Schedule-Sensitive Register Pressure Reduction in Innermost Loops, Basic Blocks and Super-Blocks

Sébastien Briais, Sid Touati

UNIVERSITE DE VERSAILLES SAINT-QUENTIN EN YVELINES

Schedule-Sensitive Register Pressure Reduction in Innermost Loops, Basic Blocks and Super-Blocks

Sébastien Briais — Sid-Ahmed-Ali Touati

N° HAL-INRIA-00436348

November 2009

RR--HAL-INRIA-00436348--FR+ENG

Schedule-Sensitive Register Pressure Reduction in Innermost Loops, Basic Blocks and Super-Blocks

Sébastien BRIAIS[*] , Sid-Ahmed-Ali TOUATI[†]

Thème : Optimisation de programmes
Équipe-Projet ARPA - Laboratoire PRiSM

Rapport de recherche n° HAL-INRIA-00436348 — November 2009 — 53 pages

Abstract: This report makes a massive experimental study of an efficient heuristic for the SIRA framework [11]. The heuristic, called SIRALINA [4], bounds the register requirement of a data dependence graph before instruction scheduling under resource constraints. Our aim is to guarantee the absence of spilling before any instruction scheduling process, without hurting instruction level parallelism if possible. Our register pressure reduction methods are sensitive for both software pipelining (innermost loops) and acyclic scheduling (basic blocks and super-blocks). The SIRALINA method that we experiment in this report is shown efficient in terms of compilation times, in terms of register requirement reduction and in terms of shorted schedule increase. Our experiments are done on thousands standalone DDG extracted from FFMPEG, MEDIABENCH, SPEC2000 and SPEC2006 benchmarks. We consider processor architectures with multiple register type and we model delayed access times to registers. Our register pressure reduction method is distributed as a C independent library (`SIRAlib`.

Key-words: Compilation, Code optimisation, Register pressure, Instruction level parallelism

[*] Sebastien.Briais@prism.uvsq.fr
[†] Sid.Touati@uvsq.fr

UFR des sciences
45 avenue des Etats Unis,
78035 Versailles cedex
Téléphone : +33 1 39 25 41 12 — Télécopie : +33 1 39 25 46 04

La réduction de la pression sur les registres pour les boucles internes, les blocs de base et les super-blocs

Résumé : Ce rapport dresse une étude expérimentale massive et complète d'une heuristique efficace de la plate-forme SIRA [11]. L'heuristique est appelée SIRALINA [4]. Elle borne la consommation en registres d'un graphe de dépendances de donnés avant tout ordonnancement d'instructions sous contraintes de ressources. Notre but est de garantir l'absence de code de vidage avant l'ordonnancement d'instructions. Nos méthodes de réduction de la pression en registres sont destinées à fonctionner an amont du pipeline logiciel (boucles interne) et de l'ordonnancement acyclique (blocs de bases et super-blocs). Nous montrons expérimentalement que la méthode SIRALINA est efficace en terme de temps de compilation, en terme de réduction du besoin en registres et en terme de l'augmentation des plus courts ordonnancements possibles. Nos expériences ont été conduites sur des milliers de graphes de dépendances de données extraits des benchmarks FFMPEG, MEDIABENCH, SPEC2000 et SPEC2006. Nous considérons des architectures de processeurs avec plusieurs types de registres, avec la possibilité de modéliser des temps d'accès décalés aux registres. Notre méthode de réduction de la pression en registres est distribuée dans une librairie C indépendante (`SIRAlib`).

Mots-clés : Compilation, optimisation de code, pression en registres, parallélisme d'instructions

Contents

Introduction

In this report we propose a new research achievement in the context of our *SIRA* framework [11] (*Schedule-Independent Register Allocation*). The SIRA framework is used to pre-condition the data dependence graph (DDG) before software pipelining in order to guarantee that the register pressure (MAXLIVE) created by any instruction schedule does not exceed the number of available registers. In case of innermost loops, this guarantees that a spill-free register allocation exists. 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.

An efficient heuristic of SIRA, called *SIRALINA*, has been published in [4]. This efficient heuristic outperforms the previous ones published in [11] both in terms of execution speed and efficiency. In addition to loops, we use SIRALINA to bound the register pressure inside directed acyclic graphs instead of cyclic ones. This allows us to reduce the register pressure in basic blocks and super-blocks devoted to acyclic instruction scheduling.

SIRALINA is formulated as two-steps heuristic. The first step is based on integer linear programming and the second step is algorithmic. In some situations, it is not admitted to include a linear solver inside a compiler. Consequently, we study in this report how to convert the integer linear program to a min-cost flow problem, that can be solved with a min-cost flow algorithm. This is possible because the constraints matrix of the linear program of SIRALINA is totally unimodular.

In this report, we deeply analyse the numerical results obtained after massive experimentations of SIRALINA on numerous benchmarks from FFMPEG, MEDIABENCH, SPEC2000 and SPEC2006. SIRALINA considers the optimisation of multiple register types conjointly, with delayed access to registers.

Our report is organised as follows. In Chapter 1, we recall briefly the SIRA framework and SIRALINA heuristic. In Chapter 2, we analyse the experimental results of SIRALINA on cyclic data dependency graphs (loops). In Chapter 3, we describe and experiment the adaptation of SIRALINA heuristic to reduce the register pressure of acyclic data dependency graph. Finally, before concluding, we study the min-cost flow implementation of SIRALINA in Chapter 4.

Chapter 1

Background on the SIRA framework

In this chapter, we recall briefly the SIRA framework[11] and SIRALINA heuristic[4]. Before that, let us start by defining some notations used in this report.

1.1 Mathematical notations and definitions

In this section, we introduce mathematical notations and definitions that are used afterwards.

Directed multi-graphs

A directed multi graph is a triple (V, E, ϕ) where V is a set of *vertices*, E is a set of *directed edges* and $\phi : E \rightarrow V \times V$. If $e \in E$ and $\phi(e) = (u, v)$, we define $source(e) = u$ and $target(e) = v$. In the sequel, we omit the ϕ component when manipulating graphs. Hence, a graph is just a pair (V, E) and we assume to have two functions $source : E \rightarrow V$ and $target : E \rightarrow V$ that define the *endpoints* of any edge $e \in E$.

By abuse of notation, we sometimes write $e = (u, v)$ when $source(e) = u$ and $target(e) = v$. Given a graph $G = (V, E)$ we define:

- $\Gamma^-_G(u) = \{source(e) \mid e \in E \wedge target(e) = u\}$ the set of predecessors of a vertex $u \in V$ in the graph G.

  The *input degree* of $u \in V$ is $\deg^-_G(u) = |\Gamma^-(u)|$.

  $u \in V$ is a *source iff* $\deg^-_G(u) = 0$.

- $\Gamma^+_G(u) = \{target(e) \mid e \in E \wedge source(e) = u\}$ the set of successors of a vertex $u \in V$ in the graph G.

  The *output degree* of $u \in V$ is $\deg^+_G(u) = |\Gamma^+(u)|$.

  $u \in V$ is a *sink iff* $\deg^+_G(u) = 0$.

- $In_G(u) = \{e \in E \mid target(e) = u\}$ the *input edges* of a vertex $u \in V$ in the graph G.

- $Out_G(u) = \{e \in E \mid source(e) = u\}$ the *output edges* of a vertex $u \in V$ in the graph G.

- A path in G is a sequence of edges $e_1, \ldots, e_n$ where $target(e_i) = source(e_{i+1})$ for $1 \leq i < n$ and $e_i \in E$ for $1 \leq i \leq n$.

  G is *acyclic iff* there are no path $e_1, \ldots, e_n$ in G such that $target(e_n) = source(e_1)$. Otherwise G is cyclic. Note that an acyclic graph has at least one source and one sink.

  A *cycle* in G is a path $e_1, \ldots, e_n$ in G such that $target(e_n) = source(e_1)$. We note $\mathcal{C}(G)$ the set of all the cycles in G. Hence G is acyclic *iff* $\mathcal{C}(G) = \emptyset$.

- Given a set of edges E' such that for any $e \in E'$, $source(e) \in V$ and $target(e) \in V$, the extended graph $G\backslash^{E'}$ is $(V, E \cup E')$.

Network flows

A network is characterised by a graph $G = (V, E)$ and three functions:

1. $lcap : E \to \mathbb{N}$ defines lower capacities of edges,
2. $ucap : E \to \mathbb{N}$ defines upper capacities of edges,
3. $b : V \to \mathbb{Z}$ defines supply or demand for each vertex.

A flow in a network is a function $f : E \to \mathbb{N}$ such that:

- Capacity constraints are satisfied, i.e.

$$\forall e \in E : 0 \leq lcap(e) \leq f(e) \leq ucap(e)$$

- Supply and demand constraints are satisfied, i.e.

$$\forall v \in V : \sum_{e \in Out(v)} f(e) - \sum_{e \in In(v)} f(e) = b(v)$$

Given a cost function $cost : E \to \mathbb{Z}$, the cost of a flow is $\sum\limits_{e \in E} cost(e) \times f(e)$.

The next section defines our processor and loop data dependency graph model;.

1.2 DDG and processor model

A loop *data dependency graph* (DDG) is a quadruple $G = (V, E, \delta, \lambda)$ where V is a set of vertices (loop statements), E is a set of edges (data dependencies and serial constraints between statements) such that (V, E) is a directed graph, $\delta : E \to \mathbb{Z}$ gives the latency of edges and $\lambda : E \to \mathbb{Z}$ defines the distance in terms of number of iterations. By abuse of notation, we sometimes write G for the underlying graph (V, E). Each operation u has a (strictly) positive latency $lat(u) \in \mathbb{N}$.

A DDG is said *lexicographic positive iff* it has no cycle of negative distance, i.e. for any $c \in \mathcal{C}(G)$, $\lambda(c) = \sum\limits_{e \in c} \lambda(e) > 0$. In the sequel, we assume that the initial DDGs are lexicographic positive, because they are created from a data dependence analysis of a sequential program.

A *software pipelining* is defined by a *periodic schedule function* $\sigma : V \to \mathbb{N}$ and an *initiation interval* II. The operation u of the k^{th} iteration is scheduled at time $\sigma(u) + k \times \text{II}$. The schedule function is valid *iff* it satisfies the usual precedence constraints:

$$\forall e \in E : \sigma(source(e)) + \delta(e) \leq \sigma(target(e)) + \lambda(e) \times \text{II}$$

If G is cyclic, one necessary condition for a valid schedule to exist is that

$$\text{II} \geq \max_{c \in \mathcal{C}(G)} \frac{\sum\limits_{e \in c} \delta(e)}{\sum\limits_{e \in c} \lambda(e)} = \text{MII}$$

MII is the usual the *minimum initiation interval*, $MII = \max_{c \in \mathcal{C}(G)} \frac{\sum_{e \in c} \delta(e)}{\sum e \in c \lambda(e)}$. If G is acyclic, then it is always schedulable with a periodic schedule and we define $\text{MII} = 1$.

The modelled target processor can have several register types (sometimes called register classes). We note $\mathcal{T}$ the set of register types. An operation which stores a value in a register of type $t \in \mathcal{T}$ is simply said to be a *value* of type t. Note that an instruction can produce multiple values of several types simultaneously. However, our processor model does not support operations that store two or more values in the same register type.

For a given type $t \in \mathcal{T}$, we note $V^{R,t}$ the set of operations $v \in V$ that are values of type t. The set of edges E is partitioned in two parts: *flow* edges of type t —written $E^{R,t}$— and *serial* edges. The set of flow edges $E^{R,t}$ model the values to be stored inside registers of type t.

The set of *consumers* (readers) of a value $u \in V^{R,t}$ is

$$Cons^t(u) = \{target(e) \mid e \in E^{R,t} \wedge source(e) = u\}$$

Given a type $t \in \mathcal{T}$, we model possible delays when reading or writing registers of type t with two delay functions $\delta_{r,t}$ and $\delta_{w,t}$. Thus, the read cycle of u from a register of type t is $\sigma(u) + \delta_{r,t}(u)$ and the write cycle into a register of type t is $\sigma(u) + \delta_{w,t}(u)$.

1.3 Schedule Independent Register Allocation (SIRA)

As argued in [11], performing register allocation before scheduling enjoys several benefits, especially regarding the absence of spilling without hurting instruction level parallelism. A theoretical framework has thus been presented in [11] to address this problem. In the following, we briefly recall this framework.

Given a DDG $G = (V, E)$, a desired initiation interval II and a register type $t \in \mathcal{T}$, the problem amounts to finding a valid *reuse graph* $(V^{R,t}, E^{\text{reuse},t}, \mu)$ that minimises $\mu_t = \sum_{e_r \in E^{\text{reuse},t}} \mu(e_r)$. Once the reuse graph computed, then we can build G' and extension of the DDG G, which is associated to the reuse graph. As proved in [11], G' has two properties: (1) its critical circuit (MII) does not exceed II (2) Any software pipelining schedule of G' cannot require more than μ_t registers of type t.

If constructing a valid reuse graph with a period II is not possible, we can try for another value of II: either increment II or use a binary search on II. Building a reuse graph that minimises μ_t for a fixed period II is NP-complete.

A reuse graph for the register type t is a triple $(V^{R,t}, E^{\text{reuse},t}, \mu)$ where $V^{R,t}$ is the set of values of type t, $E^{\text{reuse},t}$ is the set of *reuse edges* and $\mu : E^{\text{reuse},t} \rightarrow \mathbb{Z}$ gives the reuse distance of the reuse edges. A reuse graph is *valid iff* G' is schedulable and $(V^{R,t}, E^{\text{reuse},t})$ is the graph of a bijection (i.e. $|\,In(u)| = |\,Out(u)| = 1$ for any $u \in V^{R,t}$).

The associated DDG G' is obtained from G as follows. For any reuse edge $e_r = (u, v) \in E^{\text{reuse},t}$,

- if $Cons^t(u) \neq \emptyset$, then, for any $c_u \in Cons^t(u)$, add an anti-dependency edge $e_a = (c_u, v)$ with a latency of $\delta(e_a) = \delta_{r,t}(c_u) - \delta_{w,t}(v)$ and a distance of $\lambda(e_a) = \mu(e_r) - \max_{e=(u,c_u) \in E^{R,t}} \lambda(e)$.
- otherwise, add an anti-dependency edge $e_a = (u, v)$ with a latency of $\delta(e_a) = \delta_{r,t}(u) - \delta_{w,t}(v)$ and a distance of $\lambda(e_a) = \mu(e_r)$.

In both cases, filter out anti-dependency edges e_a such that $source(e_a) = target(e_a)$ and $\lambda(e_a) = 0$, because they represent redundant precedence constraints (satisfied by definition).

The SIRA framework opens the opportunities to design many heuristics for the construction of valid reuse graphs. In [11] we proposed methods for fixing reuse edges (to limit the combinatorics of the problem). The following section describes a better approach.

1.4 SIRALINA heuristic for a single register type

Computing a valid reuse graph for a fixed period II that minimises μ_t is NP-complete [9]. SIRALINA [4] is an efficient heuristic for this problem. We now review briefly this method. In this section, we consider the optimisation of the register requirement of a single register type t. We will see in a subsequent section how to generalise to multiple register types.

SIRALINA decomposes the problem into two polynomial steps summarised as follows (here, the period II is fixed):

1. Step 1: Determine minimal reuse distances for all pairs of values (i.e. compute a function $\overline{\mu} : V^{R,t} \times V^{R,t} \rightarrow \mathbb{Z}$);
2. Step 2: Determine a bijection $E^{\text{reuse},t} : V^{R,t} \rightarrow V^{R,t}$ that minimises $\sum_{e_r \in E^{\text{reuse},t}} \mu(e_r)$.

These two steps allows the construction of a reuse graph for a period II. Then G' the associated DDG is constructed. The two following sections details each of the two above steps.

1.4.1 Step 1: the scheduling problem for a fixed II

The scheduling problem is formulated as an integer linear problem with totally unimodular constraints matrix. It aims at determining minimal reuse distances for all pairs of values. The linear program satisfies all scheduling constraints, consequently the constructed reuse graph will be necessarily valid.

Integer variables of the linear problem
For any $u \in V$, define a variable σ_u representing a scheduling date. For any $u \in V^{R,t}$, define a variable $\sigma_{k_u^t}$ representing its killing date.

Linear program formulation
The scheduling problem is expressed as follows:

$$\begin{cases} \text{minimise} & \displaystyle\sum_{u \in V^{R,t}} \sigma_{k_u^t} - \sum_{u \in V^{R,t}} \sigma_u \\ \text{subject to} & \\ & \text{for any } e = (u,v) \in E \\ & \quad \sigma_v - \sigma_u \geq \delta(e) - \text{II} \times \lambda(e) \\ & \\ & \text{for any } e = (u,v) \in E^{R,t} \\ & \quad \sigma_{k_u^t} - \sigma_v \geq \delta_{r,t}(v) + \text{II} \times \lambda(e) \\ & \\ & \text{for any } u \in V^{R,t} \text{ such that } Cons^t(u) = \emptyset \\ & \quad \sigma_{k_u^t} - \sigma_u \geq 1 \end{cases}$$

The constraints matrix of this integer linear program is an incidence matrix of the DDG G, consequently it is totally unimodulatr. So hence it can be solved with a polynomial algorithm.

Let ${\sigma_u}^*$ and ${\sigma_{k_u^t}}^*$ be the values of the variables of the optimal solution of the above scheduling problem. The minimal reuse distance function is then defined as follows for all pairs of values (u, v).

$$\overline{\mu_{u,v}^t} = \left\lceil \frac{{\sigma_{k_u^t}}^* - \delta_{w,t}(v) - {\sigma_v}^*}{\text{II}} \right\rceil$$

1.4.2 Step 2: the assignment problem

The assignment problem is to find a bijection $\theta : V^{R,t} \rightarrow V^{R,t}$ such that $\sum_{u \in V^{R,t}} \overline{\mu_{u,\theta(u)}^t}$ is minimal. It can be solved in polynomial time with the so-called Hungarian algorithm[6].

Such an optimal bijection θ defines a set of reuse edges $E^{\text{reuse},t}$ as follows.

$$E^{\text{reuse},t} = \{(u, \theta(u)) \mid u \in V^{R,t}\}$$

After executing the two steps of SIRALINA, a reuse graph is constructed. If μ_t exceeds the number of available registers of type t, then we must try for another value of II: either increment II or use a binary search on II. If II reaches a maximal value without having a satisfactory μ_t, then we say that SIRALINA did not find a solution, spilling becomes necessary (no enough registers are available).

Non uniqueness of the solution of SIRALINA A careful reader may have noticed that the set of reuse edges computed by siralina are not unique in general. There are basically two reasons for this. Firstly, there might be several bijections θ that are optimal solutions of the assignment problem. Secondly, and more importantly, the assignment problem formulation depends itself on the optimal solution found for the scheduling problem. Since there might be several optimal solutions to this problem, which only share the value of the objective function, there are several possible assignment problems. This problem is not theoretically annoying since it is common that the solutions for many optimisation problems are not unique.

1.5 SIRALINA for multiple register types

Imagine that register allocation problem has to be solved for multiple register types $t_1, t_2, \ldots, t_k$. One possibility is to solve this problem by first applying SIRALINA with $t = t_1$, then, continuing with the modified DDG in place of G, applying SIRALINA with $t = t_2$, and so on. The final DDG obtained depends of course on the order in which types have been treated (e.g. $t_1, t_2, \ldots$ and $t_2, t_1, \ldots$ might produce different results).

Another possibility is to solve the problem simultaneously for all the types. SIRALINA heuristic can indeed be easily generalised to treat several types at once.

Let $\mathcal{T}$ be the set of types of interest. SIRALINA heuristic is generalised by (1) solving a generalised scheduling problem and (2) solving $|\mathcal{T}|$ independent assignment problems, one for each $t \in \mathcal{T}$.

The scheduling problem is generalised as follows.

$$\left\{\begin{array}{lll}
\text{minimise} & & \displaystyle\sum_{t\in\mathcal{T}} \alpha_t \left(\sum_{u\in V^{R,t}} \sigma_{k_u^t} - \sum_{u\in V^{R,t}} \sigma_u \right) \\
\text{subject to} & & \\
& \multicolumn{2}{l}{\text{for any } e = (u,v) \in E} \\
& & \sigma_v - \sigma_u \geq \delta(e) - \mathrm{II} \times \lambda(e) \\
& \multicolumn{2}{l}{\text{for any } t \in \mathcal{T}, \text{ for any } e = (u,v) \in E^{R,t}} \\
& & \sigma_{k_u^t} - \sigma_v \geq \delta_{r,t}(v) + \mathrm{II} \times \lambda(e) \\
& \multicolumn{2}{l}{\text{for any } t \in \mathcal{T}, \text{ for any } u \in V^{R,t} \text{ such that } Cons^t(u) = \emptyset} \\
& & \sigma_{k_u^t} - \sigma_u \geq 1
\end{array}\right.$$

This generalised problem is parametrised by α_t coefficients (for $t \in \mathcal{T}$). These allow to focus on some types at the expense of others. For example, to model that type t_1 is more important than type t_2, one can choose $\alpha_{t_1} > \alpha_{t_2}$.

If R_t is the number of available registers of type t, we say that SIRALINA finds a solution for a given II if $\forall t \in \mathcal{T}, \mu_t \leq R_t$. The next chapter makes a full experimental study about the efficiency of SIRALINA as a standalone tool.

Chapter 2

Experimental study of bounding the periodic register pressure in loop data dependence graphs

In this chapter, we report experimental results of SIRALINA heuristic. Let us start by describing the experimental setup that served to obtain the numerical results of this report.

2.1 Experimental setup

2.1.1 The data dependency graphs used as input test data

The data dependency graphs (DDG) used for the experiments come from SPEC2000, SPEC2006, MEDIABENCH and FFMPEG sets of benchmarks. They have been computed by the `st200cc` compiler (industrial compiler from STmicroelectronics, based on `Open64` compiler). We capture all the DDG going to the software pipelining module, after super-block formation. The number of DDG per benchmark family[1] is the following.

MEDIABENCH	SPEC2000	SPEC2006	FFMPEG	ALL
1592	3841	1274	2030	8737

The compiler has been configured to consider a target processor architecture with three types of registers, namely FP (floating point registers), GR (general registers) and BR (branch registers). Thus $\mathcal{T} = \{\mathrm{FP}, \mathrm{GR}, \mathrm{BR}\}$. The number of benchmarks per family involving these types is the following.

Type	MEDIABENCH	SPEC2000	SPEC2006	FFMPEG	ALL
FP	313	317	87	47	764
GR	1592	3838	1274	2030	8734
BR	1592	3841	1274	2030	8737

The following resource constraints hold on the target architecture of st200cc (ST2xx processor family):

- Up to 4 operations per cycle.
- The processing time of any operation is a single clock cycle, while the latencies between operations range from 0 to 3 cycles;
- A maximum of one control operation (`goto`, `jump`, `call`, `return`), one memory operation (`load`, `store`, `prefetch`), and two multiply operations per cycle.

The distribution of the sizes of the DDGs (the number of vertices) is in the following table. MIN stands for MINimum, FST for FirST quantile (25% of the population), MED for MEDian (50% of the population), THD for THirD quantile (75% of the population) and MAX for MAXimum.

[1] ALL indicates the whole set of benchmarks

	MEDIAB.	SPEC2000	SPEC2006	FFMPEG	ALL
MIN	3	3	5	4	3
FST	10	12	16	18	13
MED	16	22	24	37	24
THD	28	28	30	111	32
MAX	212	163	212	783	783

The distribution of the number of values per type is the following.

Type		MEDIAB.	SPEC2000	SPEC2006	FFMPEG	ALL
FP	MIN	1	1	1	1	1
	FST	2	3	3	2	2
	MED	4	6	4	5	4
	THD	8	14	12	8	12
	MAX	68	72	132	32	132
GR	MIN	1	1	1	2	1
	FST	6	7	8	12	7
	MED	9	12	12	29	12
	THD	16	17	18	105	21
	MAX	208	81	74	749	749
BR	MIN	1	1	1	1	1
	FST	1	1	1	1	1
	MED	1	3	3	1	1
	THD	3	5	4	1	4
	MAX	21	27	35	139	139

2.1.2 Running hardware for our experimental study

The computers used for experiments were Intel based PC. The typical configuration was Core 2 Duo PC at 1.6 GHz, running GNU/Linux 64 bits (kernel 2.6), with 4 Gigabytes of main memory.

2.1.3 Naming conventions for register optimisation orders

In this report, we experiment many configurations for register optimisation. Typically, the order of register types used for optimisation is a topic of interest. For $\mathcal{T} = \{t_1, \dots, t_n\}$ a set of register types, and $p : [\![1; n]\!] \to [\![1; n]\!]$ a permutation, we note $\mathcal{O} = t_{p(1)}; t_{p(2)}; \dots; t_{p(n)}$ the register type optimisation order consisting in optimising the registers sequentially for the types $t_{p(1)}$, $t_{p(2)}$, $\dots$, $t_{p(n)}$ in this order. We note $\mathcal{O} = t_1 t_2 \dots t_n$ (or indifferently any other permutation) when no order is relevant (i.e. types $\{t_1, \dots, t_n\}$ altogether): by abuse of language, we also call this a register optimisation order.

Example: Assume that $\mathcal{T} = \{FP, GR, BR\}$. Then:

- FP;GR;BR is the register optimisation order which focus first on FP type, then on GR type and finally on BR type.
- FP;BR;GR is the register optimisation order which focus first on FP type, then on BR type and finally on GR type.
- FP GR BR is the register optimisation order where all the types are solved simultaneously. It is equivalent to FP BR GR, to GR FP BR, ...

2.2 Experimental results of SIRALINA

We have implemented the SIRALINA heuristic as an independent C library (`SIRAlib`). The implementation relies on the GNU Linear Programming Kit Library[5] for solving the scheduling problem.

We have modelled three possible target processor architectures by setting the following register constraints (R_t is the number of available registers of type t).

Configuration of the target architecture	R_{FP}	R_{GR}	R_{BR}
Configuration 1: Small architecture	32	32	4
Configuration 2: Medium architecture	64	64	8
Configuration 3: Large architecture	128	128	8

For each architectural configuration, and each DDG G, we determined whether SIRALINA is able to find a solution with the smallest II. We thus measured:

- Whether a solution exists or not.
- The elapsed time needed to determine whether a solution exists;
- The smallest II for which a solution exists (when applicable).

We did these experiments with every SIRALINA register optimisation orders that involve the three register types.

2.2.1 Experimental efficiency of SIRALINA

For each architectural configuration, for each register types order, Figure 2.1 illustrates the percentage of solutions found by SIRALINA and the percentage of DDG that need spilling: we say that SIRALINA finds a solution for a given DDG it it finds a value for MII (which is the value of II in the SIRALINA linear program) such that all the register requirements of all registers types are below the limit imposed by the processor architecture: $\forall t \in \mathcal{T}, \mu_t \leq R_t$. Each barre of the figure represents a register optimisation order as defined in Section 2.1.3. Figure 2.1 also shows, in the case where a solution exists, whether the critical circuit (MII) has been increased or not compared to its initial value.

We note that SIRALINA found most of the time a solution that satisfied the architectural constraints. Of course, the percentage of success increases when the number of architectural registers is greater. Thus SIRALINA succeeds in about 95% to find a solution for the small architecture and in almost 100% for the large architecture.

We also observe that the proportion of cases for which a solution was found for II = MII is between 60% and 80%, depending on the benchmark family and the siralina register optimisation order. Thus the performance of the software pipelining would not suffer from the extension of the DDG made after applying SIRALINA.

Finally, the simultaneous register optimisation order FPGRBR gives very good results, often better than the results obtained with the sequential orders.

2.2.2 Measuring the increase of the MII

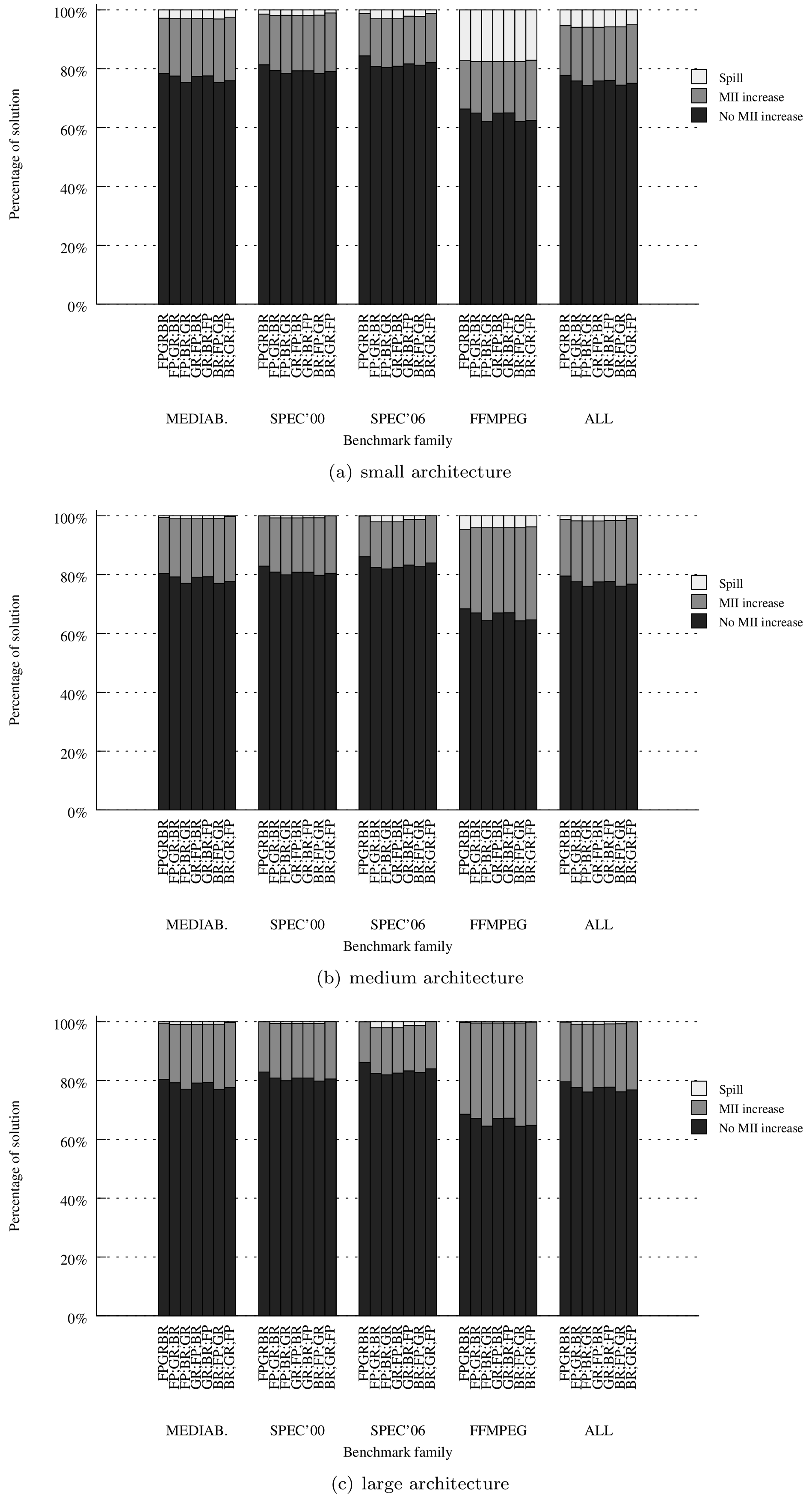
The previous section shows that most of the DDGs do not end up with an increase in MII. Still we need to quantify the overall MII increase. Figure 2.2 shows the increase of the MII when SIRALINA found a solution for a DDG. The figure plots the results for for each register optimisation order and for each benchmark family. This increase is computed in overall on all DDGs by the formula $\frac{\sum \mathrm{MII}(G')}{\sum \mathrm{MII}(G)} - 1$, where $MII(G')$ is the new critical circuit constructed after applying SIRALINA on the DDG G, while MII(G) is it initial value.

We observe that the global increase of the MII is relatively low (about 15% in the worst case). More precisely, the increase is negligible for MEDIABENCH, SPEC2000 and SPEC2006 benchmarks whereas it cannot be neglected on FFMPEG benchmarks. The reason is that the FFMPEG benchmark contains much more complex and difficult DDG instances compared to the other benchmarks.

2.2.3 Efficiency of execution times

Figure 2.3 shows the boxplot[2] of execution times of SIRALINA, depending on the register optimisation order. We measured the execution time taken by SIRALINA to find a solution for a DDG, given an

[2] Boxplot, also known as *box-and-whisker diagram*, is a convenient way of graphically depicting groups of numerical data through their five-number summaries: the smallest observations (min), lower quartile ($Q1 = 25\%$), median ($Q2 = 50\%$), upper quartile ($Q3 = 75\%$), and largest observations (max). The min is the first value of the boxplot, and the max is the last value. Sometimes, the extrema values (min or max) are very close to one of the quartiles. This is why we do not distinguish sometimes between the extrema values and some quartiles.

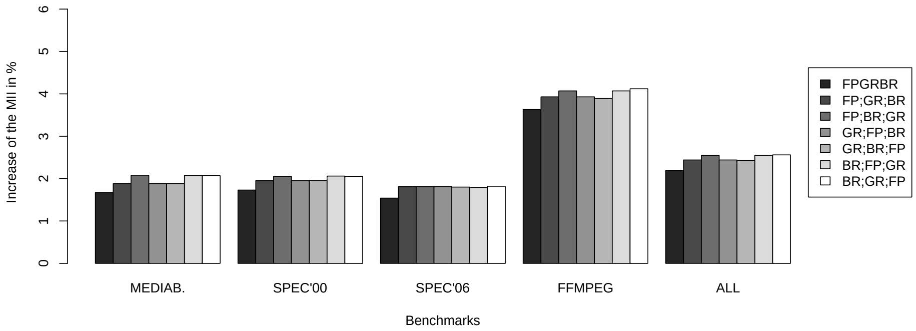


(a) small architecture

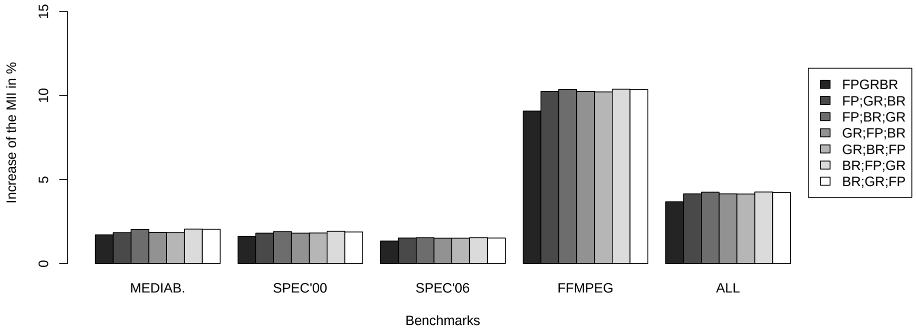
(b) medium architecture

(c) large architecture

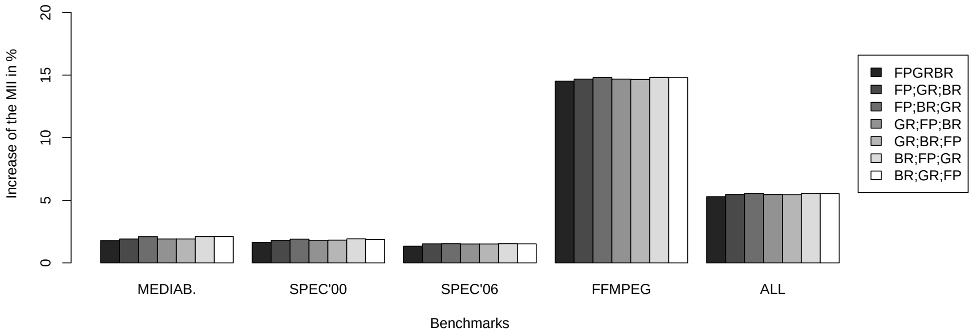
Figure 2.1: Percentage of DDG treated successfully by SIRALINA and the impact on the MII



(a) small architecture



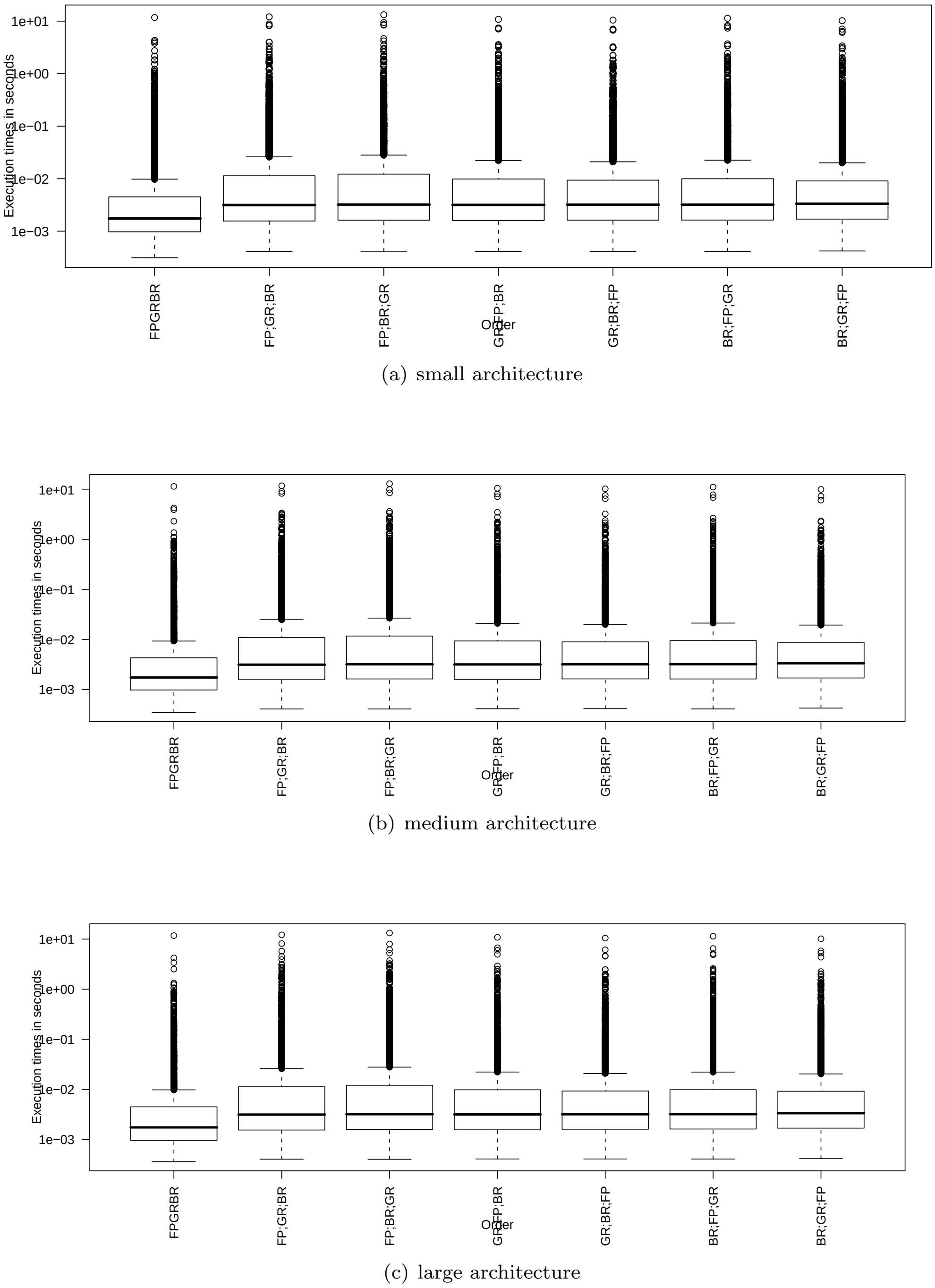
(b) medium architecture



(c) large architecture

Figure 2.2: Average increase of the MII

architectural configuration. All execution times are reported, including the cases where SIRALINA did not find a solution.



(a) small architecture

(b) medium architecture

(c) large architecture

Figure 2.3: Boxplots of all execution times of SIRALINA (all DDG)

We observe that simultaneous SIRALINA register optimisation order outperforms clearly the sequential register optimisation orders, since it is almost twice as fast as these.

A close examination (not clear from Figure 2.3) of the execution times shows also that the speed of sequential register optimisation orders is highly dependent on the order in which register types are treated. This is not surprising since a search for a solution (iterating on II) may continue unnecessarily if a subset of the constraints can be satisfied but not the entire set. For instance, imagine an (hypothetical) architecture with GR=FP=∞ and BR=0, then depending on the order in which types are treated, a sequential search procedure will fail more or less quickly on any DDG that has at least one BR value.

2.3 Conclusion about SIRALINA efficiency

We have presented experimental results of SIRALINA heuristic on standalone DDG extracted from various benchmarks. These experiments demonstrate that SIRALINA succeeds in reducing significantly the register pressure of innermost loops (thus avoiding the generation of spill code). We have also observed that the increase of the MII remains null in most of the cases. For the cases where we observe an increase in the value of MII, we observe that it remains quite low in most of the cases (less than 3% in average for SPEC2000, SPEC2006 and MEDIABENCH), but cannot be neglected in the worst cases (till 15% of MII increase in worst case for FFMPEG). Finally, we have noted that it is usually better to optimise all register types conjointly instead of one by one: not only obtained results in term of register pressure minimisation are better but execution times are also much faster.

Chapter 3

Reducing the register saturation of directed acyclic graphs

We report in this chapter experimental results of a SIRALINA-based heuristic that aims at reducing the register saturation (RS) of directed acyclic DDGs (basic blocks, super-blocks).

Contrary to the previous chapter, the SIRA framework does not model exactly the register requirement in a DAG, but a conservative estimation of it. That is, $\sum \mu$ computed for a cyclic DDG becomes an over-estimation of MAXLIVE in the acyclic case (loop body). Thus, we can safely use SIRALINA to build a heuristic for reducing the register saturation of the DAG.

3.1 A heuristic for the reducing the acyclic register saturation

SIRALINA heuristic can be used to reduce the register saturation (RS) [10] of DAGs. In [10], we have already designed a heuristic for reducing the RS, but its performance was not really satisfactory. This section provides a heuristic using SIRALINA, shown really efficient in the previous chapter, that we apply in the acyclic context.

A DAG $G = (V, E, \delta)$ is simply an acyclic DDG (V, E, δ, λ) where λ is the zero function (i.e. $\lambda(e) = 0$ for any $e \in E$).

To ease writing of mathematical definitions and results, we assume that (V, E) has a unique source $\top$ and a unique sink $\perp$ (otherwise, we apply a classic transformation to it).

We now describe a heuristic to reduce the register saturation of G. This heuristic add edges to G (in order to reduce its RS) as follows:

1. **Tranform the DAG to a cyclic DDG:** Compute $\overline{G} = (V, E \cup \{(\perp, \top)\}, \delta, \lambda)$ where $\delta(\perp, \top) = 1$, $\lambda(\perp, \top) = 1$ (i.e. simply add a edge from the sink to the source with positive latency and distance)
2. **Reduce the register pressure in the cyclic DDG:** Apply SIRALINA heuristic to the cyclic DDG $\overline{G}$ (for the desired register types) to obtain the associated DDG $\overline{G'}$. For each $\overline{G}$, SIRALINA is applied with a fixed II set to $L_{\overline{G}}$ a maximal value of II depending on $\overline{G}$. II is set to its maximal value to model the fact that we do not consider any overlap (software pipelining) between the successive iterations of the loop.
3. **Come back from the cyclic DDG to a DAG:** Filter out from $\overline{G'}$ the edges e such that $\lambda(e) \neq 0$ in order to get G'. Since only edges with $\lambda(e) = 0$ are kept, the resulting DDG remains a DAG[1].
4. Return G'.

Note that, by definition, register saturation of G' is at most equal to register saturation of G. This is because G' is an extension of G. We report in the following experimental results of this heuristic.

[1] Indeed, this assertion is not correct if no special care is taken. Since our processor model admits delayed access times to registers, then the edges inserted to the DDG by SIRALINA may be non-positive. Consequently, we have to ensure that the resulting DDG after applying SIRALINA remains lexico-graphic positive. This has been correctly done in the experiments of this report. Ensuring that a DDG remains lexico-graphic positive after applying SIRALINA is a complex problem studied in a separate research report.

3.2 Measured data during the experiments

The experimental setup, input data and running hardware have been described in Section 2.1. The DAG used in our experiments are the bodies of the loops after super-block formation. Furthermore, in order to experiment huge and more complex DAGs, we apply loop unrolling with factors 4 and 8. Consequently, the sizes of the DDG increases with a factor of 5 and 9. When applying loop unrolling, the number of experimented DDG becomes equal to 8737 × 3 = 29211 (the set of initial DDG plus the DDG unrolled 4 and 8 times).

Given a DAG G, we note $G'_{\mathcal{O}}$ the resulted DAG after applying the above RS reduction heuristic (based on SIRALINA). $\mathcal{O}$ is the register optimisation order used for applying SIRALINA heuristic (e.g. $\mathcal{O}$=FP;GR or $\mathcal{O}$=BRFPGR or ..., naming conventions of register optimisation orders was explained in Section 2.1.3). For each DAG G, we have measured:

- Its approximated register saturation $RS^t(G)$ for $t \in \mathcal{T}$ thanks to the precise heuristic presented in [3].
- The approximated register saturation $RS^t(G'_{\mathcal{O}})$ for $t \in \mathcal{T}$ and the elapsed time needed to compute $G'_{\mathcal{O}}$.
- The longest path in G and in $G'_{\mathcal{O}}$ (in order to study the increase of the critical path after RS reduction).

The next section provides a synthesis of the collected data.

3.3 Experimental results

3.3.1 Quality of RS reduction heuristic

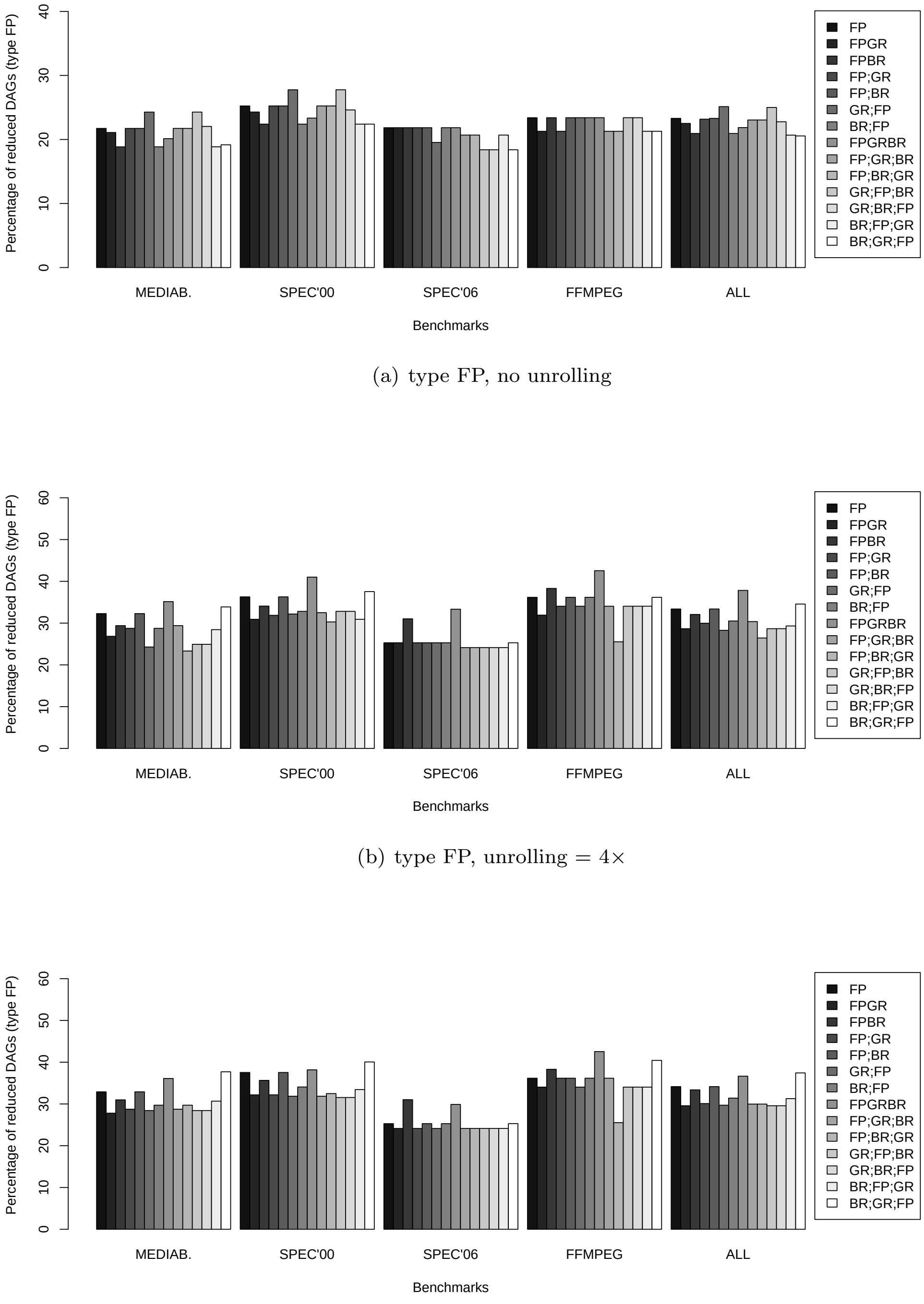
For each register type $t \in \mathcal{T} = \{\mathrm{BR}, \mathrm{GR}, \mathrm{FP}\}$ and for each register optimisation order $\mathcal{O}$ involving type t, we have measured the proportion of DAGs whose register saturation of type t has been reduced. The results are shown in Figure 3.1. We deliberately separate the results of the original DDG with the results of the DDG unrolled 4 and 8 times to have a better idea on the efficiency in function of the DDG size.

We also measured the percentage of reduced registers saturation of type t for each optimisation order $\mathcal{O}$, which is given by the formula $1 - \dfrac{\sum RS^t(G'_{\mathcal{O}})}{\sum RS^t(G)}$. Results are given on Figure 3.2.

We remark on Figure 3.1 that on smaller DAGs (i.e. no unrolling), the order in which register types are optimised does not seem to have a crucial impact on performance of the RS reduction heuristic. On the contrary, on bigger DAGs, the performance of RS reduction heuristic is clearly altered by the register optimisation order. For instance, when observing Figure 3.2(h) and Figure 3.2(i), it is clear that the register types orders that focus on BR before GR obtain better results. Similarly, for type FP, we see on Figure 3.2(b) and Figure 3.2(c) that some orders are better than others (e.g. FPGRBR clearly gives better results).

When looking at the percentage of reduced RS on Figure 3.2, we note that the RS reduction heuristic gives better results on intermediate sized DAGs (unrolling = 4×) than on small DAGs or really big DAGs. This confirms also that the register optimisation order is important. For instance, for type BR, we clearly see that the best orders are those that focus on type BR before focusing on type GR: optimising GR type first seems to add too much constraints to the DAGs and this results in bad BR optimisation.

More generally, we conclude that the RS reduction heuristic based on SIRALINA succeeds in reducing significantly register saturation of DAGs. We also note that simultaneous register optimisation orders, which have the advantage of not being subject to the order of resolution, achieve good results, comparable to the best results obtained by the sequential register optimisation orders.

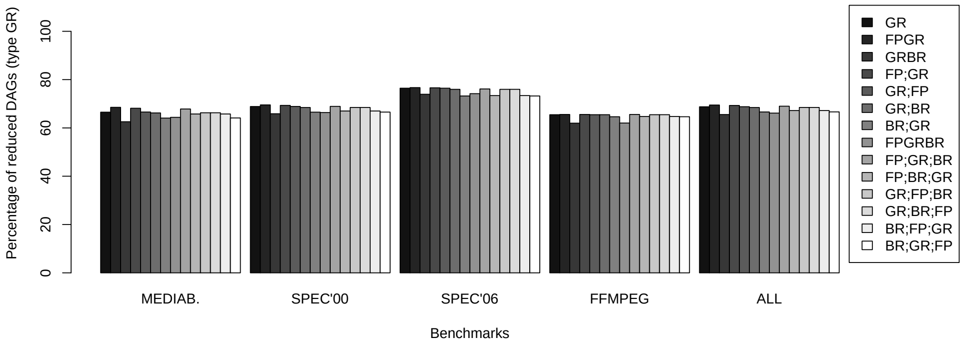


(a) type FP, no unrolling

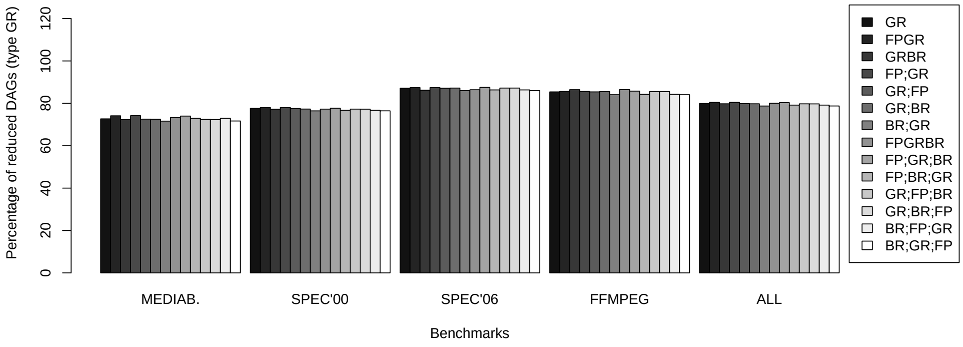
(b) type FP, unrolling = 4×

(c) type FP, unrolling = 8×

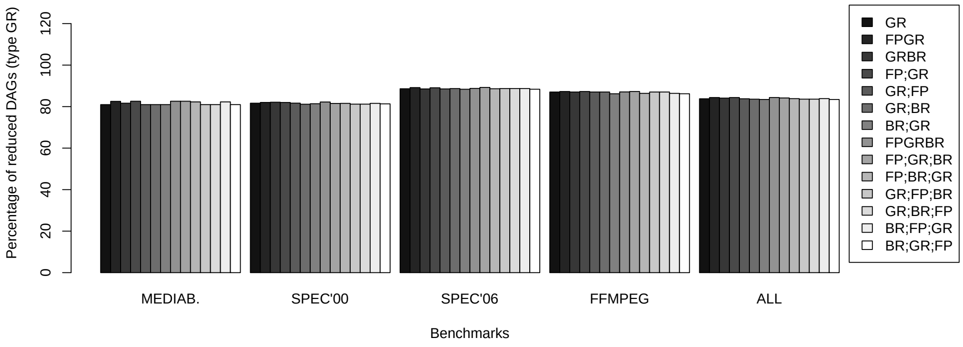
Figure 3.1: Percentage of DAGs with reduced RS



(d) type GR, no unrolling

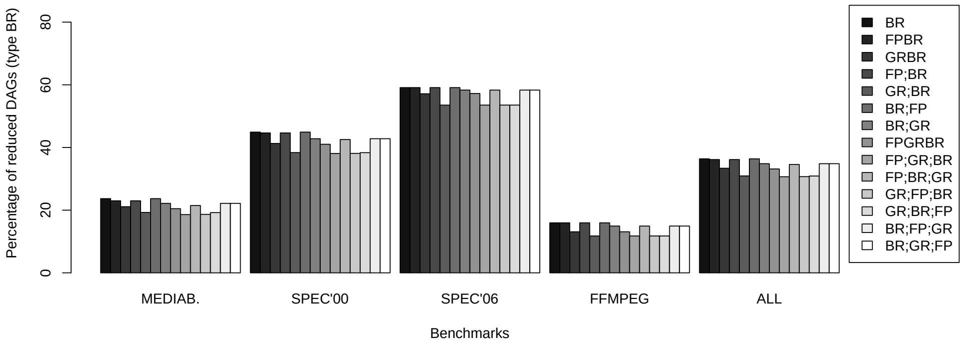


(e) type GR, unrolling = 4×

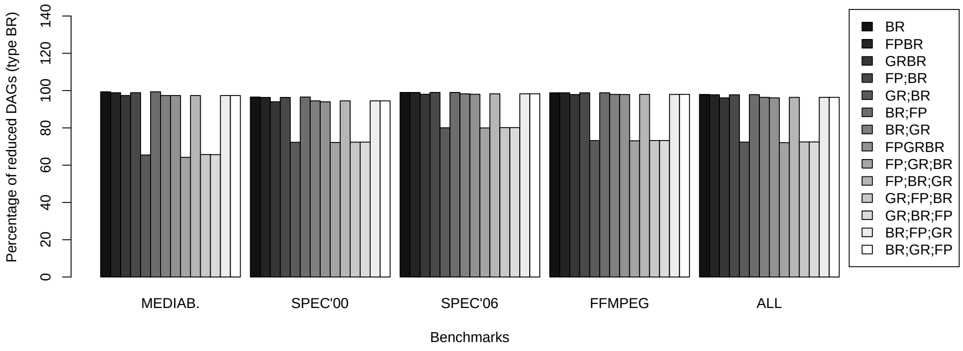


(f) type GR, unrolling = 8×

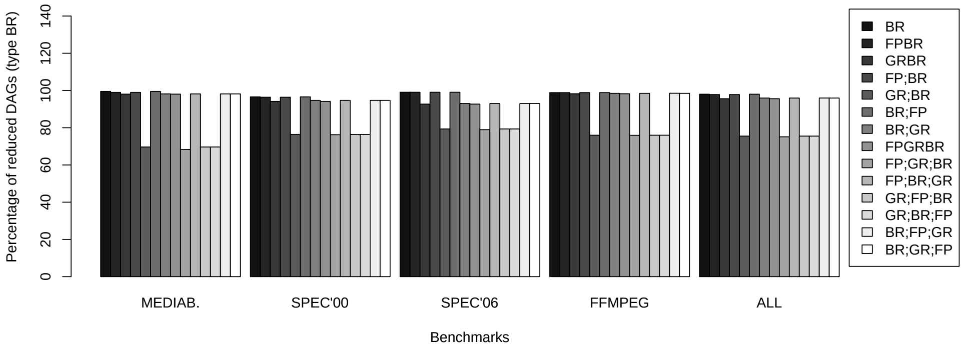
Figure 3.1: Percentage of DAGs with reduced RS (con't)



(g) type BR, no unrolling

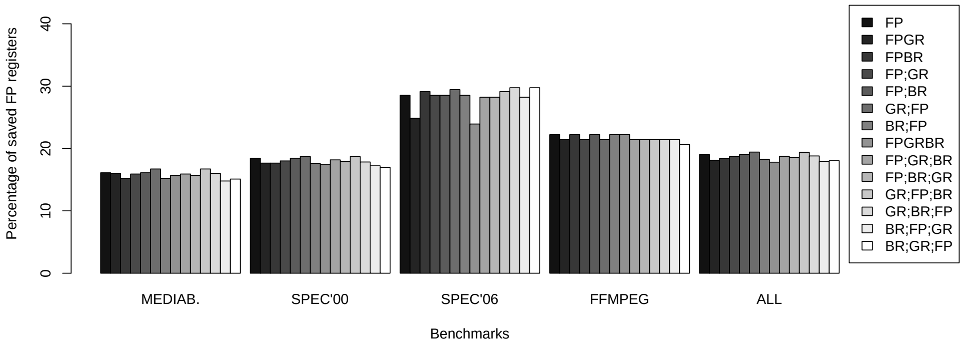


(h) type BR, unrolling = 4×

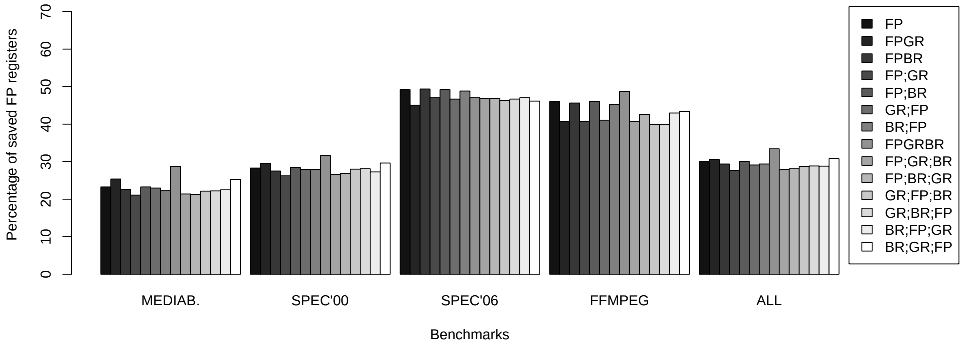


(i) type BR, unrolling = 8×

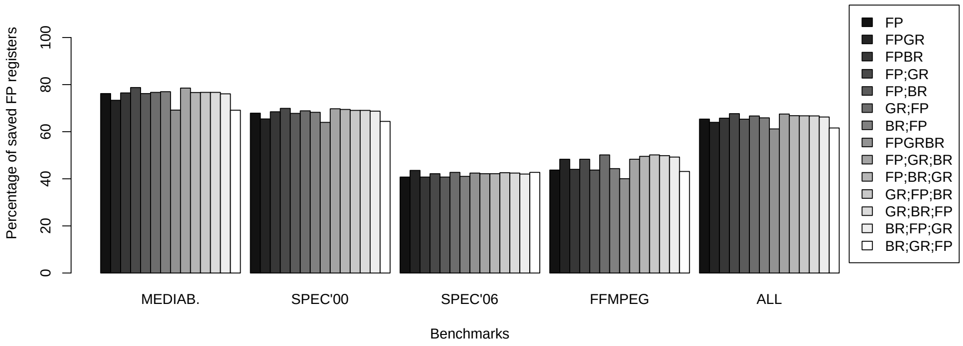
Figure 3.1: Percentage of DAGs with reduced RS (con't)



(a) type FP, no unrolling

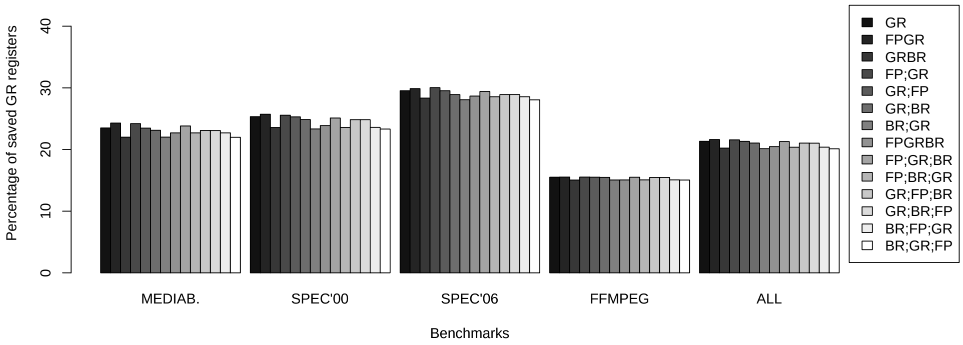


(b) type FP, unrolling = 4×

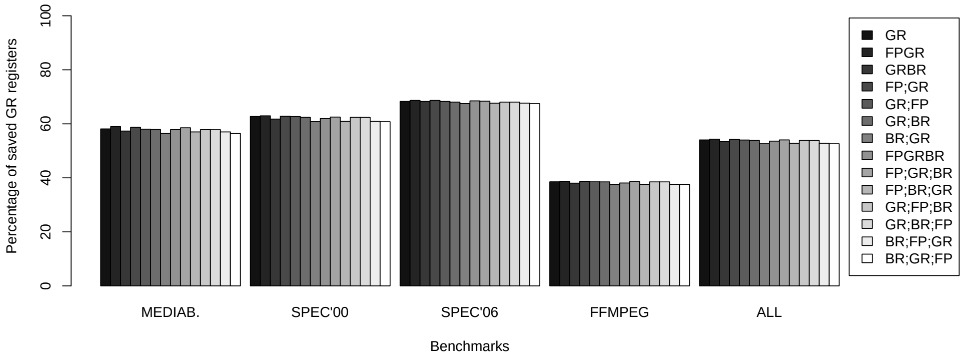


(c) type FP, unrolling = 8×

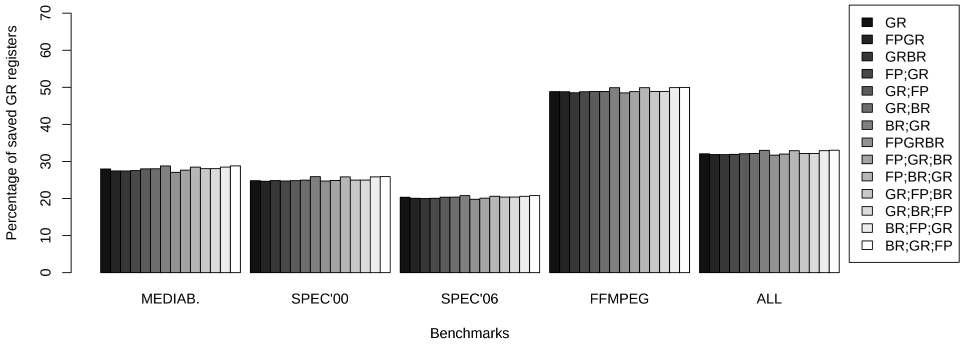
Figure 3.2: Percentage of reduced register saturation



(d) type GR, no unrolling

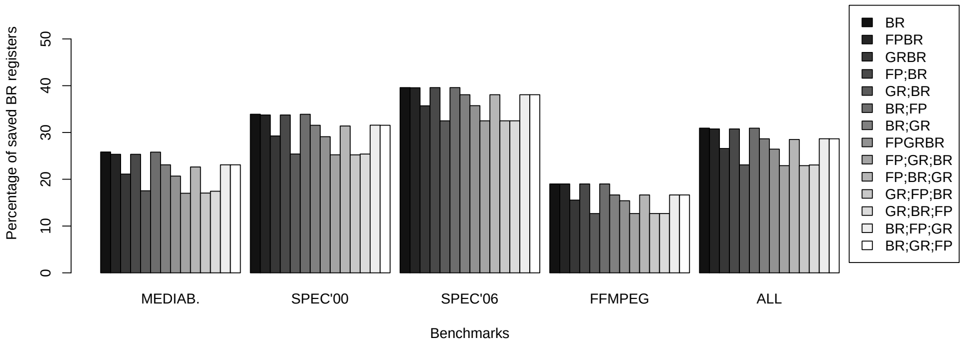


(e) type GR, unrolling = 4×

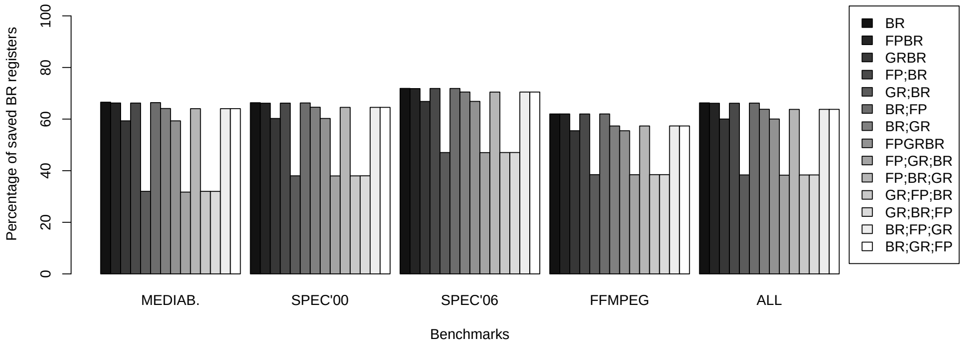


(f) type GR, unrolling = 8×

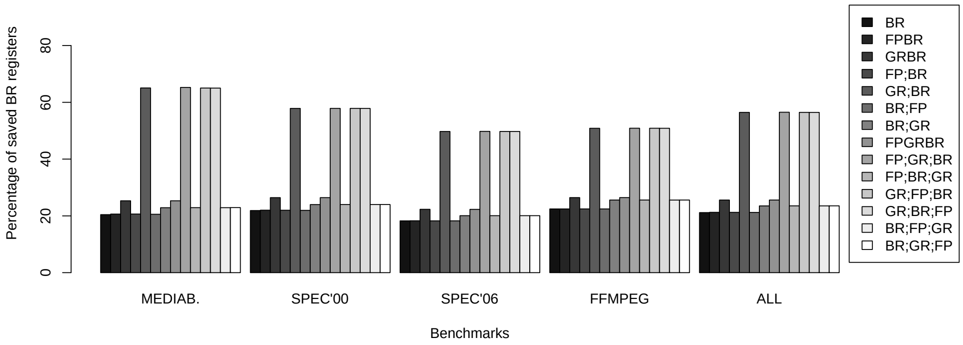
Figure 3.2: Percentage of reduced register saturation (con't)



(g) type BR, no unrolling



(h) type BR, unrolling = 4×

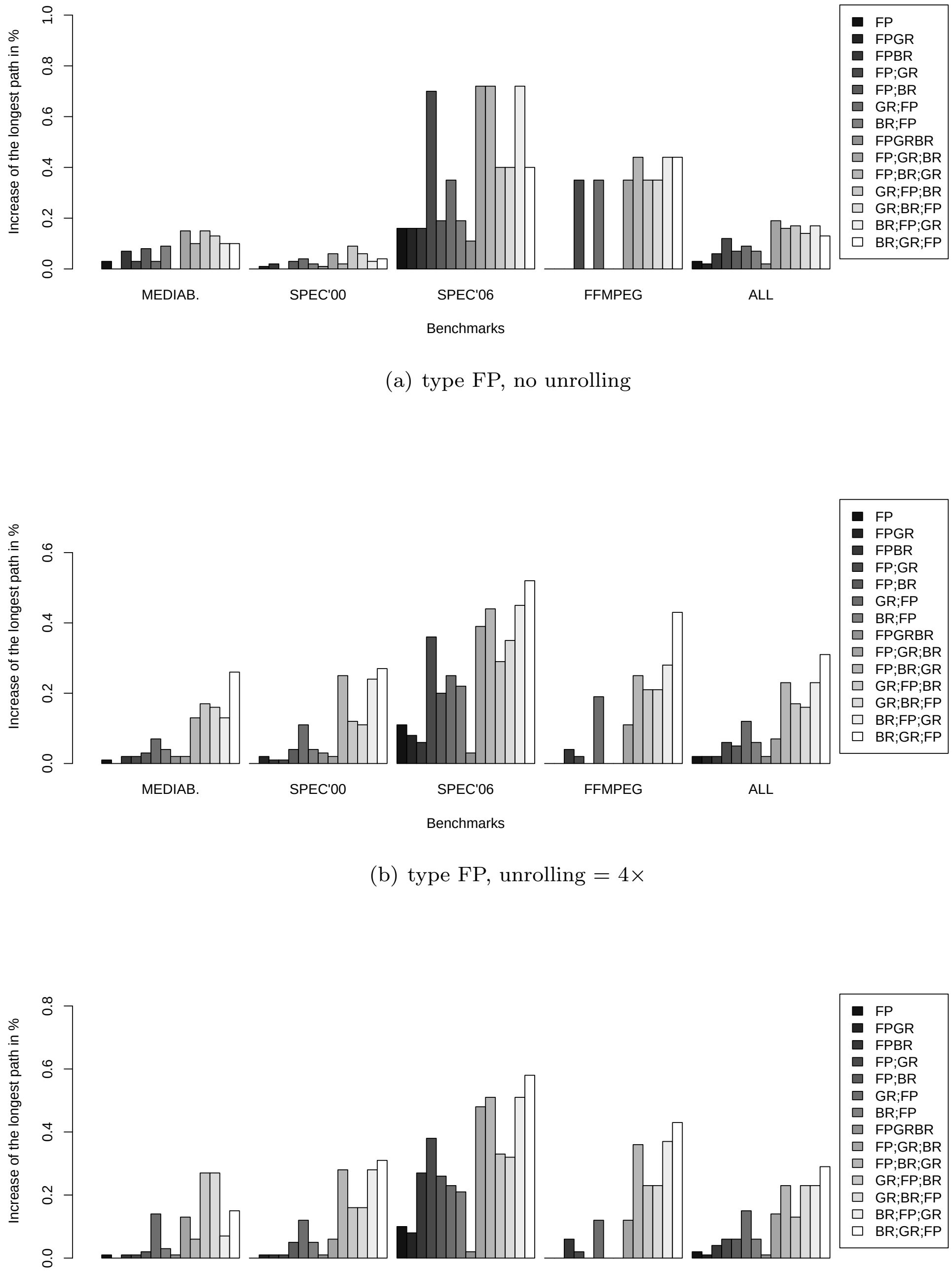


(i) type BR, unrolling = 8×

Figure 3.2: Percentage of reduced register saturation (con't)

3.3.2 Impact of the register saturation reduction on the longest paths

We have measured the increase of the longest paths due to the register saturation reduction heuristic on all the DDG after applying RS reduction. The increase is expressed by the formula $\frac{\sum lp(G'_{\mathcal{O}})}{\sum lp(G)} - 1$. The experimental results are reported on Figure 3.3.

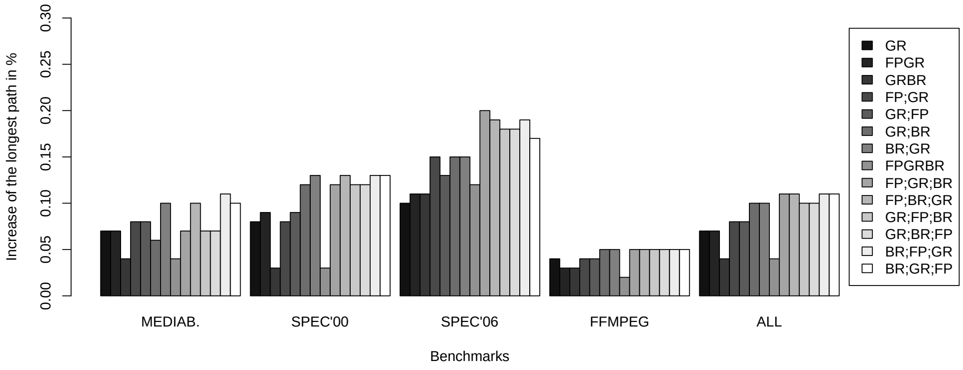


(a) type FP, no unrolling

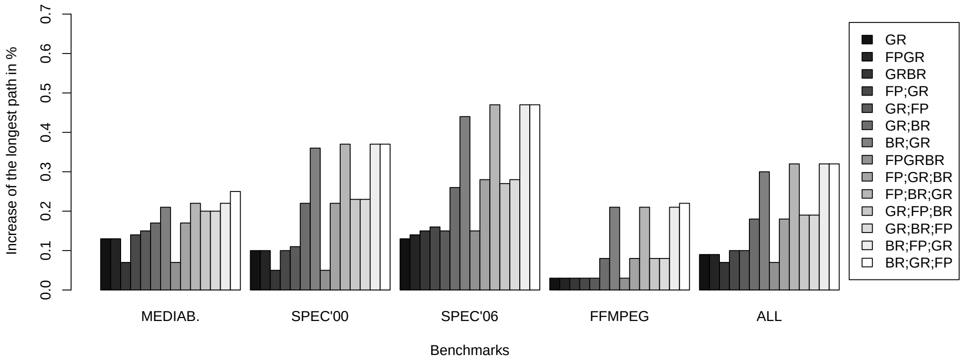
(b) type FP, unrolling = 4×

(c) type FP, unrolling = 8×

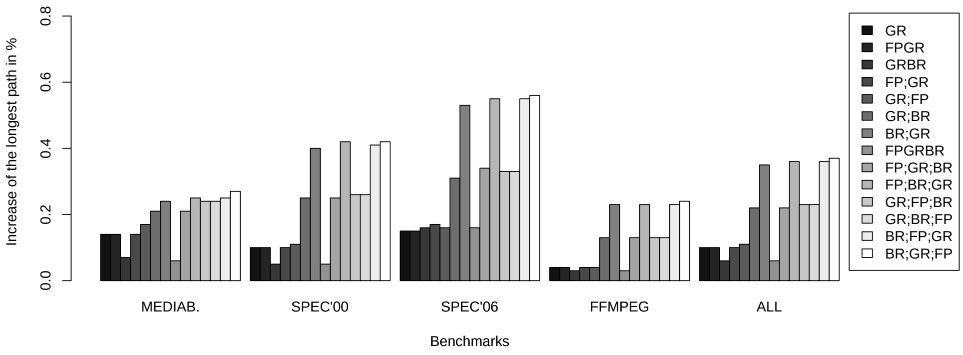
Figure 3.3: Overall increase of the longest path



(d) type GR, no unrolling

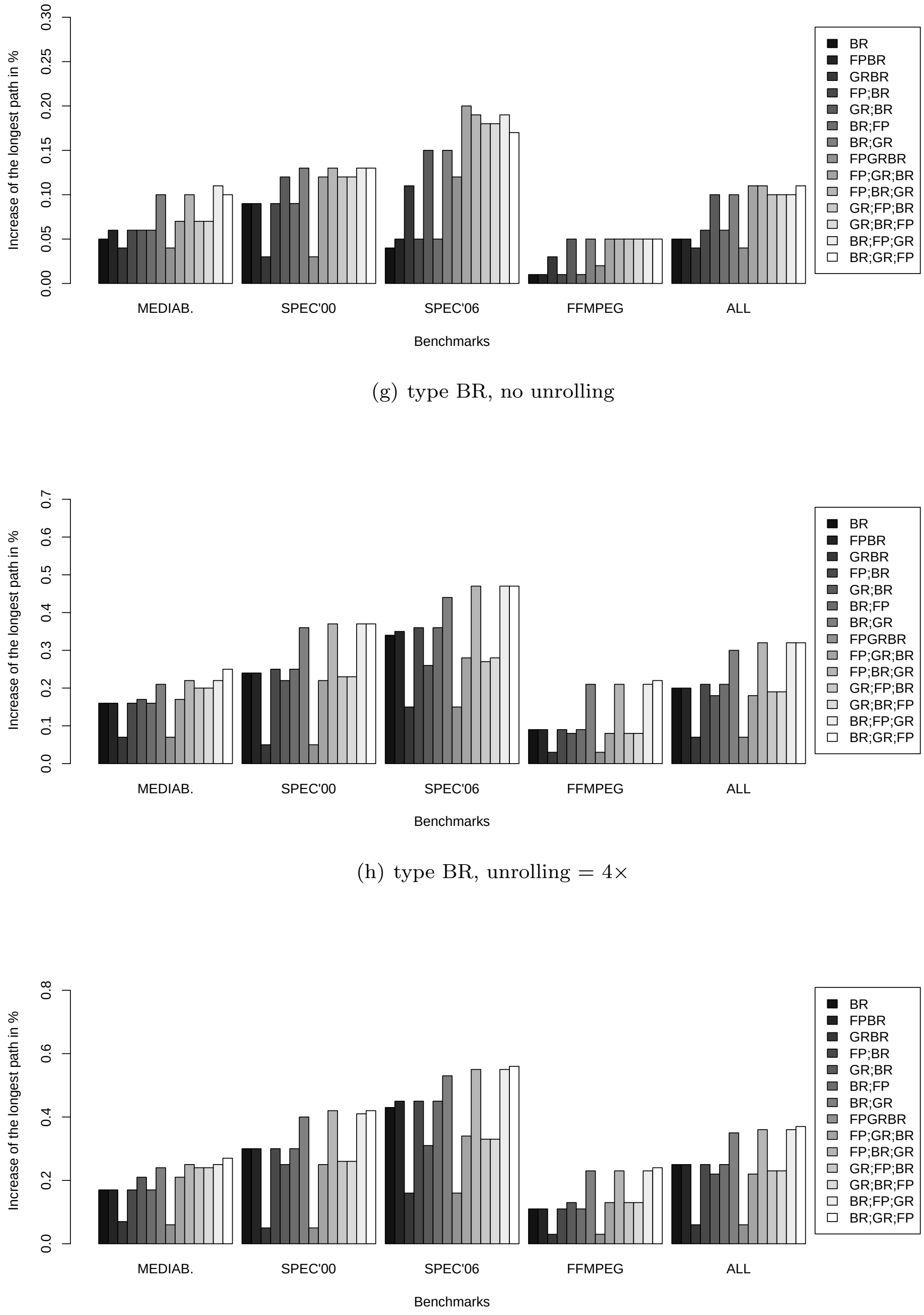


(e) type GR, unrolling = 4×



(f) type GR, unrolling = 8×

Figure 3.3: Overall increase of the longest path (con't)



(g) type BR, no unrolling

(h) type BR, unrolling = 4×

(i) type BR, unrolling = 8×

Figure 3.3: Overall increase of the longest paths (con't)

Clearly, the impact of the register saturation reduction on the longest paths is negligible, since its average does not exceed 0.6% in the worst cases. We even observe that the increase of the longest path seems to diminish as the size of the DAGs augments. This remarks has already been reported in [10]. The explanation is that bigger DDG have more values, yielding to more opportunities to reduce the RS.

3.3.3 Analysis of the heuristic execution times

We report the execution times of the RS reduction heuristic on the whole set of benchmarks (ALL), depending on the order in which register types are treated by SIRALINA. The results are given in Figure 3.4 using boxplots.

We note that, as expected, the register optimisation order involving only one type are faster than register optimisation order involving two types, which themselves are faster than register optimisation orders involving three types. Moreover, we clearly see that the cost of a register type is directly related to the number of its values. This is not surprising because the size of the scheduling problem depends on $|V^{R,t}|$. We thus see that solving type GR is more costly than solving either of the two other types.

We also see that, for a given set of types, simultaneous register optimisation orders are faster than sequential ones. Once again, this was expected since the sequential register optimisation orders require to solve several times the same constraints (the scheduling problem in the step 1 of SIRALINA). Note in particular the gap between register optimisation order FPGRBR and all the other register optimisation orders which deals with the three types. It even seems to be faster than some of the sequential register optimisation orders dealing with only two types.

3.3.4 Efficiency of the heuristic for a fixed architectural configuration

In the sequel, we analyse the efficiency of the register saturation reduction heuristic with three hardware configurations. We have modelled three possible target processor architectures by varying the number of available registers. We analyse the proportion of DAGs that fit into the processor architecture after the RS reduction but did not fit before.

The three processor configurations have the following register constraints (R_t is the number of available registers of type t).

Name of the architecture	R_{FP}	R_{GR}	R_{BR}
Small architecture	32	32	4
Medium architecture	64	64	8
Large architecture	128	128	8

For each unrolling degree, and for each hardware configuration, Figure 3.5 shows the number of DAGs that initially satisfy the register constraints ($\forall t \in \mathcal{T}, RS^t(G) \leq R_t$ the RS of each register type is below the number of available registers).

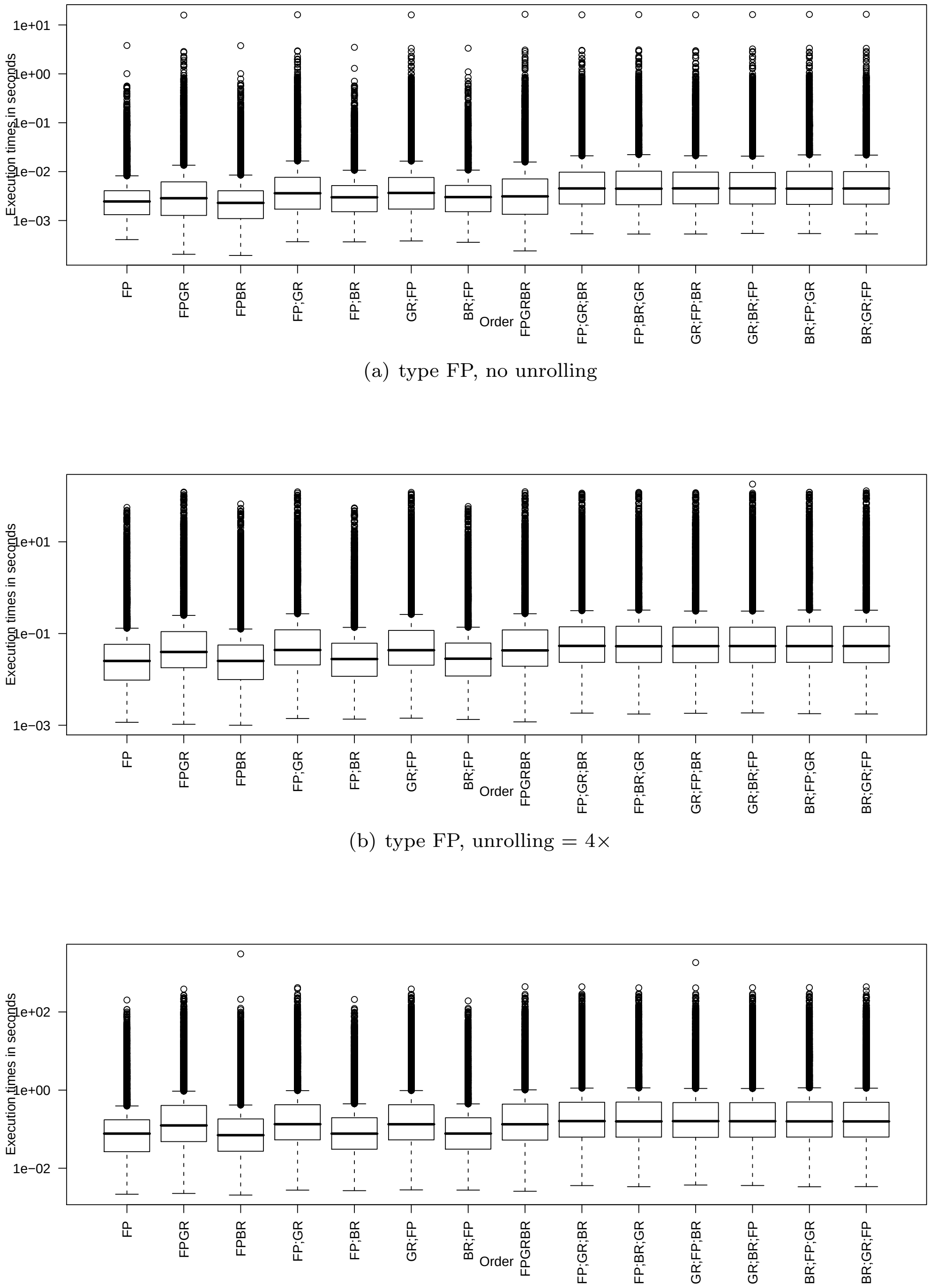
Figure 3.6 shows, among the DAGs that do not fit initially in the architecture, the number of DAGs that violate the register constraints of the considered type. Note that a DAG may be counted two times: for instance if a DAG needs too many BR registers and too many GR registers, it will be counted twice.

We say that a DAG is *recovered* if $\exists t \in \mathcal{T}$ such that $RS^t(G) > R_t$, while $\forall t \in \mathcal{T}, RS_t(G') \leq R_t$. That is, a DAG is recovered if its initial RS exceeds the number of available registers for at least one register type, while its modification (extension) by the RS reduction heuristic reduces its RS below the limit for all the register types. Figure 3.7 shows the percentage of recovered DAGs for each register types order and for each architectural configuration.

Comments

On Figure 3.5, we see that most of the original loop bodies fit on the small architecture (and thus also on the medium and large architecture). We also see most of bodies of loops unrolled four times do not fit on the small architecture whereas they fit on the medium or large architecture. At last, we see that most of bodies of loops unrolled eight times do not fit in any of the architectures.

We observe on Figure 3.6 that the BR type constraints are often violated. However, for FFMPEG benchmarks, GR type constraint is rather the discriminating criteria: this is not surprising because these benchmarks use very few BR values as seen in Section 2.1.

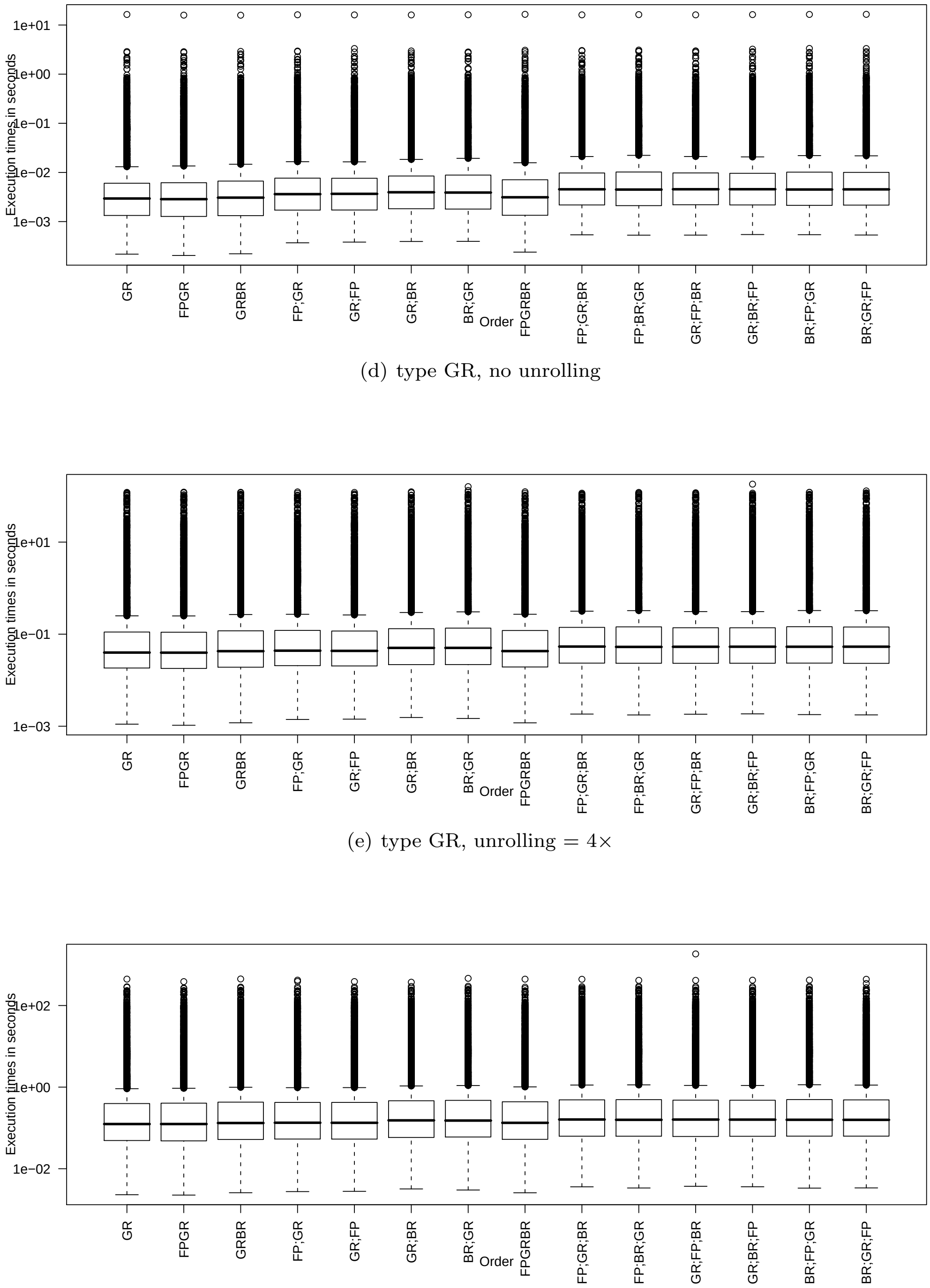


(a) type FP, no unrolling

(b) type FP, unrolling = 4×

(c) type FP, unrolling = 8×

Figure 3.4: Execution times of the RS reduction heuristic

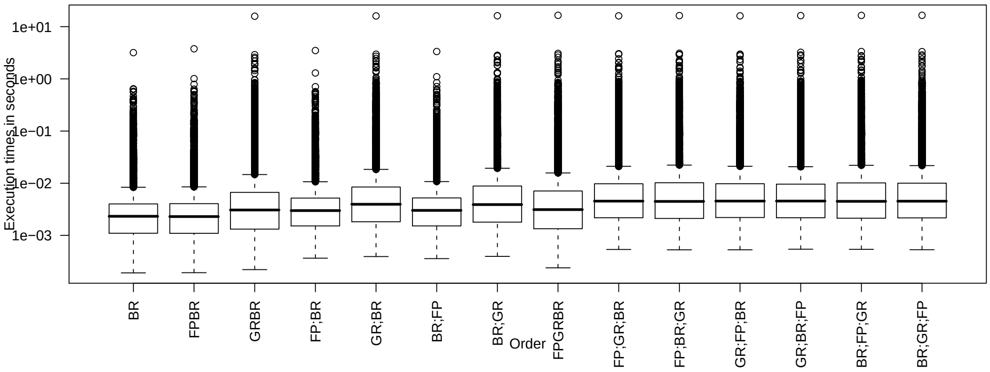


(d) type GR, no unrolling

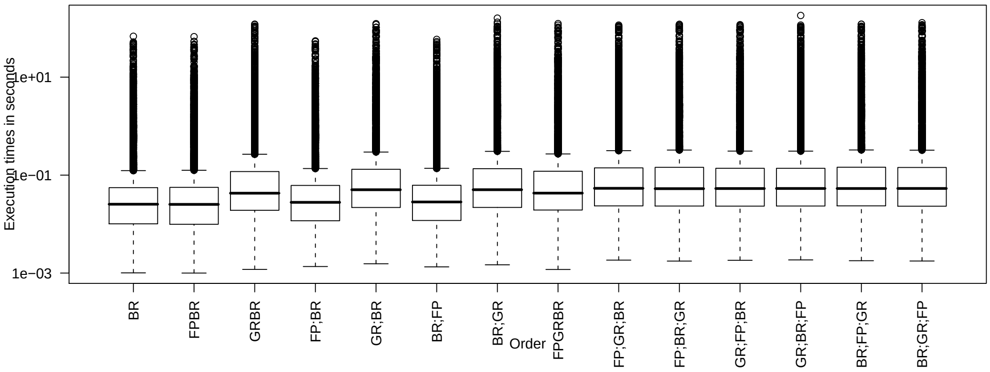
(e) type GR, unrolling = 4×

(f) type GR, unrolling = 8×

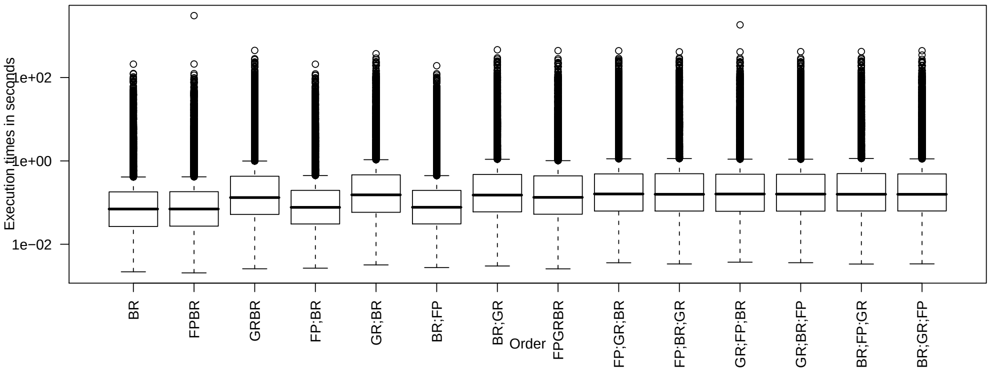
Figure 3.4: Execution times of the RS reduction heuristic (con't)



(g) type BR, no unrolling

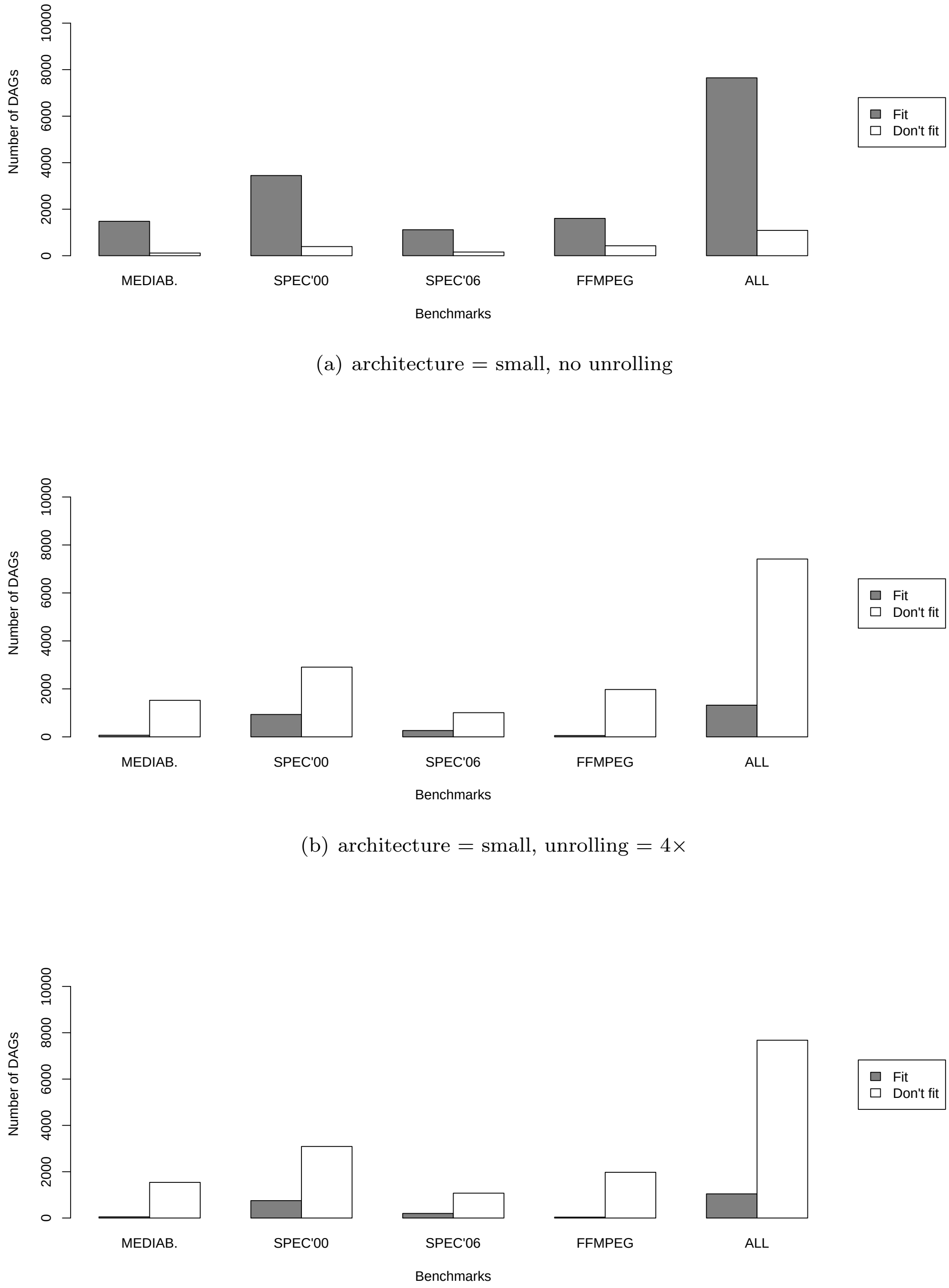


(h) type BR, unrolling = 4×



(i) type BR, unrolling = 8×

Figure 3.4: Execution times of the RS reduction heuristic (con’t)

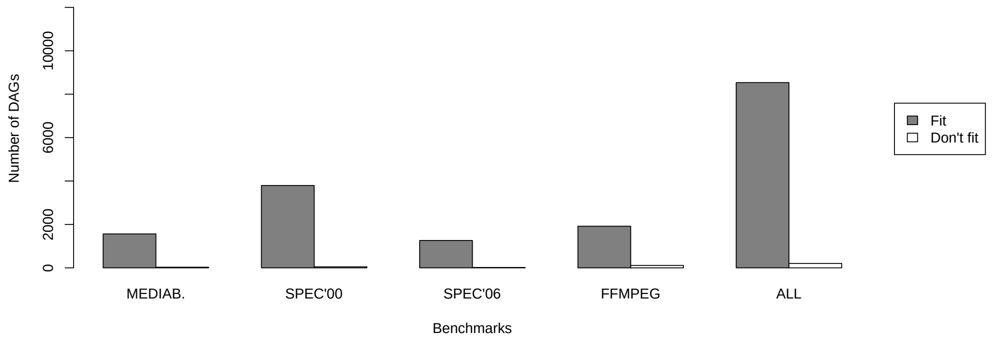


(a) architecture = small, no unrolling

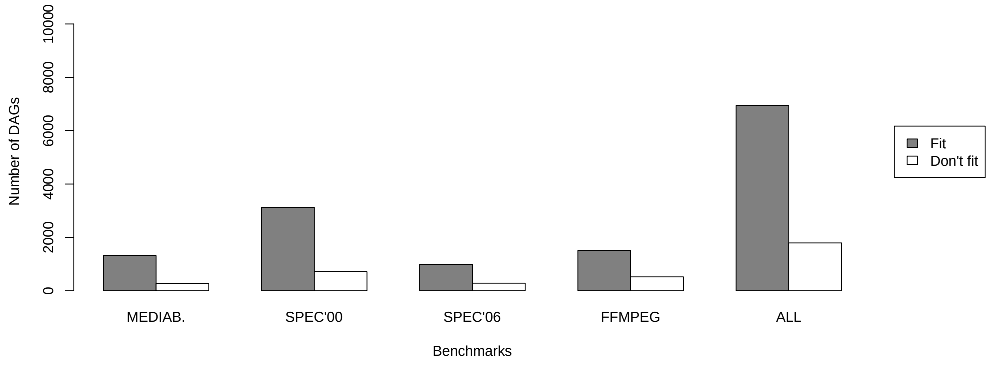
(b) architecture = small, unrolling = 4×

(c) architecture = small, unrolling = 8×

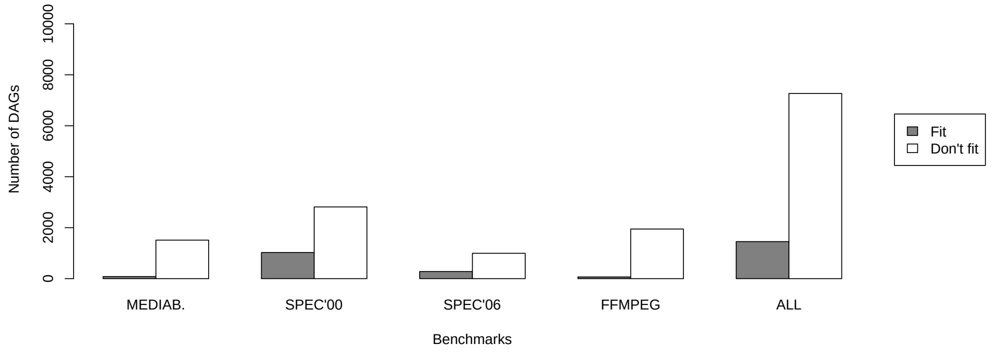
Figure 3.5: Number of DAGs that satisfy the register constraints



(d) architecture = medium, no unrolling

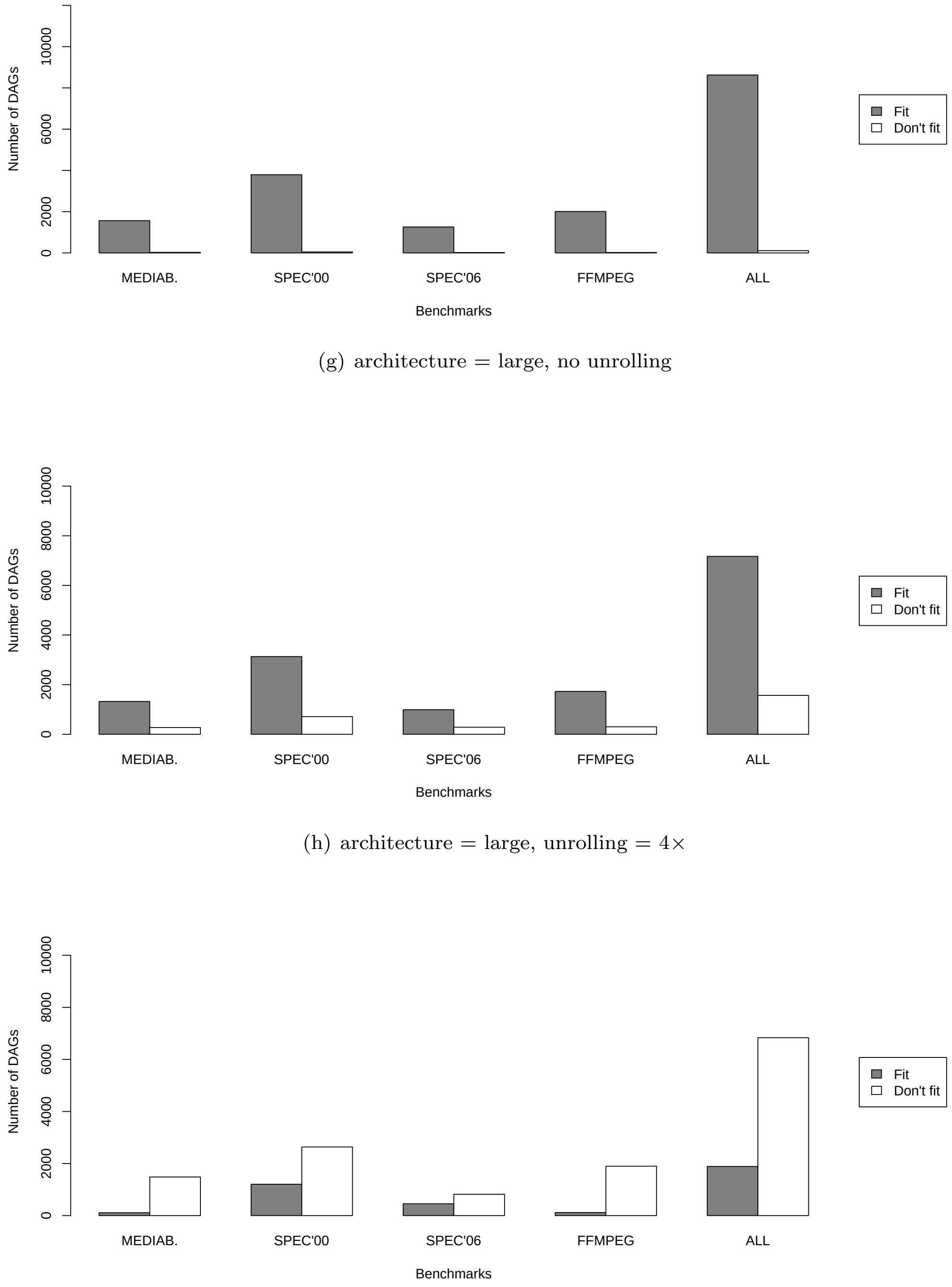


(e) architecture = medium, unrolling = 4×



(f) architecture = medium, unrolling = 8×

Figure 3.5: Number of DAGs that satisfy the register constraints (con't)

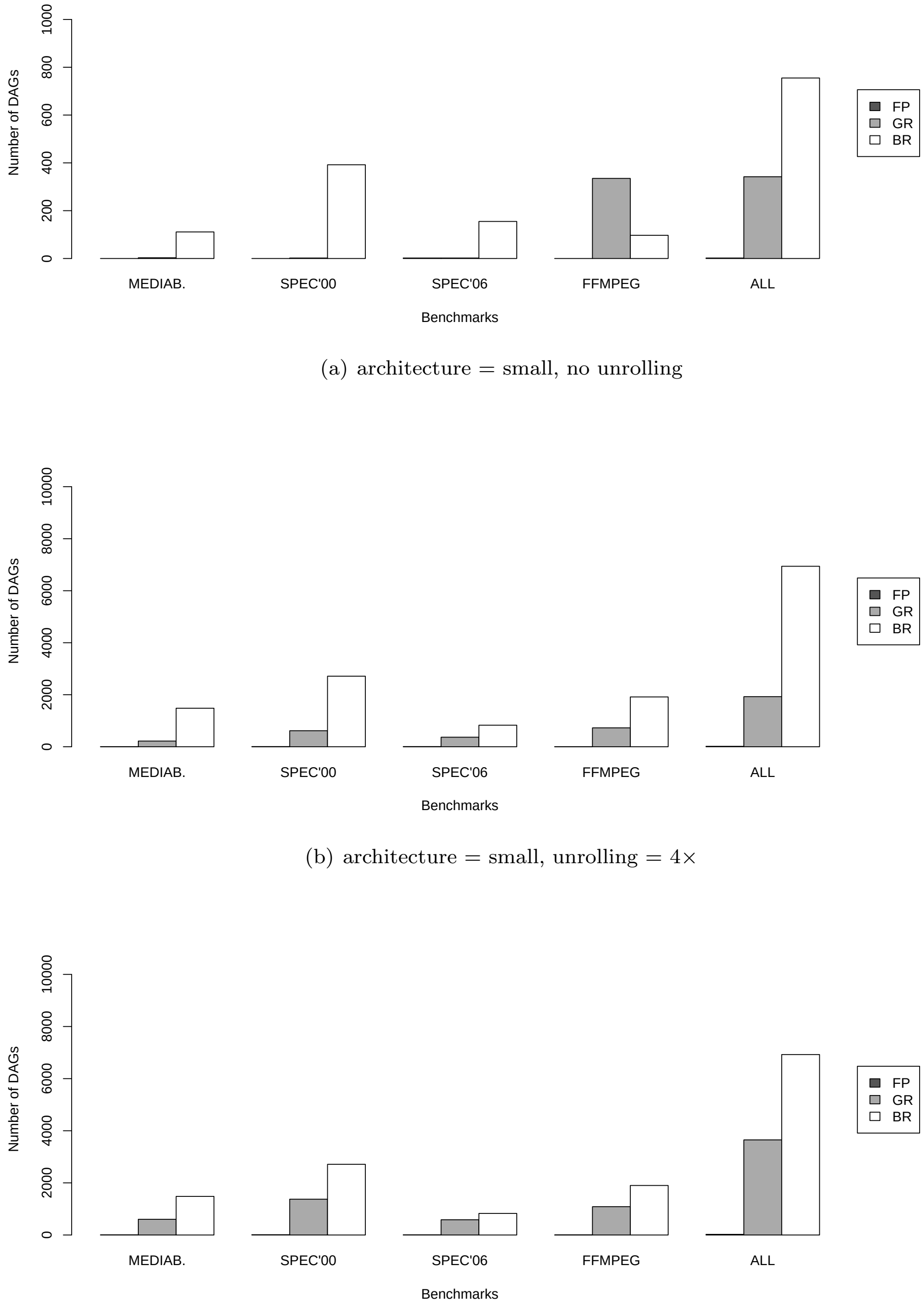


(g) architecture = large, no unrolling

(h) architecture = large, unrolling = 4×

(i) architecture = large, unrolling = 8×

Figure 3.5: Number of DAGs that satisfy the register constraints (con't)

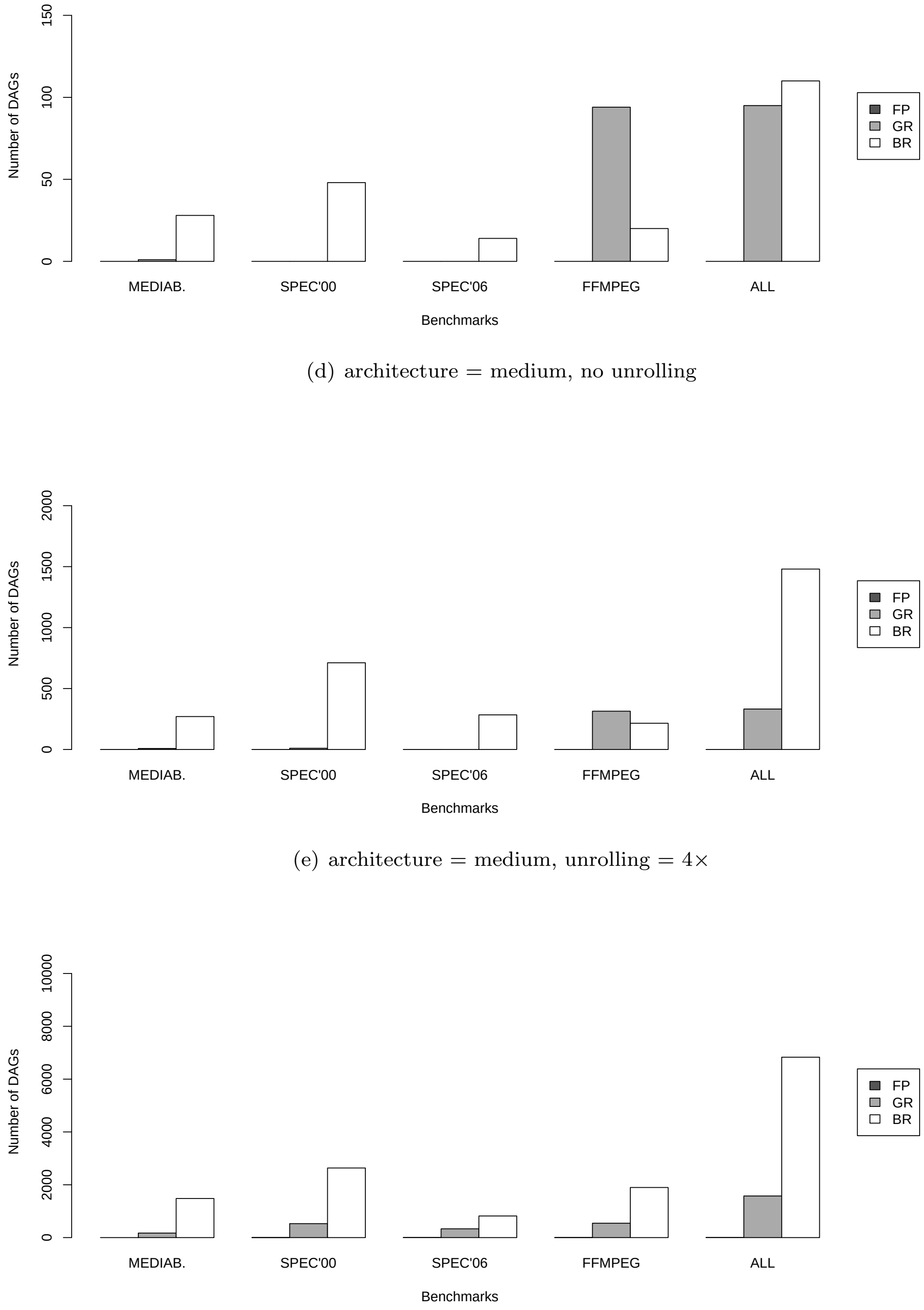


(a) architecture = small, no unrolling

(b) architecture = small, unrolling = 4×

(c) architecture = small, unrolling = 8×

Figure 3.6: Number of DAGs that violate register constraints per type

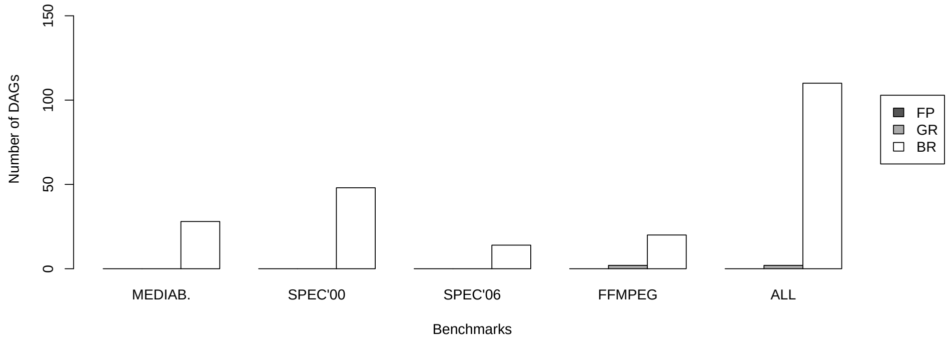


(d) architecture = medium, no unrolling

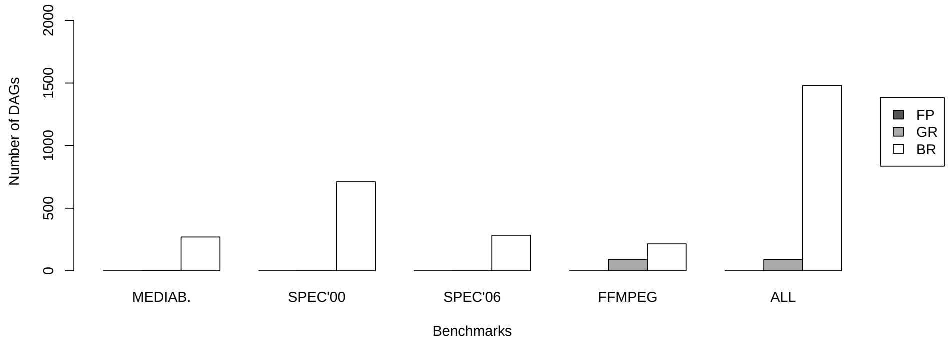
(e) architecture = medium, unrolling = 4×

(f) architecture = medium, unrolling = 8×

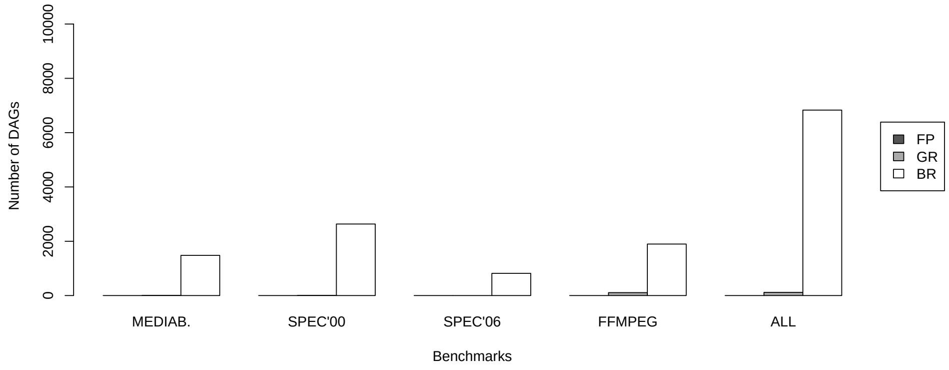
Figure 3.6: Number of DAGs that violate register constraints per type(con't)



(g) architecture = large, no unrolling

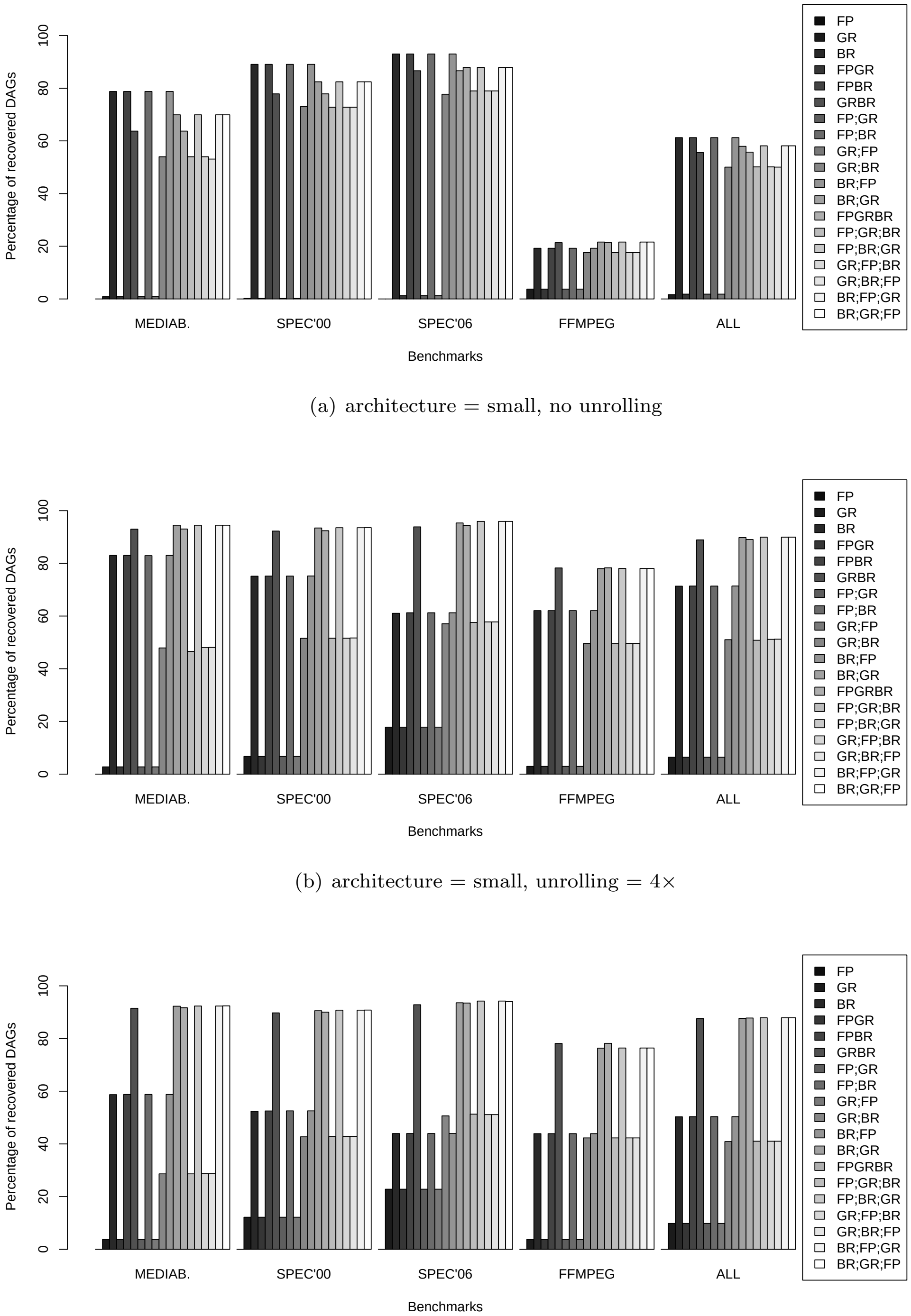


(h) architecture = large, unrolling = 4×



(i) architecture = large, unrolling = 8×

Figure 3.6: Number of DAGs that violate register constraints per type (con't)

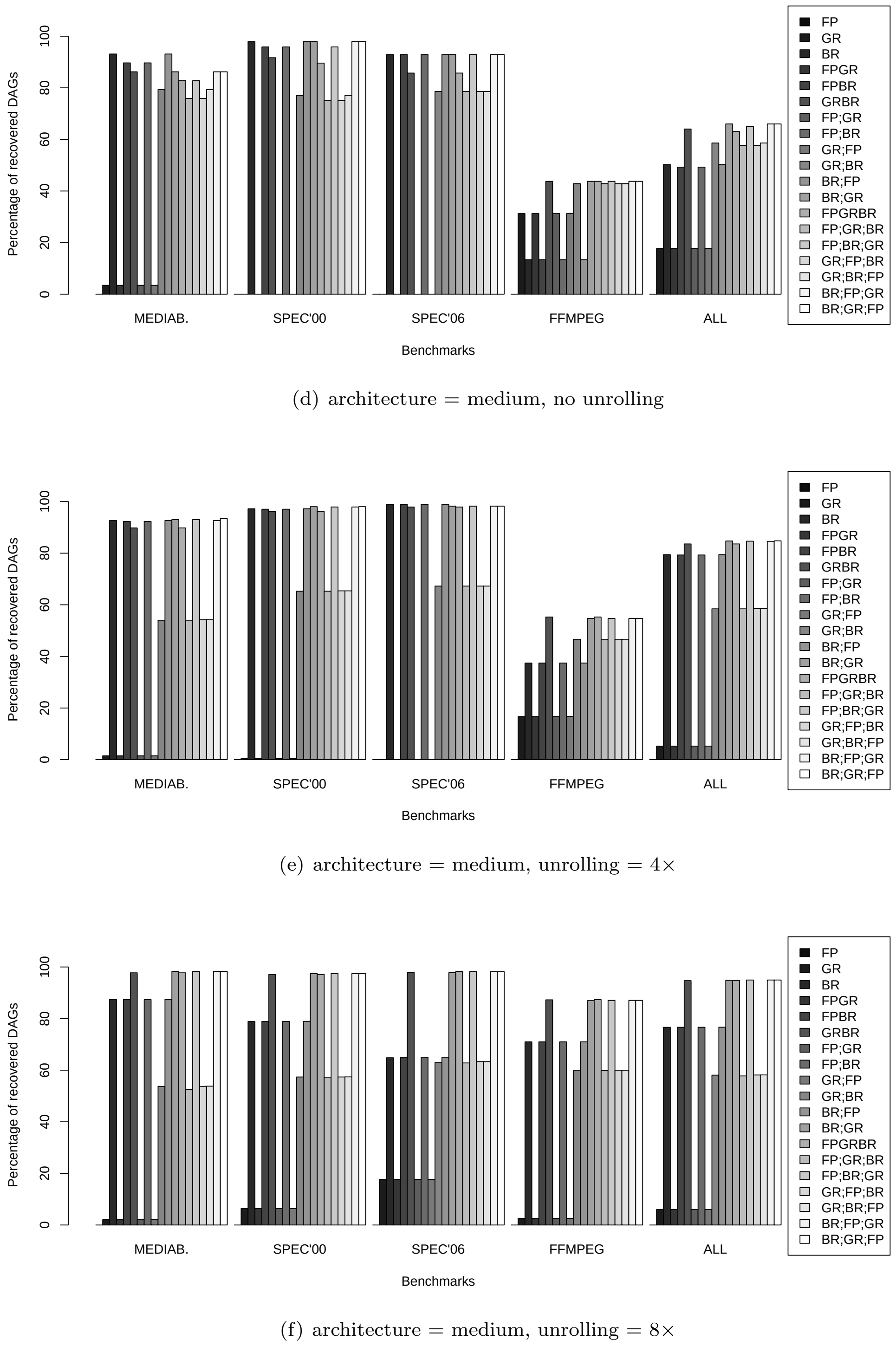


(a) architecture = small, no unrolling

(b) architecture = small, unrolling = 4×

(c) architecture = small, unrolling = 8×

Figure 3.7: Percentage of recovered DAGs

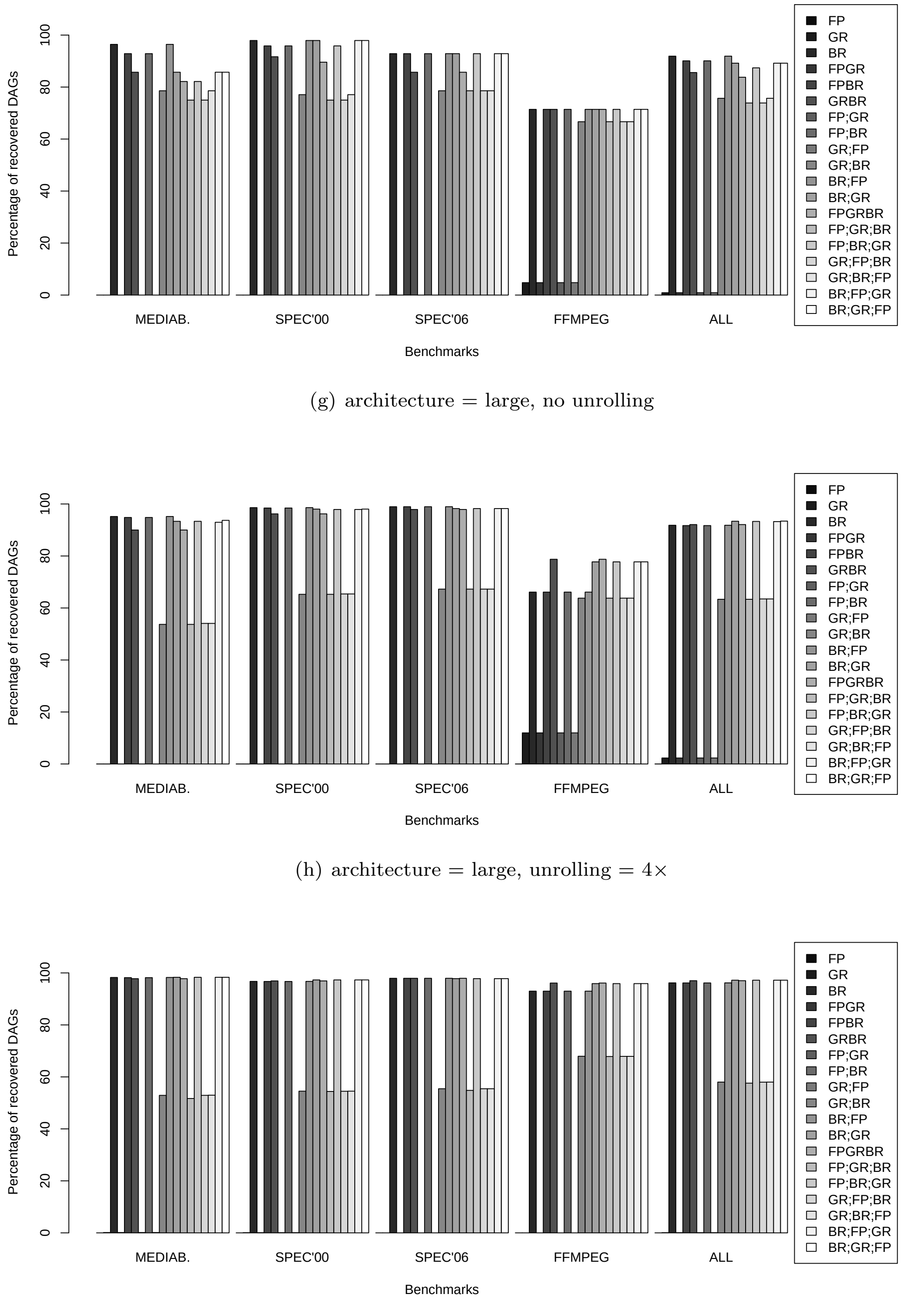


(d) architecture = medium, no unrolling

(e) architecture = medium, unrolling = 4×

(f) architecture = medium, unrolling = 8×

Figure 3.7: Percentage of recovered DAGs (con't)



(g) architecture = large, no unrolling

(h) architecture = large, unrolling = 4×

(i) architecture = large, unrolling = 8×

Figure 3.7: Percentage of recovered DAGs (con't)

We see on Figure 3.7 that the RS reduction heuristic gives satisfactory results, since the percentage of recovered DAGs is often above 80%. We remark however that the results depends largely on the chosen register optimisation order. Of course, as expected, orders that do not involve BR type at all give very bad results since most of the DAGs break the BR constraint. We note that, as previously observed, the orders that focus on BR before focusing on GR give better results than the reverse orders. We also remark the simultaneous order FPGRBR give always good results, comparable to the best results obtained by sequential orders.

3.4 Conclusion

We have presented a heuristic to reduce the register saturation of a DAG. The experimental results show that this heuristic succeeds in reducing significantly register saturation in practice. Its impact on the longest path (and thus on the scheduling) is negligible. The generalised SIRALINA heuristic that deals with conjoint register types at once offer a good compromise between performance (execution time) and RS reduction ratio. so hence we advise to use this conjoint register optimisation method instead of a sequential one, especially when the order on register types cannot be guessed easily.

Chapter 4

A min-cost network flow formulation of SIRALINA

In this chapter, we present a pure algorithmic solution to solve the scheduling problem of SIRALINA heuristic. As explained in Section 1.4, SIRALINA heuristic is composed of two steps:

1. Solve the scheduling problem (step 1).
2. Solve the assignment problem (step 2).

The assignment problem can be solved with e.g. the Hungarian algorithm [6], as previously mentioned. So only the first step has to be converted to an algorithmic solution.

4.1 A min-cost network flow formulation of the scheduling problem

The scheduling problem is an integer linear problem. Due to its particular form, it can be solved efficiently by any linear solver. However, it may be problematic to embed a linear solver in an industrial compiler because of, for instance, engineering or copyright reasons. It is thus interesting to give an algorithmic method to solve the scheduling problem.

In the sequel, we assume that $G = (V, E, \delta, \lambda)$ is a loop DDG and $\mathrm{II} \in \mathbb{N}$. Recall from Chapter 1 that the (generalised) scheduling problem is the following (multiple register types optimised conjointly).

$$\left\{\begin{array}{ll}
\text{minimise} & \displaystyle\sum_{t\in\mathcal{T}} \alpha_t \left(\sum_{u\in V^{R,t}} \sigma_{k_u^t} - \sum_{u\in V^{R,t}} \sigma_u \right) \\
\text{subject to} & \\
& \text{for any } e=(u,v)\in E \\
& \qquad \sigma_v - \sigma_u \geq \delta(e) - \mathrm{II}\times\lambda(e) \\
& \\
& \text{for any } t\in\mathcal{T}\text{, for any } e=(u,v)\in E^{R,t} \\
& \qquad \sigma_{k_u^t} - \sigma_v \geq \delta_{r,t}(v) + \mathrm{II}\times\lambda(e) \\
& \\
& \text{for any } t\in\mathcal{T}\text{, for any } u\in V^{R,t} \text{ such that } Cons^t(u)=\emptyset \\
& \qquad \sigma_{k_u^t} - \sigma_u \geq 1
\end{array}\right.$$

The constraints represent an incidence matrix of a directed graph. This problem contains at most one (+1) and at most one (-1) in each row, representing an arc in the network. It can thus be transformed in a minimum cost flow problem, as explained in [1]. In this particular case, the transformation consists simply in taking the dual problem. For more details on linear programming theory and in particular primal-dual transformation, we refer the reader to [2]. The transformation gives the following problem.

- For each register type $t \in \mathcal{T}$ and for each $u \in V^{R,t}$, let k_u^t be a new vertex (it virtually represents the killer of value u)
- Define the network $N = (V \cup \bigcup_{t \in \mathcal{T}} V_k^t, E \cup \bigcup_{t \in \mathcal{T}} E_k^t)$ where

$$\begin{aligned} V_k^t &= \{k_u^t \mid u \in V^{R,t}\} \\ E_k^t &= \bigcup_{u \in V^{R,t}} \kappa^t(u) \\ \kappa^t(u) &= \begin{cases} \{(v, k_u^t) \mid v \in \mathit{Cons}^t(u)\} & \text{if } \mathit{Cons}^t(u) \neq \emptyset \\ \{(u, k_u^t)\} & \text{otherwise} \end{cases} \end{aligned}$$

  The definition of $\kappa^t(u)$ depends on whether value u of type t has at least a consumer or not. If u has at least one consumer, then it contains all the edges going from any consumer v of u to the killer k_u^t. Otherwise, it contains a unique edge going from u itself to the killer k_u^t. These edges basically encode the fact that the killer k_u^t of u is the last operation that consumes u.

  We note V_N the set of vertices of N and E_N its set of edges.

- We define the supply and demands as follows.
  - For $u \in V$, define $b(u) = \sum_{t \in \mathcal{T}} \sum_{u \in V^{R,t}} \alpha_t$.
  - For $t \in \mathcal{T}$ and $u \in V^{R,t}$, define $b(k_u^t) = -\alpha_t$.
- We define the costs of edges as follows.
  - For $e \in E$, define $\mathit{cost}(e) = -(\delta(e) - \mathrm{II} \times \lambda(e))$.
  - For $t \in \mathcal{T}$ and $u \in V^{R,t}$
    1. if $\mathit{Cons}^t(u) \neq \emptyset$, define $\mathit{cost}(v, k_u^t) = -\left(\delta_{r,t}(v) + \mathrm{II} \times \max_{e=(u,c_u) \in E^{R,t}} \lambda(e)\right)$.
    2. otherwise, define $\mathit{cost}(u, k_u^t) = -1$.
- The capacities are defined as follows.

  For $e \in E_N$, define $\mathit{lcap}(e) = 0$ and $\mathit{ucap}(e) = +\infty$ (i.e. capacities of edges are unbounded).

The dual problem of the scheduling problem is to find a flow of minimum cost in the network previously described.

In Section 1.1, we defined flow for problems when capacities are bounded whereas in this particular case, we deal with unbounded capacities. This is not a problem because there are two possible cases:

1. either there is a cycle in N of infinite negative cost and the problem has no (bounded) solution,
2. or there is no cycle in N of negative cost and then our problem is equivalent to one where $\mathit{ucap}(e) = B$ for a sufficiently large $B \in \mathbb{N}$ (see [1] for more details).

Retrieving the optimal solution of the scheduling problem from the optimal solution of the min-cost flow problem: If the dual problem has a solution, let f be an optimal flow.

Let N_f be the residual network, corresponding to f. In our case, N_f is simply N where reverse edges $\overline{e}$ are added for any $e \in E_N$ such that $f(e) > 0$.

In order to retrieve the solution of the scheduling problem (σ variables), we have to compute a potential on N_f the residual network [1]. Define the distance of $e \in E_N$ to be $\mathit{cost}(e)$ and distance of reverse edge $\overline{e}$ to be $-\mathit{cost}(e)$. Since f is optimal, N_f has no cycle of negative distance. Moreover, optimal primal values can be computed by solving a longest path problem in the residual network, as explained in [1]: simply add a source $\top$ at distance 0 from any other vertex and compute the longest distance from $\top$ to any other vertex.

The values of the computed potential defines σ_u for any $u \in V$ and $\sigma_{k_u^t}$ for any $t \in \mathcal{T}$ and $u \in V^{R,t}$. These values define an optimal solution of the corresponding scheduling problem.

4.2 Experimental results of the min-cost flow implementation of SIRALINA

In this section, we compare the two implementations of SIRALINA. The first one is based on integer linear programming (solved with GLPK), and the second one is based on the min-cost flow algorithm. All register types are optimised conjointly, with the same weight ($\forall t \in \mathcal{T}, \alpha_t = 1$).

4.2.1 Size of the software binaries

We first compare the size of the binaries of our two implementations of SIRALINA. We thus statically linked[1] our two implementations and compared the sizes of the executables. The min-cost network flow version is smaller than the GLPK version, the difference of weight being about 372 kilobytes.

4.2.2 Comparison of the results computed by the two implementations (GLPK vs. min-cost flow)

As we explained in Section 1.4, it may happen that the solutions computed by the two implementations differ, since the scheduling problem has not a unique optimal solution. So the optimal solution computed by GLPK may be different from the optimal solution computed with min-cost network flow algorithm, even if the two objective functions have the same optimal value.

Since the variables of the scheduling problem may have distinct results depending on whether we solve it with GLPK or with a min-cost flow algorithm, the second step of SIRALINA (the linear assignment problem) may produce different results too. Consequently, given a DDG, SIRALINA may compute a different result depending on the internal implementation (GLPK or min-cost flow). In order to have a global rigorous statistical analysis to check if an implementation is better or not than the other, we use the student's t-test as explained in [8] based on the `R` software [7].

For each of the implementation, for each DDG, we thus measured for II=MII the minimal number of registers computed by SIRALINA. All register types are optimised conjointly.

We have then compared the two sets of results (number of registers GLPK vs Network flows) with a two sided student's t-test, in paired mode. The confidence level was set to 90%. The confidence intervals of the average differences between the two sets of results is given below, for each benchmark family.

Benchmark family	Confidence interval
SPEC2000	$-0.0002395127; 0.0006006539$
SPEC2006	no difference
MEDIABENCH	$-0.0005718079; 0.0001394689$
FFMPEG	$-0.0001075718; 0.0004410718$
ALL	$-0.0001265108; 0.0002845636$

Since 0 belongs to these confidence intervals, according to [8], we can conclude that the solutions computed by the GLPK implementation are statistically equivalent to the ones computed by the network flows implementation. To remove 0 from these intervals, we need to decrease confidence level down to 50%, which precisely means that GLPK implementation and network flows implementation give equivalent results statistically (even if they differ in some cases).

4.2.3 Comparison of the execution times of the two implementations (GLPK vs. min-cost flow)

As shown in the previous section, the two implementations of SIRALINA give comparable results regarding their ability to reduce register pressure of innermost loops.

Using the same experiments, we have also compared the execution times of the two implementations. We thus used a two sided student's t-test. Since the execution times may have fluctuations[2], we cannot really use a paired mode for the student's t-test, so we considered a non paired mode. The confidence

[1] Note that due to licensing issues, it may be problematic to statically link an executable against a library covered by the GNU General Public License.

[2] Running the same program with the same input data ends with the same result but not with the execution time

level was set, as previously, to 90%. The confidence intervals of the differences between the two sets of executions times is given below, for each benchmark family.

Benchmark family	Confidence interval
SPEC2000	$-0.004929759; -0.004396605$
SPEC2006	$-0.008450649; -0.006632882$
MEDIABENCH	$-0.006448999; -0.004921148$
FFMPEG	$-0.11596669; -0.04759932$
ALL	$-0.03115889; -0.01521621$

Since 0 is at the right of these confidence intervals, we conclude that the first implementation is statistically faster than the second implementation. In our case, the first implementation was the GLPK based version: this was unexpected because GLPK uses the simplex algorithm which has an exponential complexity in the worst case whereas cost scaling algorithm is a polynomial algorithm. A single sided student's t-test confirms these observations, as shown in the table below.

Benchmark family	Confidence interval
SPEC2000	$-\infty; -0.004455498$
SPEC2006	$-\infty; -0.006833769$
MEDIABENCH	$-\infty; -0.005089972$
FFMPEG	$-\infty; -0.05515299$
ALL	$-\infty; -0.01697704$

Since 0 does not belong to the above confidence intervals, we conclude with a confidence level 90% that the GLPK implementation is faster than the min-cost flow implementation.

The speedup ratio of the GLPK version against the network flow version is summarised in the table below. The ratio was obtained by dividing the sum of execution times of Network Flow implementation by the sum of execution times of GLPK implementation, for each benchmark family.

Benchmark family	Speedup (GLPK vs Network Flow)
SPEC2000	4.50
SPEC2006	5.33
MEDIABENCH	4.67
FFMPEG	5.88
ALL	5.62

We thus observe that the GLPK version is between 4.5 and almost 6 times faster than our network flow implementation. As mentioned previously, this was unexpected and it is rather disappointing. However, it may be possible that an other implementation of the min-cost flow algorithm gives better results. Indeed, initially we used the mean cycle cancelling algorithm to solve the min cost flow problem and the results were at least ten times worse than the actual ones!

General conclusion

SIRALINA is implemented and distributed as an independent C-library (`SIRAlib`). Our implementation can be built based on multiple choices for the solver. If built and linked with LP_SOLVE, then the library becomes under LGPL licence terms. If built and linked with the GNU Linear Programming Kit[5], then the library becomes under GPL licence terms. If the user does not want to use a linear solver, then we implemented a min-cost flow optimisation version of SIRALINA (LGPL).

SIRALINA heuristic is designed to bound the register pressure in innermost loops. It can be adapted, as shown in this report, to reduce the acyclic register saturation of a DAG (basic block and super-block).

Experiments, led over a large set of common benchmarks (FFMPEG, MEDIABENCH, SPEC2000, SPEC2006), have shown that SIRALINA succeeds in limiting the register requirement of innermost loops. Hence SIRALINA avoids the generation of spill code in about 95% of the loops. These results are not at the expense of instruction level parallelism. Indeed, except from some critical cases, the increase of the MII remains low (less than 5% in average). Experiments on innermost loops have also shown that it is better to optimise all register types conjointly instead of one by one (regarding the register need as well as the execution times).

We have also demonstrated how SIRALINA can be exploited to reduce the acyclic register saturation of directed acyclic data dependency graphs. We have thus given a register saturation reduction heuristic based on SIRALINA. Experiments have shown that this heuristic achieves its goal successfully. Indeed, thanks to this heuristic, we were able to make about 80% of the acyclic graphs satisfy strong register constraints. Not only our register saturation reduction heuristic is good for reducing significantly acyclic register saturation but also its impact on the longest path –and thus on the scheduling quality– is negligible. Regarding the register optimisation orders, experiments have shown that the chosen order can greatly alter the success ratio. However we have found out that optimising all register types conjointly is generally one of the best possible strategies. Moreover it has the advantage to perform a bit faster than the other strategies that involve the register types in sequential order. We would thus recommend, as for the case of innermost loops, to use the generalised SIRALINA technique dealing with the register types conjointly.

In this report, we have also presented a min-cost flow implementation of SIRALINA. We then have experimentally compared the two implementations (GLPK vs. min-cost flow). The experiments, led over a set of the benchmarks (FFMPEG, SPEC2000, SPEC2006, MEDIABENCH), have shown that the two implementations give similar results, regarding their power to bound the register pressure in innermost loops.

However, and quite surprisingly, the straightforward implementation relying on GLPK is about much more faster in average than the min-cost flow implementation (between 4 and 5 times faster). This is surprising because the network flow implementation has a polynomial complexity whereas the GLPK one has an pseudo-polynomial (exponential) complexity, since it uses the simplex algorithm to solve a linear problem.

Nevertheless, we think that our min-cost flow implementation has its own interest. Firstly, as we have seen, our network flow implementation is lighter in terms of binary size than our implementation relying on GLPK. If the size of the binary could at first sight seem to be a minor detail, it becomes an important point when considering an integration into a just-in-time compiler running on embedded devices, where each byte of memory is precious. Secondly, using an external library such as GLPK can cause license issues that do not apply to our standalone implementation. Indeed, linking statically a program against a library covered by the GPL such as GLPK could be a problem for an industrial compiler. Finally, it may well be possible that a better implementation of a min cost flow algorithm (which is the heart of our algorithmic method) results in a program faster than the GLPK version.

Bibliography

[1] *Network Flows: Theory, Algorithms, and Applications.* Prentice Hall, 1993.

[2] Mokhtar S. Bazaraa, John J. Jarvis, and Hanif D. Sherali. *Linear Programming and Network Flows, 3rd Edition.* Wiley, 2005.

[3] Sebastien Briais and Sid-Ahmed-Ali Touati. Experimental Study of Register Saturation in Basic Blocks and Super-Blocks: Optimality and Heuristics. Technical Report HAL-INRIA-00431103, University of Versailles Saint-Quentin en Yvelines, October 2009. Research report, http://hal.inria.fr/inria-00431103.

[4] Karine Deschinkel and Sid-Ahmed-Ali Touati. Efficient method for periodic task scheduling with storage requirement minimization. In *Proceedings of 2nd Annual International Conference on Combinatorial Optimization and Applications (COCOA 2008)*, LNCS, Saint Johns, Newfoundland, Canada, August 2008. Springer.

[5] Free Software Foundation. Gnu Linear Programming Kit.

[6] H. W. Kuhn. The hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2(1-2):83–97, 1955.

[7] R Development Core Team. *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria, 2008. ISBN 3-900051-07-0.

[8] Raj Jain. *The Art of Computer Systems Performance Analysis : Techniques for Experimental Design, Measurement, Simulation, and Modelling.* John Wiley and Sons, inc., New York, 1991.

[9] Sid-Ahmed-Ali Touati. *Register Pressure in Instruction Level Parallelism.* PhD thesis, Université de Versailles Saint-Quentin, 2002.

[10] Sid-Ahmed-Ali Touati. Register saturation in instruction level parallelism. *International Journal of Parallel Programming*, 33(4), August 2005.

[11] Sid-Ahmed-Ali Touati and Christine Eisenbeis. Early periodic register allocation on ilp processors. *Parallel Processing Letters*, 14(2), 2004.

UNIVERSITÉ DE VERSAILLES
SAINT-QUENTIN-EN-YVELINES

UFR des sciences	:	45 avenue des Etats Unis. 78035 Versailles cedex
IUT de Velizy et de Rambouillet	:	10-12 avenue de l'Europe. 78140 Vélizy.
UFR des Sciences Sociales et des Humanité	:	47 boulevard Vauban. 78047 Guyancourt cedex
Faculté de droit et de science politique	:	3, rue de la Division Leclerc. 78280 Guyancourt
IUT de Mantes en Yvelines	:	7 rue Jean Hoët - 78200 Mantes la Jolie
UFR de Médecine Paris-Ile-de-France Ouest	:	9 boulevard d'Alembert Bâtiment François Rabelais. 78280 Guyancourt
Institut des Langues et des Etudes Internationales	:	5-7, boulevard d'Alembert. 78280 Guyancourt
Institut des Sciences et Techniques des Yvelines	:	45 avenue des Etas Unis - 78035 Versailles cedex
Observatoire des Sciences de l'Univers de l'UVSQ	:	11 boulevard d'Alembert. 78280 Guyancourt

Éditeur
UVSQ - Siège : 55 avenue de Paris - 78035 Versailles Cedex (France)
http://www.uvsq.fr