}} \delta_{w,t}(v) \right) \tag{5}$$

In this context, it is useful to find an appropriate schedule in which the right hand side of Inequation 5 is minimal for all register types $t \in T$. Since II and $\sum_{v\in V_{R,t}} \delta_{w,t}(v)$ are two constants, we can ignore them in the following linear optimisation problem. We consider *the scheduling problem (P)*:

$$\begin{cases} \min \sum_{t\in T} \alpha_t \left(\sum_{u\in V_{R,t}} \sigma_{K_{u^t}} - \sum_{v\in V_{R,t}} \sigma_v \right) \\ \text{subject to:} \\ \sigma_v - \sigma_u \geq \delta(e) - II \times \lambda(e), & \forall e = (u,v) \in E \\ \sigma_{K_{u^t}} - \sigma_v \geq \delta_r(v) + II \times \lambda(e), & \forall t \in T, \forall u^t \in V_{R,t},\ \forall v \in Cons(u^t) \end{cases} \tag{6}$$

These constraints guarantee that the resulting reuse graph is valid, *i.e.*, its associated DDG is schedulable with SWP. As can be easily seen, the constraints matrix of the integer linear program of System 6 is an incidence matrix, so it is totally unimodular [A. Schrijver 1986]. Consequently, we can use a polynomial algorithm to solve this problem. We can for instance use a linear solver instead of a mixed integer linear one. Also, we can use a min-cost network-flow algorithm to solve this scheduling problem in $O(|V|^3 \log |V|)$ [Ravindra K. Ahuja Ravindra and Thomas L. Magnanti and James B. Orlin 1991].

The resolution of problem (P) (by simplex method or by network-flow algorithm) provides optimal values σ^*_u for each $u \in V$ and optimal values $\sigma^*_{K_{u^t}}$ for each killing node K_{u^t}.

The objective function of the scheduling problem described above tries to minimise the sum of the lifetime intervals of all register types considering them as weighted. In the experiments described later, we give the same weights $\alpha_t = 1$ to all register types, and this works well for our case.

4.2.3 *Step 2: The Linear Assignment Problem.* The goal of this second step is to decide about reuse arcs (compute the values of $\theta^t_{u,v}$ variables) such that the resulting reuse graph is valid. Once the scheduling variables have been fixed

in the same conjoint scheduling problem (P) for all register types, the minimal value of each potential reuse distance becomes equal to $\overline{\mu^t_{u,v}} = \lceil \frac{\sigma^*_{K_{u^t}} - \delta_{w,t}(v) - \sigma^*_v}{II} \rceil$ according to Inequation 1. Knowing the reuse distance values $\overline{\mu^t_{u,v}}$, the periodic register allocation becomes now a problem of deciding which instruction reuses which released register, *i.e.*, compute the value of $\theta^t_{u,v}$ variables. This problem can be modeled as a linear assignment problem for each register type t. The constraints is that the produced reuse graph (modeled by an assignment relationship) should be a bijection between loop statements. We consider *the linear assignment problem (A^t) for the register type t* as:

$$\begin{cases} \min \quad \sum_{(u,v)\in V^2_{R,t}} \overline{\mu^t_{u,v}}\theta^t_{u,v} \\ \text{Subject to} \\ \sum_{v\in V_{R,t}} \theta^t_{u,v} = 1, & \forall u \in V_{R,t} \\ \sum_{u\in V_{R,t}} \theta^t_{u,v} = 1, & \forall v \in V_{R,t} \\ \theta^t_{u,v} \in \{0,1\} \end{cases} \tag{7}$$

where $\overline{\mu^t_{u,v}}$ is a fixed value for each arc $e = (u,v) \in V^2_{R,t}$.

Each linear assignment problem A^t is optimally solved with the well known Hungarian algorithm in $O(n^3)$ complexity. The Hungarian algorithms computes for each register type t the optimal values ${\theta^t_{u,v}}^*$. If ${\theta^t_{u,v}}^* = 1$, then (K_{u^t}, v) is a anti-dependence arc and the reuse distance is equal to $\overline{\mu^t_{u,v}}$. Otherwise, (K_{u^t}, v) does not exist. Our two step heuristic has now computed all what we need for a valid periodic register allocation for all register types: the set of anti-dependence arcs of type t (represented by the set of ${\theta^t_{u,v}}^*$ variables equal to one), and the reuse distances (represented by the values $\overline{\mu^t_{u,v}}$).

Finally, provided a number R_t of available registers of type t, we should check that $\forall t \in T | \sum \mu^t_{u,v} \leq R_t$. If not, this means that SIRALINA did not find a solution for the desired value of the critical circuit II. We thus increase II: since it is proved in [Sid-Ahmed-Ali Touati 2007] that the minimal periodic register need is a non increasing function of II, we can then use a binary search for II (between $MinII$ and the upper limit L). If we reach the upper limit for II without finding a solution, this means that the register pressure is too high and spilling becomes necessary: we can do spilling either before SWP (this is an open problem), or after SWP (as currently done in our experiments). The SIRA framework does not insert any spill, it is let for a subsequent pass of the compiler (the register allocator for instance).

The next section shows that SIRALINA is efficient in practice. We clearly demonstrates that performing SIRALINA before SWP is a better approach for spill code reduction than a regular SWP followed by register allocation without statistically significant hurt of II.

5. EXPERIMENTAL STUDY

5.1 Experimental Setup

Our experimental setup is based on st200cc, a STMicroelectronics production compiler based on the Open64 technology (`www.open64.net`), whose code generator has been extensively rewritten in order to target the STMicroelectronics ST200 VLIW processor family. These VLIW processors implement a single cluster derivative of the Lx architecture [Geoffrey Farabosch and Joseph A Fisher and Giuseppe Desoli and Fred Homewood 2000], and are used in several successful consumer electronics products, including DVD recorders, set-top boxes, and printers. At the end of 2008, the number of shipped ST200 processors was over 33 million units.

The ST231 processor used for our experiments executes up to 4 operations per cycle with a maximum of one control operation (`goto`, `jump`, `call`, `return`), one memory operation (`load`, `store`, `prefetch`), and two multiply operations per cycle. All arithmetic instructions operate on integer values with operands belonging either to the General Register (GR) file (64 $\times$ 32-bit), or to the Branch Register (BR) file (8 $\times$ 1-bit). Floating point computation are emulated by software. In order to eliminate some conditional branches, the ST200 architecture also provides conditional selection. The processing time of any operation is a single clock cycle, while the latencies between operations range from 0 to 3 cycles.

The st200cc compiler augments the Open64 code generator with super-block instruction scheduling optimisations, including a software pipeliner based on a generalised variant of decomposed software pipelining [Benoît Dupont-de-Dinechin 1997; Jian Wang and Christine Eisenbeis and Martin Jourdan and Bogong Su 1994]. We inserted the SIRA optimiser that preconditions the dependence graph before software pipelining in order to bound MAXLIVE for any

subsequent schedule. The present register allocator inside st200cc is called after SWP. It is a heuristic based on Chow priority based method.

The st200cc compiler has the capability of compiling for variants of the ST200 VLIW architecture, including changes in the instruction latencies, the issue width, and the number of allocatable registers. When we configure the processor to have 64 GR and 8 BR registers, we find that the register pressure is not problematic in most of the applications (only few spill instructions are generated): when register pressure is low, any weak register optimisation method would work fine and it is not necessary to use more clever method as we experiment in this article. In order to highlight the efficiency of a register optimisation method as ours, we must experiment harder constraints by compiling for smaller processors with less registers. For this work, we configured the compiler to assume the embedded VLIW processors to have 32 general-purpose registers (GR) and 4 branch registers (BR). The compiler can use all the available registers for SWP loops, or can dedicate a subset of them for other scalar variables (scalar promotion, global variables, etc.). In our experiments we can either decide to dedicate all available registers to SWP or not. We experimented both of the two configurations:

(1) In a first configuration, we dedicate all the available registers to SWP (32 GR and 4 BR). The experimental results we obtained are similar to the second configuration. Since our conclusions are equivalent, the experimental results we report in this section are conducted with the following configuration since it makes a higher stress on register pressure and on SWP;

(2) In a second configuration, we allocate a reasonable subset of available registers to SWP. We configured the compiler to dedicate 22 GR and 3 BR registers to SWP, the remaining available registers can be used for other purposes. According to STMicroelectronics, these restrictions are representative of mainstream small embedded processors used for media processing.

5.2 Qualitative Benchmarks Presentation

We conducted an extensive set of experiments on both high performance and embedded benchmarks. We chose to optimise the set of the following collections of well known applications programmed in C and C++.

(1) FFMPEG is the reference application benchmark used by STMicroelectronics for their compilation research and development. It is a representative application for the usage of ST231 (video mpeg encoder/decoder). The application is a set of 119 C files, containing 112997 lines of C code.

(2) MEDIABENCH is a collection of ten applications for multimedia written in C (encryption, image and video processing, compression, speech recognition, etc.). In its public version, MEDIABENCH is not a portable to any platform because some parts are coded in assembly language of some selected workstation targets (excluding VLIW targets). Our used MEDIABENCH collection has first been ported to ST231 VLIW platform. The whole MEDIABENCH applications have 1467 C files, containing 788261 lines of C code.

(3) SPEC2000 is a collection of applications for high performance computing and desktop market (scientific computing, simulation, compiler, script interpreters, multimedia applications, desktop applications, etc.). It is a group of 12 big applications of representative integer programs and 4 big applications of floating point programs. The whole collection contains 469 C files, 151 C++ files (656867 lines of C and C++ code).

(4) SPEC CPU2006 is the last collection of applications for scientific computing, intensive computation and desktop market. Compared to SPEC2000, SPEC2006 has larger code size and data sets (2386 C file, 528 C++ files, 3365040 C/C++ lines).

Both FFMPEG and MEDIABENCH collections have successfully been compiled, linked and executed on the embedded ST231 platform. For SPEC2000 and SPEC CPU2006, they have been successful compiled and statically optimised but not executed because of one of the three following reasons:

(1) Our target embedded system does not support some required dynamic function libraries by SPEC (the dynamic execution system of an embedded system is not as rich as a desktop workstation).

(2) The large code size of SPEC benchmarks does not fit inside small embedded systems based on ST231.

(3) The amount of requested dynamic memory (heap) cannot be satisfied at execution time on our embedded platform.

Consequently, our experiments report static performance numbers for all benchmarks collections. The dynamic performance numbers (executions) are reported only for FFMPEG and MEDIABENCH applications. This is not a restriction

of the study because neither SPEC2000 nor SPEC2006 are representative of the embedded applications we target; we statically optimise SPEC2000 and SPEC2006 applications to simply check and demonstrate at compile time that our spill optimisation method works also well for these kind of large applications.

The next section provides some useful quantitative metrics to analyse the complexity of our benchmarks. These quantitative measures allow to analyse the practical efficiency of SIRALINA heuristic. Since our heuristic complexity is $O(|V|^3 \log |V|)$, a detailed quantitative analysis of the benchmarks gives more hint about the input problems sizes and their complexity.

5.3 Quantitative Benchmarks Presentation

Our spill reduction strategy is plugged inside the st200cc compiler. It is called just before SWP module. The total number of optimised loops with SIRALINA is equal to 9027 (all benchmarks collections). SIRALINA followed by SWP are called by the compiler for optimising innermost loops at the backend level. The st200cc compiler may apply multiple code transformations before SWP (instruction selection, super-block formation, function inlining, loop unrolling, scalar promotion, etc.). Consequently, the loops optimised at the backend level are not necessarily correlated with the loops of the source codes.

Our SIRALINA heuristic has an algorithmic complexity equal to $O(|V|^3 \log |V|$. In order to have a precise idea on problem sizes treated by SIRALINA, we report six metrics using histograms (the x-axis represent the values, the y-axis represent the number of loops of the given values):

(1) The numbers of nodes (loop statements) are depicted in Figure 2 for each benchmark collection. The whole median[1] is equal to 24 nodes; the maximal value is 847. FFMPEG has the highest median of nodes numbers (29).

(2) The number of nodes writing inside general registers (GR) are depicted in Figure 3. The whole median is equal to 15 nodes; the maximal value is 813 nodes. FFMPEG has the highest median (21 nodes).

(3) The numbers of nodes writing inside branch registers (BR) are depicted in Figure 4. The whole median is equal to 3 nodes; the maximal value is 35 nodes. Both FFMPEG and MEDIABENCH has a median of 1 node, meaning that half of their loops has a unique branch instruction (the regular loop branch). As can be remarked, our framework considers loops with multiple branch instructions inside their bodies.

(4) The numbers of arcs (data dependences) are depicted in Figure 5 for each benchmark collection. The whole median is equal to 73 arcs; the maximal value is 21980 arcs. The highest median is FFMPEG one (99 arcs).

(5) The MinII values are depicted in Figure 6. We recall that $MinII = \max(MII_{dep}, MII_{res})$. The whole median of MinII values is equal to 12 clock cycles; the maximal value is 640 clock cycles. The highest median is the one of FFMPEG (20 clock cycles).

(6) The numbers of strongly connected components are depicted in Figure 7. The whole median is equal to 9 strongly connected components, which means that, if needed, half of the loops can be splitted by loop fission into 9 smaller loops; The maximal value is equal to 295. FFMPEG has the smallest median (7 strongly connected components).

Theses quantitative measures show that the FFMPEG application brings *a priori* the most difficult and complex DDG instances for code optimisation. This analysis is confirmed by our experiments below.

The next section shows statistics on the compilation times consumed by SIRALINA.

5.4 SIRALINA Compilation Times

The st200cc compiler is used as a cross compiler on a regular workstation. First, we measured the run time in milliseconds of our SIRALINA method when executed on a 1.6 GHz Pentium R dual core workstation (4 Gbytes of memory, and 1 MBytes of cache per core). We measured the optimisation times of all SWP loops when all register types are optimised conjointly, as well as when we optimise each register type separately (BR followed by GR, and *vice-versa*). This section demonstrates that optimising each register type separately (in any order) requires more resolution time that optimising all register types conjointly.

Tables I -II -III illustrate full statistics on each benchmark collection. These tables present the siralina resolution times in mili seconds for three scenarios: 1) conjoint optimisation of all register types, 2) separate optimisation of GR followed by BR, 3) and BR followed by GR. We report the minimal compilation time per loop, the first quartile value

[1] We deliberately choose to report the median value instead of the mean value, because the histograms show a skewed (biased) distribution [Raj Jain. 1991].

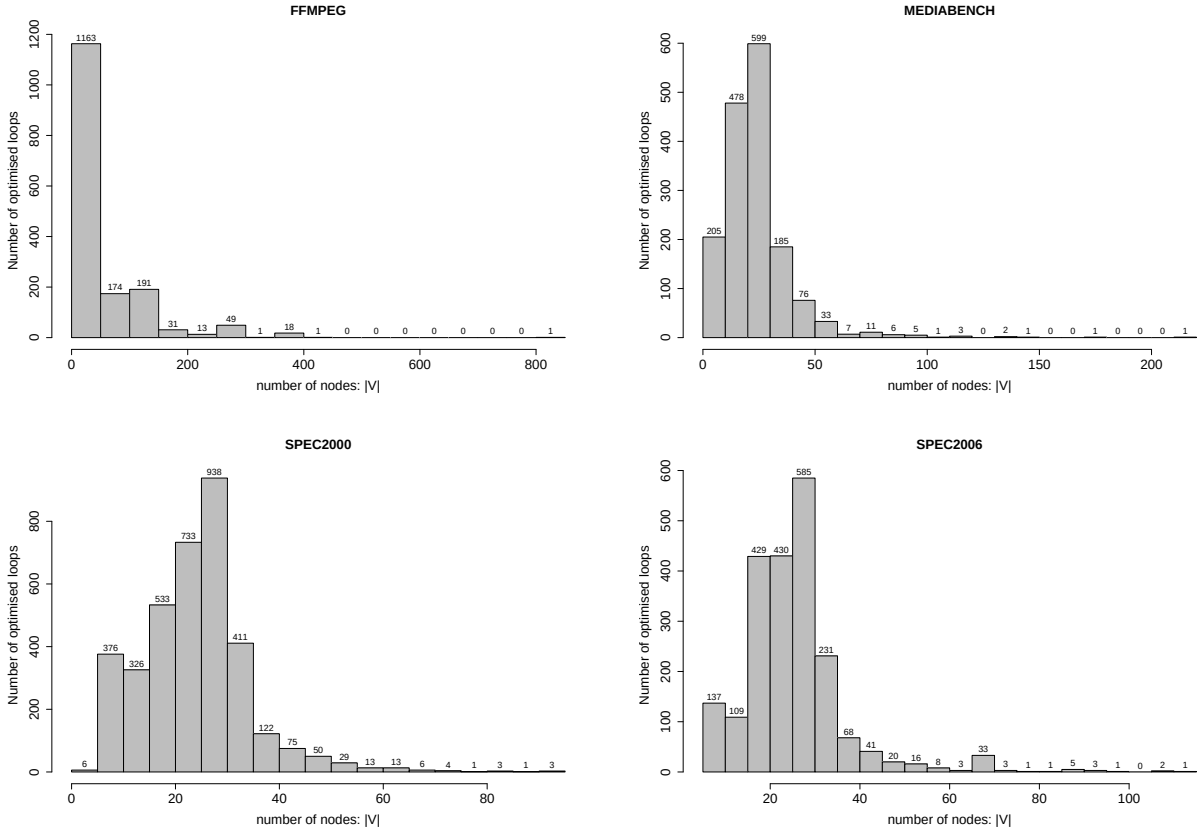


Fig. 2. Histograms on the Number of Nodes (Loop Statements): $|V|$

(25% of the observed compilation times are below this limit), the median value (50% of the observed compilation times are below this limit), the mean compilation time, the third quartile (75% of the observed compilation times are below this limit) and the maximal compilation time per loop. We also compute, thanks to the test of student [Raj Jain. 1991], the confidence interval of the mean with a confidence level equal to 99%. We present below a synthesis of our observation (a graphical comparison between the three alternatives is shown in Figure 8):

—For any scenario, for any benchmark collection, we observe that siralina resolution times are reasonably low (in terms of median an mean). Consequently, our method is fast enough to be considered as a solution for register pressure reduction inside a commercial quality cross compiler such as st200cc.

—For any scenario, for all benchmarks (see last column of Tables I -II -III), we observe that the average resolution time is greater than the third quartile. According to [Raj Jain. 1991], we conclude that the distribution of the resolution times is skewed (has a bias). Consequently, we should not rely on the average (mean) to have a comparative study between the three scenarios, but we must better rely on the quartiles (first quartile, median, third quartile).

—For each scenario, FFMPEG requires more resolution times than the other benchmarks collections. If we compare the FFMPEG's quartiles in each scenario against the quartiles of the other benchmarks, we see that they are always greater or equal. This confirms our assumption observed in Section 5.3 that FFMPEG application brings the most complex cases for register pressure reduction.

—When we consider all benchmarks (last column of Tables I -II -III), we observe that the quartiles of the first scenario (all register types optimised conjointly) are below the quartiles of the other scenarios (when we optimise register types separately). We conclude that, in terms of resolution times, performing SIRALINA on all types conjointly is a better alternative than performing SIRALINA on each register type separately.

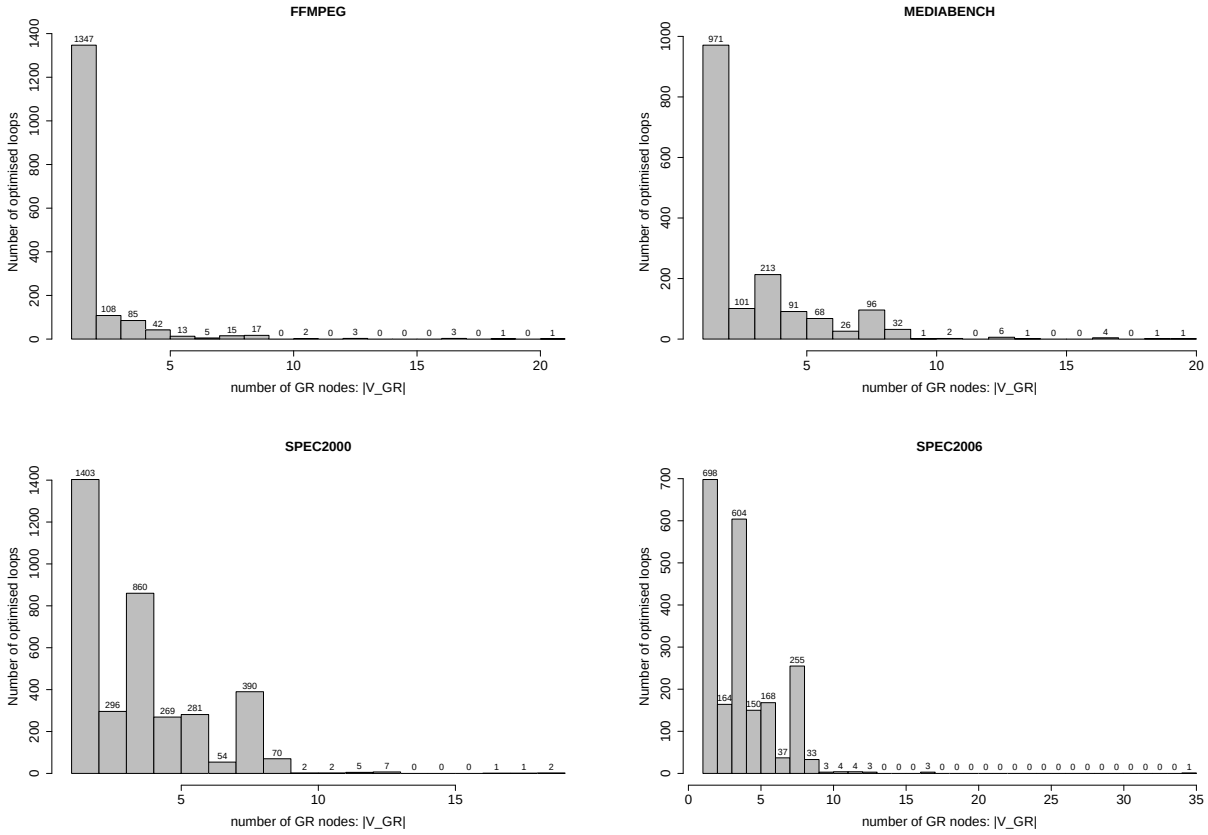


Fig. 3. Histograms on the Number of Statements writing inside General Registers $|V_{GR}|$

Metric	FFMPEG	MEDIABENCH	SPEC2000	SPEC2006	All Benchmarks
Minimal Observed Resolution Time	0.7	0.6	0.7	0.7	0.6
First Quartile Observed Resolution Time	2.3	2.0	2.4	2.4	2.3
Median Observed Resolution Time	7.0	3.4	3.7	3.8	3.8
Mean Observed Resolution Time	133.8	7.3	5.3	6.5	29.3
Third Quartile Observed Resolution Time	43.0	6.2	5.9	6.7	7.4
Maximal Observed Resolution Time	33958.5	521.1	108.8	152.0	33958.5
Mean Confidence Interval (99% of conf.)	[74.1, 193.5]	[6.0, 8.6]	[5.0, 5.6]	[5.9, 7.2]	[18.4, 40.3]

Table I. Resolution Times of SIRALINA per Benchmark Family (in mili seconds): Conjoint Optimisation of BR and GR

Metric	FFMPEG	MEDIABENCH	SPEC2000	SPEC2006	All Benchmarks
Minimal Observed Resolution Time	1.1	0.9	0.9	1.2	0.9
First Quartile Observed Resolution Time	3.6	3.0	3.4	3.0	3.5
Median Observed Resolution Time	8.6	4.9	5.4	5.5	5.4
Mean Observed Resolution Time	111.9	10.0	7.3	9.1	27.2
Third Quartile Observed Resolution Time	34.8	7.9	7.8	8.4	9.3
Maximal Observed Resolution Time	27949.4	1733.7	110.5	214.0	27949.4
Mean Confidence Interval (99% of conf.)	[74.4, 149.5]	[6.9, 13.2]	[6.9, 7.6]	[8.3, 9.9]	[18.2, 36.3]

Table II. Resolution Times of SIRALINA per Benchmark Family (in mili seconds) : Separate Optimisation of GR then BR

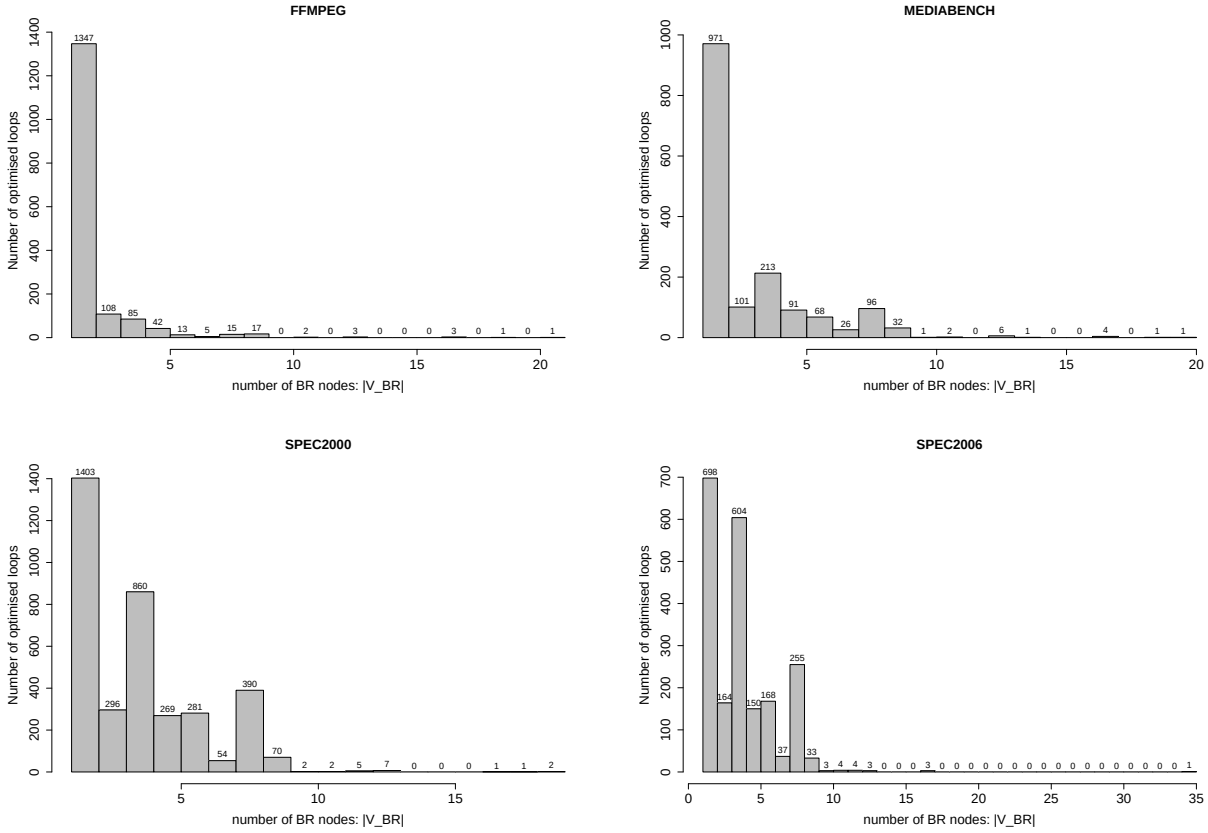


Fig. 4. Histograms on the Number of Statements writing inside Branch Registers $|V_{BR}|$

Metric	FFMPEG	MEDIABENCH	SPEC2000	SPEC2006	All Benchmarks
Minimal Observed Resolution Time	1.0	0.9	0.9	1.2	0.9
First Quartile Observed Resolution Time	3.6	3.0	3.6	3.6	3.5
Median Observed Resolution Time	10.1	4.9	5.2	5.4	5.4
Mean Observed Resolution Time	118.0	10.1	8.0	10.0	28.8
Third Quartile Observed Resolution Time	39.7	9.0	8.3	9.0	10.2
Maximal Observed Resolution Time	28313.0	856.0	208.6	680.0	28313.0
Mean Confidence Interval (99% of conf.)	[67.7, 168.3]	[8.2, 12.1]	[7.5, 8.4]	[8.7, 11.2]	[19.6, 38.1]

Table III. Resolution Times of SIRALINA per Benchmark Family (in mili seconds) : Separate Optimisation of BR then GR

The next section shows statistics on the code quality generated when applying SIRALINA before SWP followed by register allocation.

5.5 Static Performance Results

5.5.1 *Spill Code Reduction.* Without applying SIRALINA, the absolute initial spill statements counts in our SWP loops are: FFMPEG=294, MEDIABENCH=416, SPEC2000=1396, SPEC2006=585. These spill numbers are produced when the st200cc compiler applies SWP followed by register allocation. We then statically measure the amount of spill code reduced thanks to our SIRALINA method. Again, we test three scenarios: SIRALINA with multiple register types, SIRALINA on each register type separately (BR followed by GR, and *vice-versa*). Here, the SIRALINA method is applied before software pipelining, so we add arcs to the DDG to bound MAXLIVE for any software pipelining schedule. The spill code decrease is computed for all SWP loops. It is measured on all loops as $\frac{\text{InitialSpill Count} - \text{ReducedSpillCount}}{\text{InitialSpillCount}}$. Figure 9 plots the spill code decrease in each benchmarks collection: the plotted bars

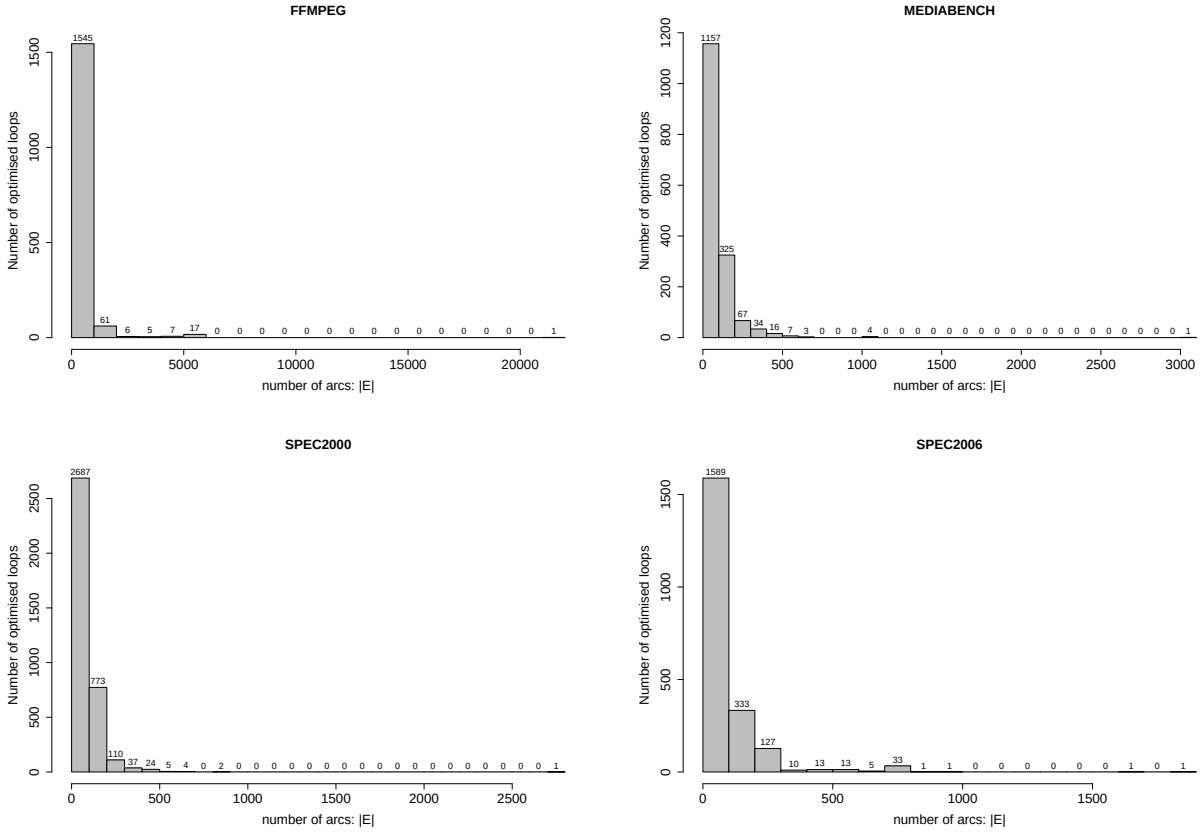


Fig. 5. Histograms on the Number of Data Dependences $|E|$

correspond to FFMPEG, MEDIABENCH, SPEC2000 and SPEC2006 benchmarks. The suffixes BRGR and GRBR on each benchmark name correspond to the case when we apply SIRALINA on each register type separately as studied in [Karine Deschinkel and Sid-Ahmed-Ali Touati 2008]: the label BRGR means that we first optimise BR registers then GR registers, while GRBR corresponds to the opposite order. When neither BRGR or GRBR is specified, it means that both register types have been optimised conjointly by the SIRALINA method explained in Section 4.

For FFMPEG and MEDIABENCH (the representative collection of embedded applications), optimising all register types conjointly reduces the spill and outperforms the case when we optimise each register separately. For SPEC2000 and SPEC2006 (high performance and desktop applications), optimising each register type separately seems to better reduce the spill code. However, it is not clear in which order (BR then GR or GR followed by BR). Because of the complex nature of data dependences, it is not easy to decide about the best order *a priori* for optimising the register types. In this situation, we advise to optimise all the register types conjointly, since it brings significant spill code reduction (between 40% and 50 %) in less compilation time. If compilation time is not an issue, then optimising the register types separately (by exploring all possible orders) using SIRALINA may be a first beneficial solution. Having $|T|$ register types, the number of possible orders is equal to $|T|$!. Usually, $|T|$ is a small value (2 or 3 in general), yielding to small number of distinct orders ($2! = 2$ or $3! = 6$).

From Figure 9 we can clearly conclude that using SIRALINA greatly reduces the amount of spill code. Optimising all types conjointly is more beneficial for MEDIABENCH and FFMPEG rather than SPEC2000 and SPEC2006, where optimising each register type separately seems to be a better choice.

5.5.2 *The Impact on the Crtitical Circuits of the Data Dependance Graphs (MII_{dep}).* The way SIRALINA adds arcs to the DDG before software pipelining may in theory increase the critical circuit of the data dependence graphs. More precisely, it may increase the so called MII_{dep} (minimal initiation interval defined by recurrent dependences). In order to measure this impact, we statically counted the number of DDG in which applying SIRALINA before

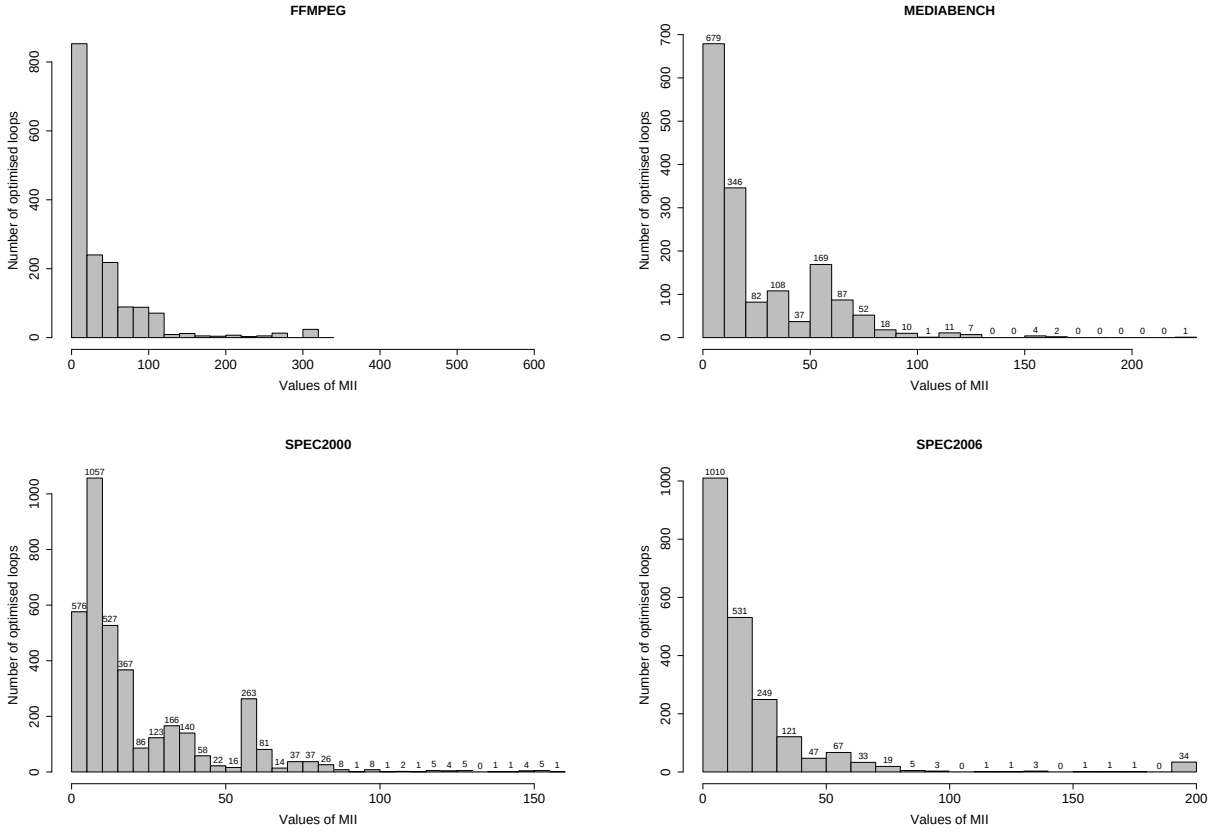


Fig. 6. Histograms on MinII Values

software pipelining increased MII_{dep} or not, see Figure 10. As can be seen, in most of the cases, the critical circuit is not altered. The percentage of loops for which the critical circuit increases or for which spilling is necessary is reasonable.

This subsection showed that adding arcs in the DDG before SWP using SIRALINA does not have significant modification of MII_{dep}. The next section studies the impact on the final SWP schedule quality when we consider both resources and data dependence constraints.

5.5.3 *The Impact on the Initiation Interval (II).* One could think that introducing arcs inside the DDG before software pipelining would also restrict the ILP scheduling, since extra constraints are added. In practice, this is not true because the usual software pipelining heuristics are not optimal. Consequently, adding extra arcs to the DDG can even help the scheduler. It amounts to better II in many cases: by better, we mean that the II computed by the SWP step on the modified DDG (after applying SIRALINA) is less than the II computed by SWP on the initial DDG (without applying SIRALINA). This is a positive unexpected side effect of SIRALINA, since in theory adding arcs to a DDG could alter SWP (while in practice, the SWP heuristic does not really suffer from these added arcs). In Table IV, we measured the average II increase resulting from our constraints on all loops. We computed the mean II increase as $\frac{\sum II_2 - II_1}{\sum II_1}$, where II_2 corresponds to the II computed after software pipelining of the constrained DDG (when applying SIRALINA), and II_1 corresponds to the II computed after software pipelining of the initial DDG (without applying SIRALINA). Except in FFMPEG where II increases in a marginal way (0.5%), all other benchmark families show minor improvement in II. We conclude from Table IV that the II is not really altered when we apply SIRALINA before software pipelining. The improvement of II is not caused by the reduction of spill code (spill code is inserted after SWP), but because the added arcs to the DDG help the SWP heuristic to find better SWP schedule.

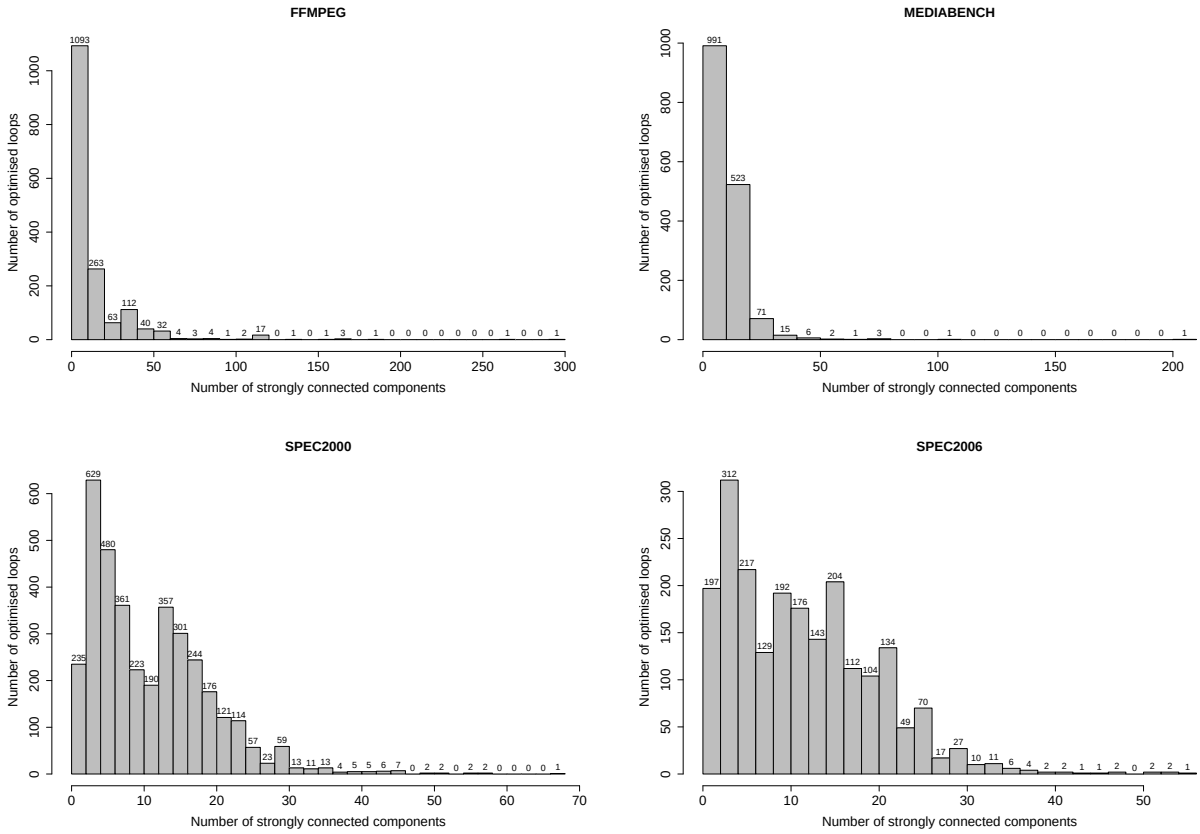


Fig. 7. Histograms on the Numbers of Strongly Connected Components

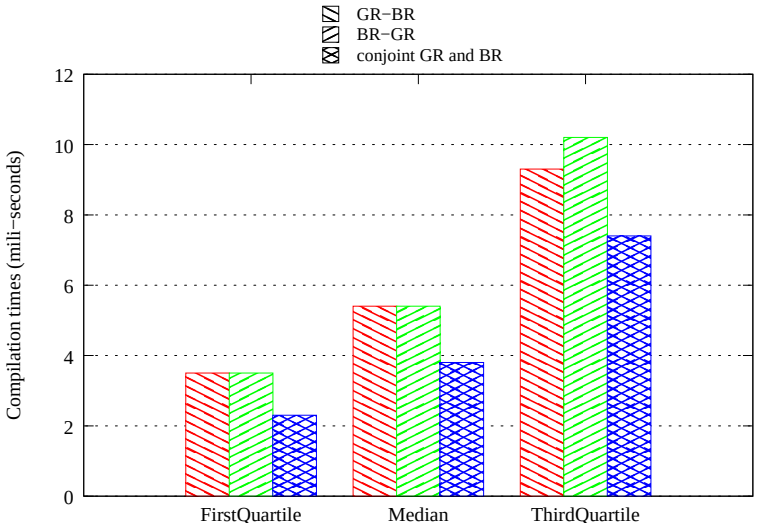


Fig. 8. Graphical comparison for the SIRALINA resolution times (all benchmarks). We clearly observe that optimising all the registers types conjointly is a faster approach than optimising each register type separately.

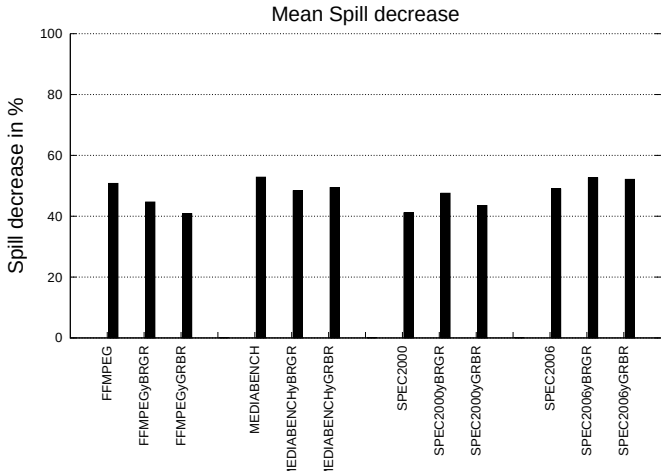


Fig. 9. Percentage of Spill Code Decrease in Each Benchmark Family

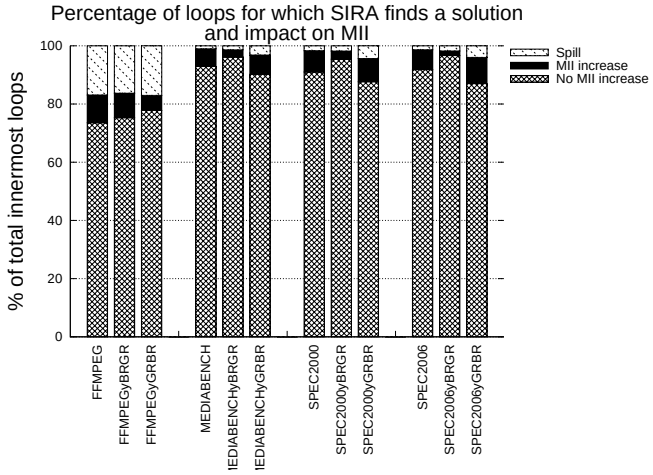


Fig. 10. SIRALINA does not Increase the Critical Circuit (MII_{dep}) in Most of Cases

Benchmark Family	Mean of II Increase
FFMPEG	0.005
FFMPEGyBRGR	0.005
FFMPEGyGRBR	0.004
MEDIABENCH	-0.013
MEDIABENCHyBRGR	-0.014
MEDIABENCHyGRBR	-0.015
SPEC2000	-0.005
SPEC2000yBRGR	-0.006
SPEC2000yGRBR	-0.003
SPEC2006	-0.015
SPEC2006yBRGR	-0.021
SPEC2006yGRBR	-0.017

Table IV. Mean II Increase when Applying SIRALINA before Software Pipelining

Statically, we can say that our method is a success, and this is our main target in the article. We want to reduce

the static number of spill code, without hurting II if possible. The static performance numbers shown in the current section does not favour an execution path against another, since no typical data input are assumed in each optimised application. The next section shows the impact of our spill code reduction on the dynamic speedup.

5.6 Execution Time Performance Results

This section provides performance numbers when we execute the generated binary code on an ST231 VLIW processor, all compiled with `-O3` optimisation level. We warn the reader to remember that some optimised loops may or may not belong to hot execution paths, depending on the application and the chosen program input. This section plots the performance using the standard input of MEDIABENCH and FFMPEG. Other input data sets may exist, bringing distinct speedups for the same applications. Also, depending on the application, software pipelining (SWP) may or may not bring a significant speedup, all depends on the time fraction spent in the software pipelined loops.

5.6.1 *Speedups.* In this section, we report the speedup of the whole application, not the speedups of the individual loops or code kernels optimised by SIRALINA. The best speedup we get is g721 where the execution speed is accelerated by a factor of more than 3, see Figure 11. For the other benchmarks, we notice satisfactory speedups compared to optimising each register type separately in adpcm-decode, gsm-decode, ffmpeg, g721, jpeg-decode, jpeg-encode, pgp-encode. Some cases do not bring significant speedup such as epic and pegwit-decode. Unfortunately, we also have some cases of slowdown such as in adpcm-encode, gsm-decode, mesa and pegwit-encode. We explain below the reason for these slowdowns.

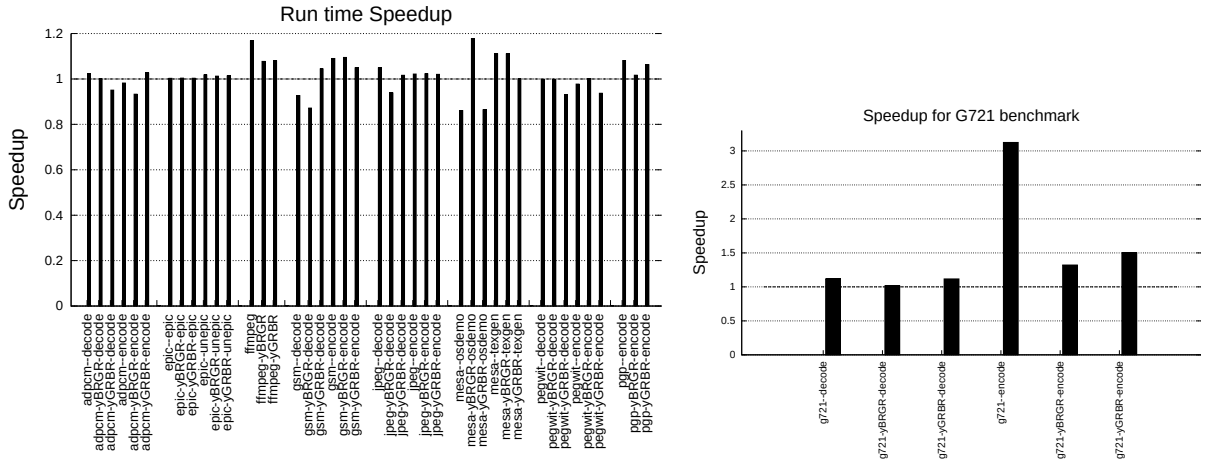


Fig. 11. Program Speedups

5.6.2 *Impact on Icache Effects.* Nowadays, with the numerous code optimisation methods implemented inside optimising compilers, inserting a new code optimisation inside an existing complex compiler suffers from the phase ordering problem and from the interaction between complex phenomena [Sid-Ahmed-Ali Touati and Denis Barthou 2006]. For instance, register allocation seems to be a code optimisation that alters spill code (Dcache effects) and instruction scheduling (ILP extraction). But it also influences the instruction cache behaviour since the instruction schedule is altered. While reducing the amount of spill code reduces the code size, and should in theory improve Icache phenomena, this is not really the case. The reason is that Icache in our embedded VLIW processors is direct mapped. Consequently, Icache conflicts account for a large fraction for Icache misses: depending on the code layout in memory, multiple hot functions and loops may share the same Icache lines, even if their size fit inside the Icache capacity [Christophe Guillon and Fabrice Rastello and Thierry Bidault and Florent Bouchez 2005]. If Icache is fully associative, capacity Icache conflicts could benefit from the reduction of code size, but this is not what happens with direct mapped caches. At our level of optimisation, we have no control on Icache effects when we do register allocation. Other code optimisation methods could be employed to improve the interaction with direct mapped Icache [Christophe Guillon and Fabrice Rastello and Thierry Bidault and Florent Bouchez 2005].

Now, we show the performance numbers that demonstrate that Icache conflicts are the main reason for our slowdowns. We measured the execution time in clock cycles and we characterise it into the five main categories of stalls on ST231: computation + Dcache stalls + Icache stalls + interlock stalls + branch penalties. Figure 12 illustrates the cases of mesa and gsm-decode. The first bar corresponds to the execution time of the code generated without using SIRALINA. The second bar shows the execution time of the code generated when we use SIRALINA, optimising all register types conjointly. he last two bars show the execution times when we apply SIRALINA on each register type separately. We can clearly see that the Icache effects increase in all cases, explaining the origin of the slowdowns.

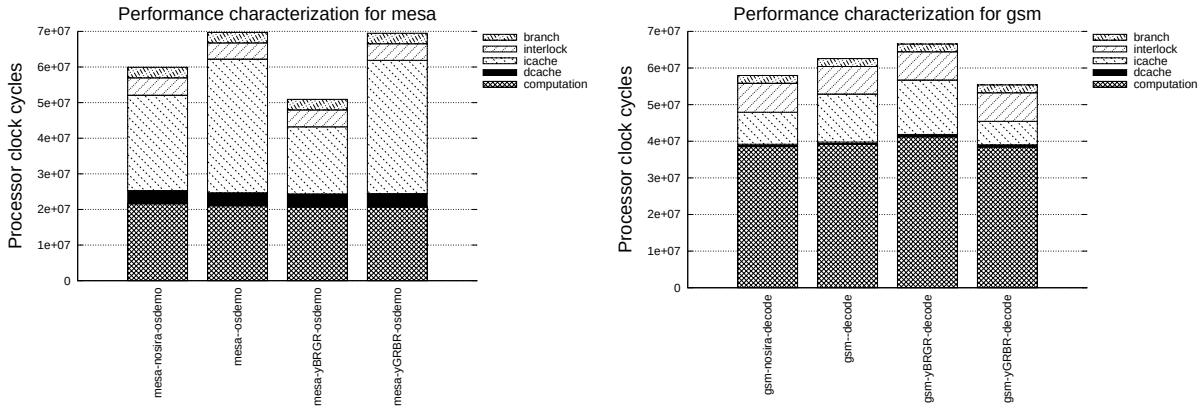


Fig. 12. Performance Characterisation to Explain Slowdowns Due to Icache Effects

6. CONCLUSIONS

We present a new efficient periodic register pressure optimisation technique that simultaneously considers multiple register types. Our experiments on the embedded ST231 VLIW processor cover a significant range of high-performance and media processing benchmarks (FFMPEG, MEDIABENCH,SPEC2000, SPEC2006). These experiments demonstrate the practical applicability and the benefits of our approach: compared to the st200cc production compiler lifetime sensitive software pipelining heuristic, our SIRALINA heuristic avoids the generation of spill code in most of software pipelined loops. When register pressure is too high, inserting spill code becomes necessary. SIRALINA greatly reduces its amount (between 40% and 60% of spill code reduction). In overall, we showed that it is better to optimise all register types conjointly instead of one by one.

The algorithmic complexity of our heuristic is polynomial and equal to $O(n^3 \log n)$. By considering the sample of 9027 optimised SWP loops in FFMPEG, MEDIABENCH,SPEC2000 and SPEC2006, we observe a median compilation time on a linux pentium R desktop equal to 3.8 ms. The $99\%-$confidence interval of the average compilation time is equal to $[18.4ms, 40.3ms]$.

SIRALINA works before software pipelining by adding extra arcs to restrict the software pipeliner by bounding MAXLIVE for any cyclic schedule. In theory, we may alter the extracted II. However, surprisingly enough, restricting data dependence graphs actually help to improve the II. This is because heuristics of software pipelining are not optimal by definition, so adding arcs to the DDG does not hurt in practice. It sometimes helps the scheduler to compute better schedules (lower values for II).

In terms of execution times of the generated binaries, the speedups depend on program input and on complex micro-architectural characteristics. We obtained satisfactory speedups in many benchmarks (up to x3 for g721), but also some slowdowns. We did a careful performance characterisation of the slowdown cases, and we found that they originate from Icache effects. Indeed, periodic register allocation alters the instruction scheduler, which in turn alters the memory layout. Since the Icache of the ST231 is direct mapped, modifying the memory layout of the code greatly impacts Icache conflicts. These phenomena show again that code optimisation is complex, because optimising one aspect of the code may hurt another uncontrolled aspect.

As far as we know, our work on periodic register pressure (SIRA framework) is the only one that handles multiple register types conjointly, with explicit delays in read and writes from/into registers, all based on formal methods

demonstrated efficient in practice. When comparing register allocation techniques, we are faced with difficulties: 1) The source codes of published register allocation techniques are not always made available. 2) Some published articles are not formal enough, or do not contain enough implementation details to reproduce the results. 3) Even if the published articles are detailed enough, it requires too much effort for the community to re-implement already published work. 4) Even if the work has been reimplemented, reproducing the exact results is not so easy because the data dependence graphs (DDG) are not made public, and distinct compilers may generate distinct DDG for the same source code (depending on the internal compilation passes, chosen heuristics, data dependence analysis techniques, intermediate language, target architectures, etc.). So, in order to enable an exact reproducibility of our results, we make public our software implementation in an independent library called `SIRAlib` [Sébastien Briais and Sid-Ahmed-Ali Touati 2009]. This library is under LGPL licence and can be plugged inside any optimising compiler.

Acknowledgement

This research result has been supported by the ANR MOPUCE project (contract number 05-JCJC-0039) and the DIGITEO foundation (contract number 2007-10D). We would like to thank Sébastien Briais from the university of Versailles Saint-Quentin-en-Yvelines for his valuable help to improve our research quality.

REFERENCES

A. SCHRIJVER. 1986. *Theory of Linear and Integer Programming*. Wiley and Sons.

ALEXANDRE E EICHENBERGER AND EDWARD S. DAVIDSON. 1997. Efficient formulation for optimal modulo schedulers. *SIGPLAN Notice 32,* 5, 194–205.

ALEXANDRU NICOLAU AND RONI POTASMAN AND HAIGENG WANG. 1992. Register Allocation, Renaming and Their Impact on Fine-Grain Parallelism. In *Proceedings of the Fourth International Workshop on Languages and Compilers for Parallel Computing*. Springer-Verlag, London, UK, 218–235.

B. RAMAKRISHNA RAU . 1994. Iterative modulo scheduling: an algorithm for software pipelining loops. In *MICRO 27: Proceedings of the 27th annual international symposium on Microarchitecture*. ACM, New York, NY, USA, 63–74.

B. RAMAKRISHNA RAU AND M. LEE AND P. P. TIRUMALAIAND AND MICHAEL S. SCHLANSKER. 1992. Register allocation for software pipelined loops. In *PLDI '92: Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*. ACM, New York, NY, USA, 283–299.

B. RAMAKRISHNA RAU AND MICHAEL S. SCHLANSKER AND P. P. TIRUMALAI. 1992. Code generation schema for modulo scheduled loops. *SIGMICRO Newsl. 23,* 1-2, 158–169.

BENOÎT DUPONT-DE-DINECHIN. 1997. Parametric Computation of Margins and of Minimum Cumulative Register Lifetime Dates. In *LCPC '96: Proceedings of the 9th International Workshop on Languages and Compilers for Parallel Computing*. Springer-Verlag, London, UK, 231–245.

CHRISTINE EISENBEIS AND SYLVAIN LELAIT AND BRUNO MARMOL. 1995. The Meeting Graph: A New Model for Loop Cyclic Register Allocation. In *Proceedings of the IFIP WG 10.3 Working Conference on Parallel Architectures and Compilation Techniques, PACT '95*, L. Bic, W. Böhm, P. Evripidou, and J.-L. Gaudiot, Eds. ACM Press, Limassol, Cyprus, 264–267.

CHRISTOPHE GUILLON AND FABRICE RASTELLO AND THIERRY BIDAULT AND FLORENT BOUCHEZ. 2005. Procedure placement using temporal-ordering information: Dealing with code size expansion. *Journal of Embedded Computing 1,* 4, 437–459.

DOMINIQUE DE WERRA AND CHRISTINE EISENBEIS AND SYLVAIN LELAIT AND BRUNO MARMOL. 1999. On a graph-theoretical model for cyclic register allocation. *Discrete Appl. Math. 93,* 2-3, 191–203.

GEOFFREY FARABOSCH AND JOSEPH A FISHER AND GIUSEPPE DESOLI AND FRED HOMEWOOD. 2000. Lx: a technology platform for customizable VLIW embedded processing. In *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*. ACM, New York, NY, USA, 203–213.

GREGORY CHAITIN. 2004. Register allocation and spilling via graph coloring. *SIGPLAN Notice 39,* 4, 66–74.

JIAN WANG AND CHRISTINE EISENBEIS AND MARTIN JOURDAN AND BOGONG SU. 1994. Decomposed software pipelining: A new perspective and a new approach. *International Journal of Parallel Programming 22,* 3 (June), 351–373.

JOHN RUTTENBERG AND G. R. GAO AND A. STOUTCHININ AND W. LICHTENSTEIN. 1996. Software Pipelining Showdown : Optimal vs. Heuristic Methods in a Production Compiler. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implemantation*. ACM Press, New York, 1–11.

JOSEPH A. FISHER AND PAOLO FARABOSCHI AND CLIFFORD YOUNG. 2005. *Embedded computing: a VLIW approach to architecture, compilers and tools*. Morgan Kaufmann Publishers, pub-MORGAN-KAUFMANN:adr.

KARINE DESCHINKEL AND SID-AHMED-ALI TOUATI. 2008. Efficient Method for Periodic Task Scheduling with Storage Requirement Minimization. In *Combinatorial Optimization and Applications, Second International Conference, COCOA*. Lecture Notes in Computer Science, vol. 5165. Springer, St. John's,NL, Canada, 438–447.

LAM, M. 1988. Software pipelining: an effective scheduling technique for VLIW machines. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*. ACM, New York, NY, USA, 318–328.

LAURIE J HENDREN . AND GUANG R. GAO AND ERIK R ALTMAN AND CHANDRIKA MUKERJI. 1992. A Register Allocation Framework Based on Hierarchical Cyclic Interval Graphs. In *CC '92: Proceedings of the 4th International Conference on Compiler Construction*. Springer-Verlag, London, UK, 176–191.

PRESTON BRIGGS AND KEITH D. COOPER AND LINDA TORCZON. 1994. Improvements to graph coloring register allocation. *ACM Transaction Programming Languages and Systems. 16,* 3, 428–455.

RAJ JAIN. 1991. *The Art of Computer Systems Performance Analysis : Techniques for Experimental Design, Measurement, Simulation, and Modelling*. John Wiley and Sons, New York.

RAVINDRA K. AHUJA RAVINDRA AND THOMAS L. MAGNANTI AND JAMES B. ORLIN. 1991. *Network Flows: theory, algorithms, and applications.* John Wiley and Sons, New York.

SANTOSH G. NAGARAKATTE AND R. GOVINDARAJAN. 2007. Register Allocation and Optimal Spill Code Scheduling in Software Pipelined Loops Using 0-1 Integer Linear Programming Formulation. In *Compiler Construction, 16th International Conference,CC 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS*. Lecture Notes in Computer Science, vol. 4420. Springer, Braga, Portugal, 126–140.

SÉBASTIEN BRIAIS AND SID-AHMED-ALI TOUATI. 2009. Schedule-Sensitive Register Pressure Reduction in Innermost Loops, Basic Blocks and Super-Blocks. Tech. Rep. INRIA-HAL-00436348, University of Versailles Saint-Quentin en Yvelines. Nov. http://hal.archives-ouvertes.fr/inria-00436348.

SID-AHMED-ALI TOUATI. 2007. On the Periodic Register Need in Software Pipelining. *IEEE Transactions on Computers 56,* 11, 1493–1504.

SID-AHMED-ALI TOUATI AND CHRISTINE EISENBEIS. 2004. Early Periodic Register Allocation on ILP Processors. *Parallel Processing Letters 14,* 2 (June), 287–??

SID-AHMED-ALI TOUATI AND DENIS BARTHOU. 2006. On the Decidability of Phase Ordering Problem in Optimizing Compilation. In *ACM Proceedings of the International Conference on Computing Frontiers*. Ischia, Italy.