Experimental Study of Register Saturation in Basic Blocks and Super-Blocks: Optimality and heuristics

Sébastien Briais, Sid Touati

UNIVERSITE DE VERSAILLES SAINT-QUENTIN EN YVELINES

Experimental Study of Register Saturation in Basic Blocks and Super-Blocks: Optimality and heuristics

Sébastien BRIAIS — Sid-Ahmed-Ali TOUATI

N° HAL-INRIA-00431103

Octobre 2009

Rapport de recherche

Experimental Study of Register Saturation in Basic Blocks and Super-Blocks: Optimality and heuristics

Sébastien Briais* , Sid-Ahmed-Ali Touati†

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

Rapport de recherche n° HAL-INRIA-00431103 — Octobre 2009 — 33 pages

Abstract: Register saturation (RS) is the exact maximal register need of all valid schedules of a data dependence graph [4]. Its optimal computation is NP-complete. This report proposes two variants of heuristics for computing the acyclic RS of directed acyclic graphs (DAG). The first one improves the previous greedy-k heuristic [4] in terms of approximating the RS with equivalent computation times. The second heuristic is faster, has better RS approximation than greedy-k, but scarifies the computation of saturating values. In order to evaluate the efficiency of these two heuristics, we designed an optimal combinatorial algorithm computing the optimal RS for tractable cases, which turns out to be satisfactory in practice. Extensive experiments have been conducted on thousands of data dependence graphs extracted from FFMPEG, MEDIABENCH, SPEC2000 and SPEC2006 benchmarks. Numerical results are presented to demonstrate the efficiency of the two proposed heuristics, so hence they can replace the greedy-k heuristic presented in [4]. Our RS computation methods are distributed as a C independent library (`RSlib`) under LGPL licence.

Key-words: Compilation, Code optimisation, Register saturation, 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

Étude expérimentale de la saturation en registres dans les blocs de base et les super-blocs: Optimalité et heuristiques

Résumé : La saturation en registres (SR) est la borne exacte maximale et atteignable du besoin en registres de l'ensemble des ordonnancements possibles d'un graphe de dépendances de donnés [4]. Le calcul de la SR étant NP-complet, ce rapport présente deux heuristiques pour le calcul de la SR acyclique (pour les blocs de bases et les super-blocs d'un programme). Une première heuristique améliore l'algorithme greedy-k [4] en terme d'approximation de la valeur optimale de la SR, en ayant un temps de calcul équivalent. Une deuxième heuristique est plus rapide que greedy-k, a une meilleure approximation de la SR optimale, mais sacrifie le calcul des valeurs saturantes. Afin de tester l'efficacité de nos heuristiques, nous avons conçu un algorithme optimal et combinatoire pour le calcul de la SR. Malgré la NP-complétude du problème, notre algorithme optimal permet de résoudre assez d'instances pratiques pour faire une étude satisfaisante. Des expériences massives ont été conduites sur les graphes de dépendances de données des benchmarks FFMPEG, MEDIABENCH, SPEC2000 et SPEC2006. Les résultats numériques montrent que nos deux nouvelles heuristiques sont efficaces et rapides, elles peuvent donc remplacer greedy-k. La méthode optimale (exponentielle) et les heuristiques sont distribuées dans une librairie C indépendante (`RSlib`) sous licence LGPL.

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

Contents

Introduction

The register saturation is a computer science concept initially studied in [4], intended for decoupling register constraints from resource constraints in back-end optimising compilation. Integrating register saturation computation into a compiler back-end can have several benefits. For instance, if register saturation is not greater than the maximum number available of registers of the target architecture, aggressive scheduling algorithms can be used since any scheduling of the DAG is guaranteed to fit in the target architecture register constraints. This breaks the usual iterative problem of scheduling followed by spilling.

Register saturation (RS) is well adapted to situations where register spilling is not a favourite or a possible solution for reducing register pressure compared to instruction scheduling: spill operations request memory data with a higher energy consumption. Also, spill code introduces unpredictable cache effects: it makes Worst-Case-Execution-Time estimation less accurate and add difficulties to instruction scheduling (because spill operations latencies are unknown at compile time). Register Saturation (RS) is concerned about register maximisation not minimisation, and has some major proved mathematical characteristics [4]. RS is an exact *reachable* upper-bound of the register requirement of all possible valid instruction schedules, independently of resource constraints. This characteristic makes RS outperforms register sufficiency, because the register sufficiency (minimal register requirement) is not a reachable lower-bound: the register sufficiency value depends on the resource constraints, a toy example demonstrating this fact has already been shown in [3].

In this report, we analyse the numerical results obtained after massive experimentations of two variants of the heuristic presented in [4]. This heuristic aims at estimating *register saturation* of acyclic data dependency graphs (DAG). DAG represent either basic blocks or super-blocks. This report is organised as follows. In Chapter 1, we recall the definition of register saturation of acyclic data dependency graphs and remind the heuristic of [4]. In Chapter 2, we describe an exponential algorithm that computes exact register saturation of DAG. We also give two two improved variants for the greedy-k heuristic [4]. In Chapter 3, we go through experimental results and analyse them. At last, we conclude.

Chapter 1

Background on register saturation

In this chapter, we recall briefly the definition of register saturation in DAG and the greedy-k heuristic[4]. Before that, we start by some used notations.

1.1 Mathematical notations and definitions

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

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 $\phi : E \to V \times V$. 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 $source : E \to V$ and $target : E \to V$ 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* $\deg^+_G(u) = 0$.

- 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 $target(e_n) = source(e_1)$.

  Note that an acyclic graph has at least one source and one sink.

- For $u, v \in V$, we write $u <_G v$ *iff* there exists a path $e_1, \ldots, e_n$ in G such that $source(e_1) = u$ and $target(e_n) = v$. In such a case, we say that v is a *descendant* of u and that u is an *ascendant* of v.

- For $u, v \in V$, we write $u \|_G v$ *iff* neither $u <_G v$ nor $v <_G u$.

  Note that G is acyclic *iff* for any $u \in V$, $u \|_G u$.

- For $u \in V$, we define $\uparrow_G u = \{v \in V \mid v = u \vee v <_G u\}$ (i.e. the ascendants of u including u itself) and $\downarrow_G u = \{v \in V \mid v = u \vee u <_G v\}$ (i.e. the descendants of u including u itself).

- An antichain in G is a set of vertices $A \subseteq V$ such that for any $u, v \in A$, $u \| v$. It is maximal *iff* there are no antichain A' in G bigger than A (i.e. such that $|A'| > |A|$).

- 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 $(V, E \cup E')$.

By abuse of notation, when G is clear from the context, we sometimes omit to precise it.

1.2 DAG and processor model

The data dependencies between the instructions of a basic block or a super-block are represented by a directed acyclic graph (DAG)[1]. It is a triple $G = (V, E, \delta)$ where V is a set of vertices (instructions), E is a set of directed edges (data dependencies and serial constraints) such that (V, E) is an acyclic graph, and $\delta : E \to \mathbb{Z}$ gives the latency of edges. 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 schedule $\sigma : V \to \mathbb{N}$ of G is a function which gives an execution time for each operation. It is valid *iff* it satisfies

$$\forall e \in E : \sigma(source(e)) + \delta(e) \leq \sigma(target(e))$$

The set of all valid schedules is written $\Sigma(G)$.

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

Thus, the total schedule time of a schedule σ is simply $\sigma(\bot)$.

The modelled processor may have several register types (sometimes called register classes): we note $\mathcal{T}$ the set of available 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 might be a value of several types simultaneously. However, our processor model does not support operations that store two or more values in registers of a same type t. The multiple results of the same operation must have distinct types.

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.

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 Recall on register saturation

In the following, we assume to have a DAG $G = (V, E, \delta)$ and a type $t \in \mathcal{T}$. The set of *consumers* (readers) of a value $u \in V^{R,t}$ is:

$$Cons^t(u) = \begin{cases} \{target(e) \mid e \in E^{R,t} \wedge source(e) = u\} & \text{if } \exists e \in E^{R,t} : source(e) = u \\ \{\bot\} & \text{otherwise} \end{cases}$$

Given a schedule $\sigma \in \Sigma(G)$ and $u \in V^{R,t}$, the *killing date* of u is

$$kill^t_\sigma(u) = \max_{v \in Cons^t(u)} (\sigma(v) + \delta_{r,t}(v))$$

Given a schedule $\sigma \in \Sigma(G)$ and $u \in V^{R,t}$, the set of *killers* of u is

$$killers^t_\sigma(u) = \{v \in Cons^t(u) \mid \sigma(v) + \delta_{r,t}(v) = kill^t_\sigma(u)\}$$

The *lifetime interval* of a value $u \in V^{R,t}$ is

$$LT^t_\sigma(u) =]\sigma(u) + \delta_{w,t}(u), kill^t_\sigma(u)]$$

At a given time i, the set of *values simultaneously alive* is

$$vsa^t_\sigma(i) = \{u \in V^{R,t} \mid i \in LT^t_\sigma(u)\}$$

[1] In this report, we use both the terms DDG and DAG as an abbreviation of acyclic data dependence graph.

The *register need* of a valid schedule σ is the maximum number of values simultaneously alive, i.e.

$$RN^t_\sigma(G) = \max_{0 \leq i \leq \sigma(\bot)} |vsa^t_\sigma(i)|$$

The *register saturation* is simply the maximum register need, amongst all valid schedule, i.e.

$$RS^t(G) = \max_{\sigma \in \Sigma(G)} RN^t_\sigma(G)$$

A *saturating schedule* of G for register type t is any valid schedule that requires exactly $RS^t(G)$ registers. The following theorem has been proved in [3].

Theorem 1. *Given a DAG $G = (V, E, \delta)$, computing $RS^t(G)$ the register saturation of a given register type is NP-complete.*

1.4 A characterisation of register saturation

As before, we assume to have a DAG $G = (V, E, \delta)$ and a type $t \in \mathcal{T}$. The set of *potential killers* of a value $u \in V^{R,t}$ is

$$pkill^t(u) = \{v \in Cons^t(u) \mid \downarrow_G v \cap Cons^t(u) = \{v\}\}$$

The *potential killing* DAG of G is $PK^t(G) = (V, E^t_{PK})$ where

$$E^t_{PK} = \{(u, v) \mid u \in V^{R,t} \wedge v \in pkill^t(u)\}$$

A *killing function* is a function $k : V^{R,t} \rightarrow V$ which selects for every value $u \in V^{R,t}$ a killer $v \in pkill^t(u)$.

A killing function is *valid iff* the graph $G_{\rightarrow k} = G \backslash^{E_k}$ associated with k is acyclic, where $E_k = \{(v, k(u)) \mid u \in V^{R,t}, v \in pkill^t(u) \setminus \{k(u)\}\}$.

Note that any schedule of $G_{\rightarrow k}$ is also a schedule of G, i.e. $\Sigma(G_{\rightarrow k}) \subseteq \Sigma(G)$.

Given a killing function k, the *disjoint value* graph associated with k is $DV^t_k(G) = (V^{R,t}, E_{DV})$ where $E_{DV} = \{(u, v) \mid u, v \in V^{R,t} \wedge v \in \downarrow_{G_{\rightarrow k}} k(u)\}$.

The following theorem[2], taken from [4], gives the link between register need of a schedule and a maximal antichain in the disjoint value graph.

Theorem 2. *Let G be a DAG, k a valid killing function and $t \in \mathcal{T}$ a register type. Let AM_k be a maximal antichain in the disjoint value graph $DV^t_k(G)$. Then:*

1. $\forall \sigma \in \Sigma(G_{\rightarrow k}) : RN^t_\sigma(G) \leq |AM_k|$
2. $\exists \sigma \in \Sigma(G_{\rightarrow k}) : RN^t_\sigma(G) = |AM_k|$

Finally, the following theorem, still taken from [4], gives an interesting characterisation of register saturation.

Theorem 3. *Let G be a DAG and $t \in \mathcal{T}$ a register type. Then*

$$RS^t(G) = \max_{\substack{k \text{ is a valid killing function of } G \\ AM_k \text{ is a maximal antichain of } DV^t_k(G)}} |AM_k|$$

A killing function that maximise $|AM_k|$ is called a *saturating killing function*. So our greedy-k heuristic tries to construct such saturating killing function, as described in the next section.

[2]The proof of the second point involves an algorithm that extends G in G' such that any schedule σ of G' is such that $vsa^t_\sigma(i) = AM_k$ for some i. We do not detail this algorithm here, but refer the interested reader to [4] for instance.

1.5 Greedy-k algorithm

Despite the computation of the register saturation is a NP-complete problem, it is possible to devise efficient heuristics that work quite well in practice. Such an heuristic is presented in [4].

The heuristic greedy-k of [4] relies on Theorem 3. It works by establishing greedily a valid killing function k which aims at maximising the size of a maximal antichain in $DV^t_k(G)$.

The heuristic examines one after one each *connected bipartite component* of $PK^t(G)$ and constructs progressively a killing function.

A connected bipartite component of $PK^t(G)$ is a triple $cb = (S_{cb}, T_{cb}, E_{cb})$ such that:

- $S_{cb} \cap T_{cb} = \emptyset$;
- $E_{cb} \subseteq E^t_{PK}$;
- $S_{cb} \subseteq V^{R,t}$;
- $T_{cb} \subseteq V$ such that any operation $t \in T_{cb}$ is a potential killer of at least one value of S_{cb}.

A bipartite decomposition of $PK^t(G)$ is a set of connected bipartite component $\mathcal{B}(G)$ such that for any $e \in E^t_{PK}$, there exists $cb = (S_{cb}, T_{cb}, E_{cb}) \in \mathcal{B}(G)$ such that $e \in E_{cb}$. According to [3], this decomposition is unique.

For further details on connected bipartite components and bipartite decomposition, we refer the interested reader to [3].

Algorithm 1 Greedy-k heuristic

```
function GREEDY-K(G = (V, E, δ), t)
    for all u ∈ V^{R,t} do
        k(u) ← ⊥
    end for
    build B(G) the bipartite decomposition of PK^t(G)
    for all connected bipartite component cb = (S_cb, T_cb, E_cb) ∈ B(G) do
        X ← S_cb                                          ▷ Values to kill
        Y ← ∅
        while X ≠ ∅ do
            select w ∈ T_cb which maximises ρ_{X,Y,cb}(w)  ▷ Chose a killer
            for all s ∈ Γ⁻_cb(tw) do                       ▷ Make it kill its yet unkilled parents
                if k(s) = ⊥ then
                    k(s) ← w
                end if
            end for
            X ← X \ Γ⁻_cb(w)                              ▷ Remove killed values
            Y ← Y ∪ (↓_G w ∩ V^{R,t})                      ▷ Add descendant values
        end while
    end for
    return k
end function
```

The greedy-k heuristic is detailed in Algorithm 1. It examines one after the other each $(S_{cb}, T_{cb}, E_{cb}) \in \mathcal{B}(G)$ and select greedily a killer $w \in T_{cb}$ that maximises the ratio $\rho_{X,Y,cb}(w)$ to kill values of S_{cb}. The ratio $\rho_{X,Y,cb}(w)$ is given by the following formula.

$$\rho_{X,Y,cb}(w) = \frac{|X \cap \Gamma^-_{cb}(w)|}{\max(1, |Y \cup (\downarrow_G w \cap V^{R,t})|)}$$

This ratio is a trade-off between the number of values killed by w, and the number of edges that will connect w to descendant values in $DV^t_k(G)$.

By always selecting a killer that maximises this ratio, the greedy-k heuristic aims at minimising the number of edges in $DV^t_k(G)$; the intuition being that the more edges there are in $DV^t_k(G)$, the less its width (the size of a maximal antichain) is.

Given a DAG $G = (V, E, \delta)$ and a type $t \in \mathcal{T}$, the estimation of the register saturation by greedy-k heuristic is the size of a maximal antichain AM_k in $DV^t_k(G)$ where $k = \textsc{Greedy-k}(G, t)$. Note that it is always lesser than or equal to $RS^t(G)$.

The next chapter studies the problem of register saturation computation (optimal and approximate methods).

Chapter 2

Algorithms computing the register saturation

In this chapter, we give a combinatorial (exponential) algorithm for computing the optimal register saturation and we present two refinements of the greedy-k heuristic.

2.1 A combinatorial algorithm for computing the optimal register saturation

In order to evaluate experimentally the quality of an heuristic, we need an algorithm to compute the exact register saturation.

In [4], an algorithm that uses *integer linear programming* is given. This solution is not tractable in practice and work only for toy examples. We thus instead use the straightforward combinatorial algorithm suggested by Theorem 3. It has the advantage to use very few memory (when written carefully), and performs not so bad in practice, on reasonable sized DAGs. The details are given in Algorithm 2. The algorithm proceeds by exploring all possible killing functions (all combinations of killers). Each possible killing function k is first checked for validity (by checking if $G_{\rightarrow k}$ the associated DAG is acyclic[1]). If so, then the algorithm builds the disjoint value DAG and computes the maximal antichain. By exploring all possible combinations of valid killing functions, and by keeping track of the maximal value of $|AM_k|$, we reach the optimal RS.

2.2 Improving greedy-k heuristic

Firstly we present a first variant with a new cost function that improves the efficiency of the greedy-k heuristic. Secondly, we present a second variant, which is an approximation of the greedy-k heuristic, and results in a faster algorithm.

2.2.1 Variant 1 heuristic: improving the greedy-k heuristic

After massive experimentations, we have devised several modifications to greedy-k heuristic that makes it perform (always) better in practice.

These modifications are based on the observation that the original formula for $\rho_{X,Y,cb}(w)$ (recall that by definition $\rho_{X,Y,cb}(w) = \frac{|X \cap \Gamma^-_{cb}(w)|}{\max(1,|Y \cup (\downarrow_G w \cap V^{R,t})|)}$) does not make any distinction between values and non-values when time comes to chose a killer. Our claim is that it makes a difference. Indeed, if w is a value, then we have $|Y \cup (\downarrow_G w \cap V^{R,t})| \geq 1$ (since at least w belongs to this set) whereas if w is not a value, it might be the case that $|Y \cup (\downarrow_G w \cap V^{R,t})| = 0$. Indeed, this happens when $Y = \emptyset$ and $(\downarrow_G w \cap V^{R,t}) = \emptyset$.

According to greedy-k algorithm, Y remains empty as long as w is chosen such that $(\downarrow_G w \cap V^{R,t}) = \emptyset$. Observe that such a chosen w does not add any edges in $DV^t_k(G)$ and thus is certainly a good choice for

[1] By abuse of language, we say that a killer is valid if it does not create a circuit inside $G_{\rightarrow k}$.

Algorithm 2 Optimal register saturation

function RS($G = (V, E, \delta), t$)
  let $\{v_0, \ldots, v_{n-1}\} = V^{R,t}$
  $RS \leftarrow 0$
  $G^k = (V, E^k) \leftarrow G$ ▷ G^k is an incremental construction of $G_{\rightarrow k}$
  for $i \leftarrow 0, n-1$ **do** ▷ Iterating on all the vertices of the DAG
    $k(v_i) \leftarrow \bot$
    $E_i \leftarrow \emptyset$ ▷ Set of added edges from v_i in G^k
  end for
  $s \leftarrow 1$ ▷ A flag set to 1 to construct a valid killing function, set to -1 for back tracking from a non valid one
  $i \leftarrow 0$ ▷ i is the node number i
  while $i \geq 0$ **do**
    if $\deg^+_{PK^t(G)}(v_i) = 0$ **then** ▷ If no potential killer, skip
      $i \leftarrow i + s$
    else
      if $E_i \neq \emptyset$ **then** ▷ Here is a case of back-tracking, undo previous choice of killer
        $k(v_i) \leftarrow \bot$
        $E^k \leftarrow E^k \setminus E_i$
        $E_i \leftarrow \emptyset$
      else ▷ Initialise a new exploration of the killers of i
        $\Gamma_i \leftarrow \Gamma^+_{PK^t(G)}(v_i)$
      end if
      $s \leftarrow -1$ ▷ Backtrack unless a valid killer is found
      while $\Gamma_i \neq \emptyset \wedge s = -1$ **do** ▷ Search a valid killer for v_i
        let $k_v \in \Gamma_i$
        $\Gamma_i \leftarrow \Gamma_i \setminus \{k_v\}$
        $E_i \leftarrow \{(k_v, u) \mid u \in \Gamma^+_{PK^t(G)}(v_i) \setminus \{k_v\}\}$
        $E^k \leftarrow E^k \uplus E_i$ ▷ Update G^k with edges of E_i
        if G^k is acyclic **then** ▷ If the killing function is valid
          $s \leftarrow 1$ ▷ Continue the exploration of all possible killing functions
          $k(v_i) \leftarrow k_v$ ▷ Fix the selection of the killer
        else ▷ Otherwise undo modification in G^k
          $E^k \leftarrow E^k \setminus E_i$
          $E_i \leftarrow \emptyset$
        end if
      end while
      $i \leftarrow i + s$
    end if
    if $i = n$ **then** ▷ The killing function is fully defined
      Compute a maximal antichain AM_k of $DV^t_k(G)$
      if $|AM_k| > RS|$ **then**
        $RS \leftarrow |AM_k|$
      end if
      $s \leftarrow -1$ ▷ Backtrack
      $i \leftarrow i + s$ ▷ If backtrack come back to the previous node. Otherwise, continue to the next node.
    end if
  end while
  return RS
end function

minimising its number of edges. So we modify greedy-k heuristic to first chose w such that $\downarrow_G w \cap V^{R,t} = \emptyset$. If such a w does not exist, we still chose a w that maximises $\rho_{X,Y,cb}(w)$. However, in the variant 1 heuristic, we modify the formula of the cost function as follows:

$$\rho'_{X,Y,cb}(w) = \frac{|X \cap \Gamma^-_{cb}(w)|}{1 + |Y \cup (\downarrow_G w \cap V^{R,t})|}$$

Thus we have removed the max operator that acted as a treshold, giving more importance to potential killers instead of discriminating them. The modified heuristic is presented in Algorithm 3.

Algorithm 3 Variant 1 heuristic

function MODIFIED-GREEDY-K($G = (V, E, \delta), t$)
  for all $u \in V^{R,t}$ **do**
    $k(u) \leftarrow \perp$
  end for
  build $\mathcal{B}(G)$ the bipartite decomposition of $PK^t(G)$
  for all connected bipartite component $cb = (S_{cb}, T_{cb}, E_{cb}) \in \mathcal{B}(G)$ **do**
    $X \leftarrow S_{cb}$ ▷ Values to kill
    $Y \leftarrow \emptyset$
    while $X \neq \emptyset$ **do**
      select $w \in T_{cb}$ such that $\downarrow_G w \cap V^{R,t} = \emptyset$ ▷ Choice function is modified
      if no such w exists, select $w \in T_{cb}$ which maximises $\rho'_{X,Y,cb}(w)$ ▷ Use the new cost function
      for all $s \in \Gamma^-_{cb}(tw)$ **do** ▷ Make it kill its yet unkilled parents
        if $k(s) = \perp$ **then**
          $k(s) \leftarrow w$
        end if
      end for
      $X \leftarrow X \setminus \Gamma^-_{cb}(w)$ ▷ Remove killed values
      $Y \leftarrow Y \cup (\downarrow_G w \cap V^{R,t})$ ▷ Add descendant values
    end while
  end for
  return k
end function

2.2.2 Variant 2 heuristic: a faster algorithm

Computing the disjoint value graph $DV^t_k(G)$ is an expensive task in practice, because it implies to compute first the graph $G_{\rightarrow k}$ associated with k, and then its transitive closure (to gain fast access to descendants in $G_{\rightarrow k}$).

An approximation of $DV^t_k(G)$ is to use G instead of $G_{\rightarrow k}$ in its definition. So, the variant 2 heuristic is made from greedy-k by applying the two following modifications:

1. Use the modified cost function of variant 1 (see Algorithm 3);
2. When computing $DV^t_k(G)$, we use use G instead of $G_{\rightarrow k}$ (this would save the cost of building $G_{\rightarrow k}$).

Theoretically, this modification hurts because the computed register saturation has no reasons to be a reachable register need (may over-estimate RS). In particular, a maximal antichain in this modified $DV^t_k(G)$ is not a set of saturating values of G. However, if we are interested in just the final value of the register saturation (without needing the saturating values), we will see in Chapter 3 that in practice this modified algorithm gives good estimation for the register saturation while being much faster (more than 25%) than the original greedy-k algorithm and faster than variant 1.

Chapter 3

Experimental results of register saturation computation

The next section starts by presenting the experimental setup.

3.1 Experimental setup and environment

3.1.1 The data dependency graphs used as input test data

The data dependency graphs (DDG) used for experiments come from the C and C++ applications of SPEC2000, SPEC2006, MEDIABENCH and FFMPEG sets of benchmarks. They have been analysed by the st200cc compiler (industrial compiler from STmicroelectronics, based on the Open64 compiler). We capture all the DDG going to the software pipelining module, after super-block formation. The experimented basic blocks and super-blocks are simply the DAGs of the bodies of loops. The number of DDG per benchmark family is the following (ALL indicates the whole set of benchmarks).

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 $\mathcal{T} = \{\text{FP}, \text{GR}, \text{BR}\}$. 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 familty):

- 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[1] of the sizes (the number of vertices) of the DAGs is the following.

[1] 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

	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 is the following (only benchmarks that involve the considered type are taken into account).

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

Since the compiler may unroll loops to achieve better performance, we have also experimented the DDG after loop unrolling with a factor of four (so the DDG sizes are multiplied by a factor of five). The distribution of the sizes of the unrolled loops may be computed by multiplying the previous sizes by a factor of five.

3.1.2 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.

3.1.3 Measured data during the experiments

For each DAG G, in addition to the elapsed computation time, we have measured:

- The optimal register saturation $RS^t(G)$ for $t \in \mathcal{T}$, using Algorithm 2. Since this algorithm is exponential, we have included a timeout (one hour) so that computation is stopped after one hour per DAG. We have reported in each case whether the computation has completed during this slice of time and measured the elapsed time when it completed in less than one hour.
- The approximated register saturation $RS_0^t(G)$ for $t \in \mathcal{T}$, as computed by greedy-k heuristic (see Section 2.2.1).
- The approximated register saturation $RS_1^t(G)$ for $t \in \mathcal{T}$, as computed by variant 1 heuristic (see Section 2.2.1). We call this version of the heuristic the *precise heuristic* because it computes a reachable RS .
- The approximated register saturation $RS_2^t(G)$ for $t \in \mathcal{T}$, as computed by variant 2 heuristic (see Section 2.2.2). Variant 2 is called *approximate heuristic* because it may over-estimate RS in theory.
- We measured the shortest schedules of G according to two configurations:
  1. By considering data dependency constraints only. In this case, the best schedule of G is the longest path, also called the critical path.

2. By considering data dependency constraints and resource constraints. In this case, the shortest possible schedule is the classical maximum value between the critical path of the DAG and the functional units capacities. The functional units constraints are defined in Section 3.1.1.

- Similarly to G, we measured the best schedules of G^t_s (for $t \in \mathcal{T}$) where G^t_s is an extension of G such that any schedule of it is a saturating one for the register type t. That is, any schedule of G^t_s needs exactly $RS^t(G)$ registers of type t: Algorithm 1 from [4] computes G^t_s. These measures allow us to analyse if saturating the register requirement has any influence on the quality of the best possible schedules.

In the next sections, we analyse the results of the two variants of the greedy-k heuristic presented in the previous chapter. Let us start by first looking at the optimal values of RS computed by the exponential algorithm (Algorithm 2)

3.2 Analysis of the optimal algorithm of RS computation

3.2.1 Running times of the exponential algorithm

We used the exponential algorithm to compute exact FP, GR and BR register saturation of original loop bodies and of bodies of loops unrolled four times. The computation was interrupted when it lasted more than one hour. Figure 3.1 illustrated the proportion of the DAGs, per register type, where optimal RS computation finishes within one hour. As can be seen, without loop unrolling, we succeed in computing the optimal RS of most of the DAG. When we unroll the loops 4 times (making the DDG sizes to increase by a factor of 5), optimal RS computation becomes more difficult, but still possible in most of the cases.

When RS computation completed successfully within the allocated time, we recorded the execution times. Distributions of these are shown on Figure 3.2. This figure represents boxplots [2].

From the previous experiments, we conclude that the exponential algorithm is usable in practice with reasonably medium sized DAGs. Indeed, we successfully computed FP, GR and BR saturation of more than 95% of the original loop bodies. The execution time did not exceed 45 ms in 75% of these cases.

However, when size of the DAG become critical, performance drops down dramatically. Thus, even if we managed to computed FP and BR saturation of more than 80% of the bodies of the loops unrolled four times, we were able to compute GR saturation of only 45% of these bodies. Execution times also literally exploded, compared to the ones obtained for initial loop bodies: the slowdown factor ranges from 10 to over 1000.

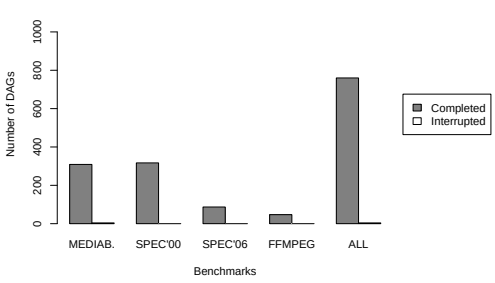
3.2.2 Distribution of the optimal register saturation

We measured exact FP, GR and BR register saturation of initial loop bodies and of bodies of loops unrolled four times, as computed by the exponential algorithm. When computation did not complete in one hour, we took as saturation value the best approximation found by the algorithm: according to the previous results. The distributions of RS are shown on Figure 3.3.

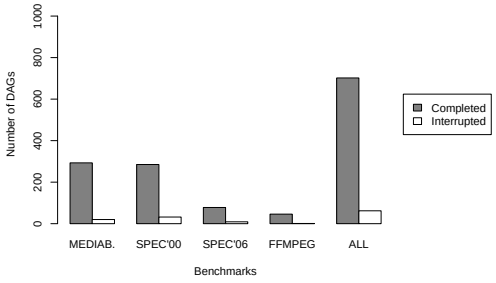
From Figure 3.3, we deduce that register saturation of initial loop bodies are quite low. Indeed, the median value is 2 for FP saturation, 7 for GR saturation and 1 for BR saturation. On bodies of loops unrolled four times, register pressure is bigger but still quite limited: the median value is thus 3 for FP saturation, 18 for GR saturation and 5 for BR saturation.

The next section demonstrates that variant 1 and 2 are better alternatives than greedy-k to approximate RS computation.

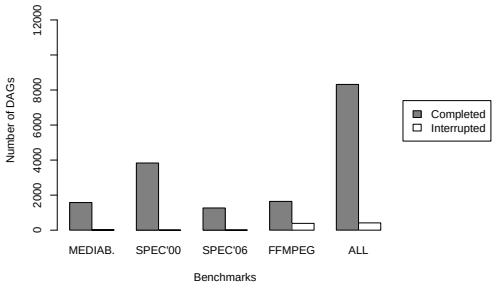
[2] Boxplot, also known as *box-and-whisker diagram*, is a convenient way of graphically depicting groups of numerical data through their five-number summaries: the smallest observations (min), lower quartile ($Q1 = 25\%$), median ($Q2 = 50\%$), upper quartile ($Q3 = 75\%$), and largest observations (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.



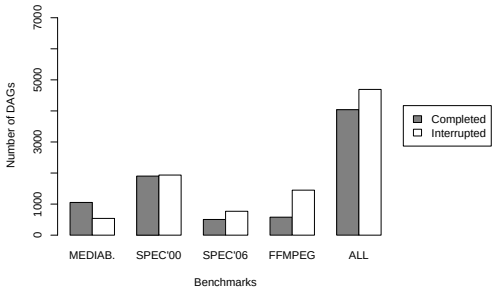
(a) type FP, no unrolling



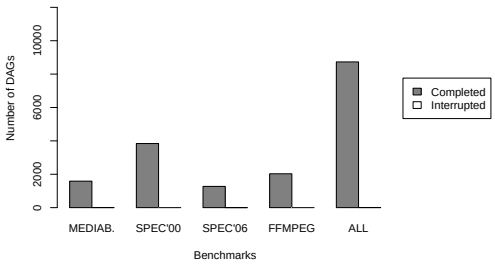
(b) type FP, unrolling = 4×



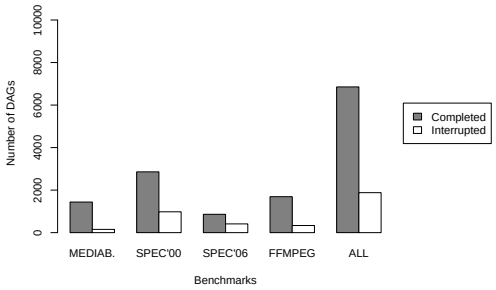
(c) type GR, no unrolling



(d) type GR, unrolling = 4×

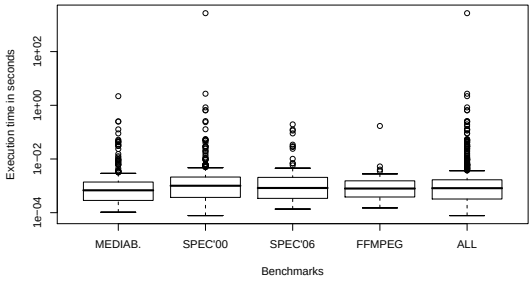


(e) type BR, no unrolling

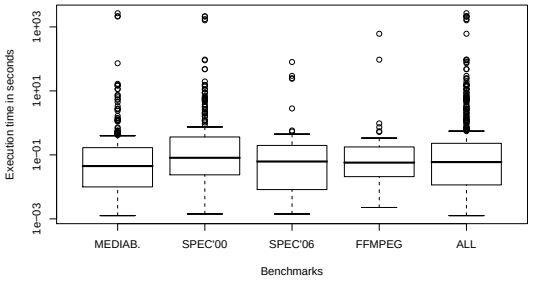


(f) type BR, unrolling = 4×

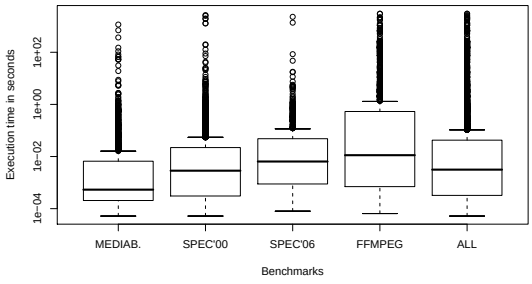
Figure 3.1: Computation of exact register saturation in less than one hour



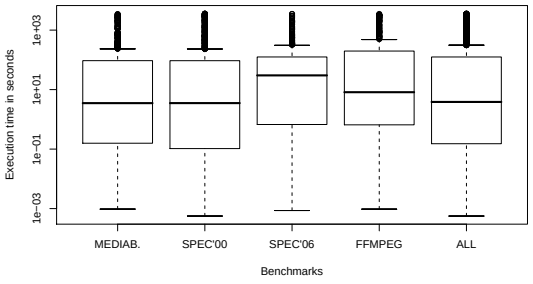
(a) type FP, no unrolling



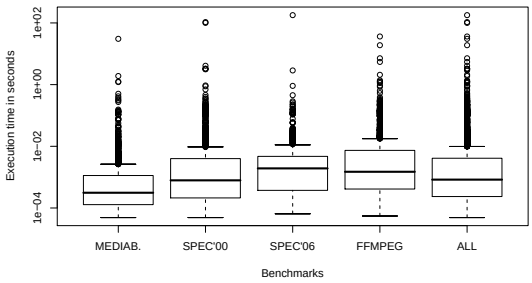
(b) type FP, unrolling = 4×



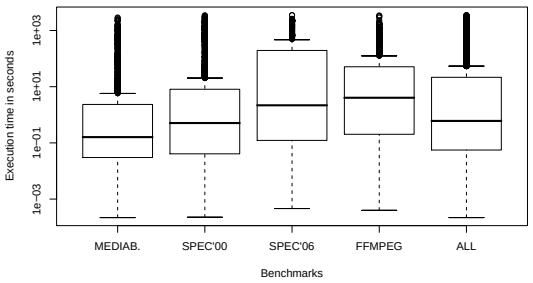
(c) type GR, no unrolling



(d) type GR, unrolling = 4×

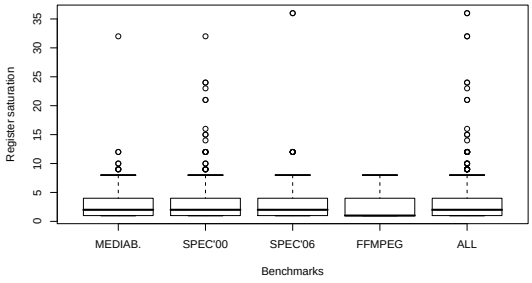


(e) type BR, no unrolling

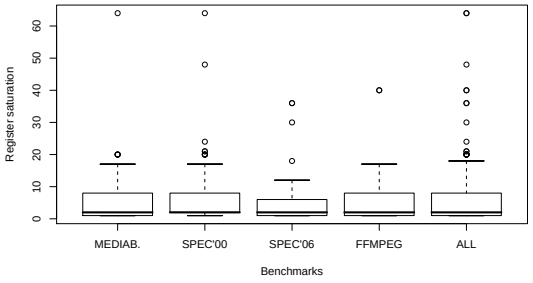


(f) type BR, unrolling = 4×

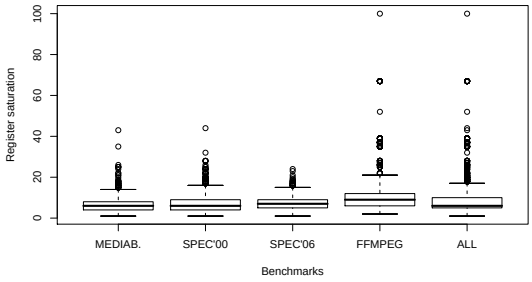
Figure 3.2: Execution times of exponential algorithm when it succeeded in less than one hour



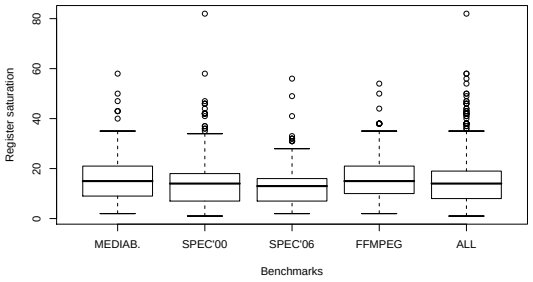
(a) type FP, no unrolling



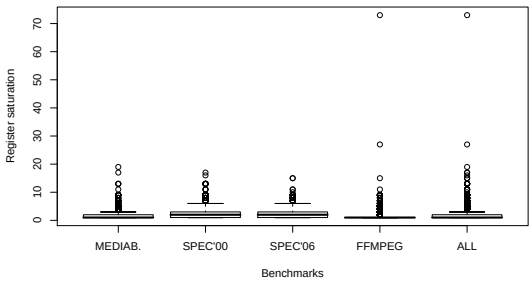
(b) type FP, unrolling = 4×



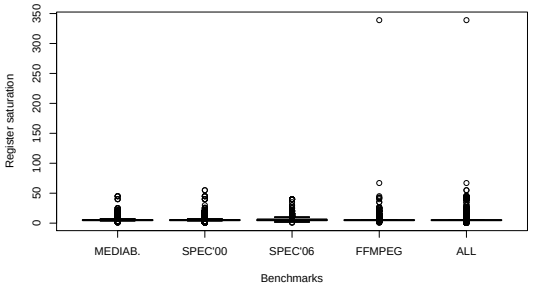
(c) type GR, no unrolling



(d) type GR, unrolling = 4×



(e) type BR, no unrolling



(f) type BR, unrolling = 4×

Figure 3.3: Distribution of saturation as computed by the exponential algorithm in less than one hour

3.3 Variants 1 and 2 heuristics versus greedy-k

In order to demonstrate that variants 1 and 2 have better experimental efficiency than greedy-k, we use the statistical test of student as explained in [2] using the `R` software [1]. For the test of student, we consider all register types, all benchmarks family, yielding to a sample data size equal to 18232. The confidence level is configured to be equal to 95%.

Regarding the computed register saturation of all register types and all benchmarks family, we conduct a one-sided test of student on the paired observations, and we can empirically ensure with 95% confidence level that $RS_0^t(G) \leq RS_1^t(G) \leq RS_2^t(G)$. Since by definition we have $RS_0^t(G) \leq RS^t(G)$ and $RS_1^t(G) \leq RS^t(G)$, this means clearly that variant 1 heuristic has better approximation of RS than greedy-k heuristic. Regarding variant 2, it is not guaranteed by definition that $RS_2^t(G) \leq RS^t(G)$. In practice, we did not observ any case where $RS_2^t(G) > RS^t(G)$. So hence we can empirically conclude that variant 2 heuristic is a better approximation for RS than greedy-k heuristic.

Regarding the execution times needed to compute the register saturation of all register types and all benchmarks family, we must recall that the execution time of a program may vary from one run to another (even with the same data input). Consequently, for the rigour of the statistics, we must use the impaired version of the test of student. We observe that:

1. greedy-k and variant 1 heuristic have equivalent average execution time. This has been checked with a two-sided test of student with a confidence level equal to 95%.

2. Variant 2 heuristic is faster in average than greedy-k and variant 1. This has been checked with a one-sided test of student with a confidence level equal to 95%.

This section provided a qualitative statistical analysis on the efficiency of variants 1 and 2. The next section provides a quantitative analysis.

3.4 Accuracy of the approximations of the heuristics

3.4.1 Numerical results

In order to quantify the accuracy of the two versions of the greedy-k heuristic, we compare the result computed by each heuristic to the value computed by the exponential (optimal) algorithm. For a given heuristic, we thus count the number of cases where the returned value is lesser than (LT) or equal (EQ) to the exact register saturation. While variant 1 heuristic is guaranteed to not over-estimate RS, variant 2 heuristic may in theory over-estimate RS. In practice, such situation didn't hold, so we do not present the case GT.

Results are shown on Figure 3.4 for the precise version of the heuristic and on Figure 3.5 for the approximated one.

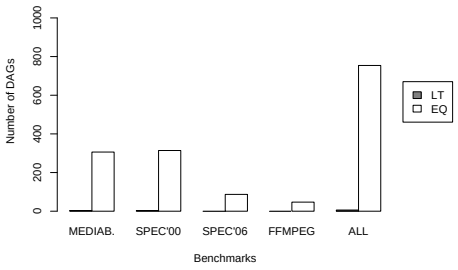
Furthermore, we estimate the error ratio of each of the two heuristics with the formula $1 - \frac{\sum RS_i^t(G)}{\sum RS^t(G)}$ for $t \in \mathcal{T}$ and $i \in \{1, 2\}$. This is actually the mean error ratio. $RS_i^t(G)$ denotes the RS computed by variant i.

These ratios are shown on Figure 3.6 for the precise heuristic (variant 1) and on Figure 3.7 for the approximated heuristic (variant 2).

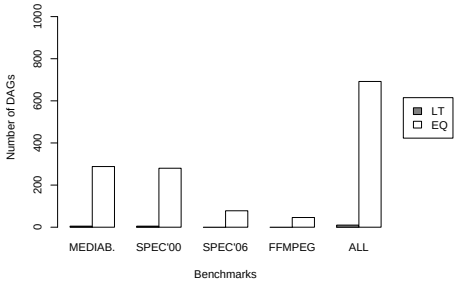
3.4.2 Comments and analysis

These experiments show that both heuristics are good for approximating register saturation of DAGs. Indeed, in most of the cases, both heuristics found the exact value of the saturation.

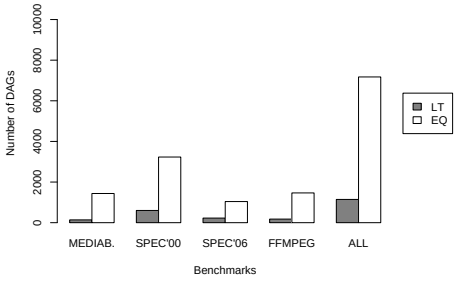
However, for the particular case of bodies of loops unrolled four times, GR saturation was underestimated in more than half of the cases as seen on Figure 3.4(d) and Figure 3.5(d). To balance this, we have first to remind from Figure 3.1(d) that exact GR saturation was unavailable for more than half of the DAGs, hence the size of the sample is clearly smaller than for the other statistics. Secondly, as seen on Figure 3.6 and Figure 3.7, the error ratio remains low, since it is lower than 12-13% in the worst cases.



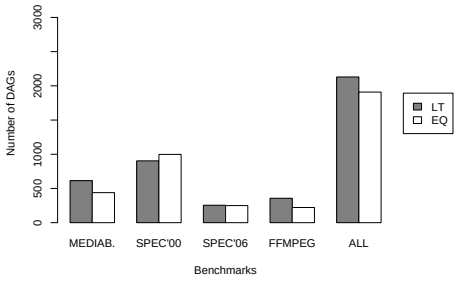
(a) type FP, no unrolling



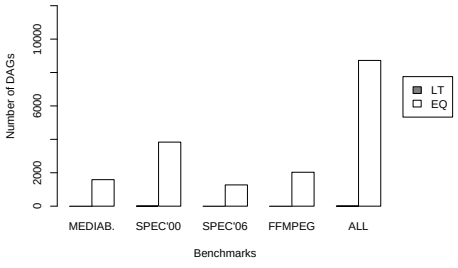
(b) type FP, unrolling = 4×



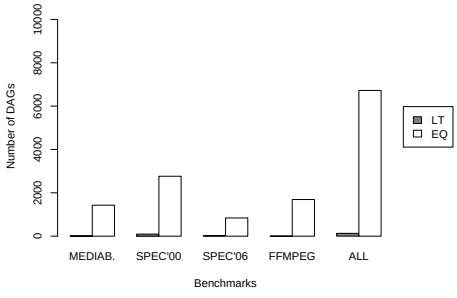
(c) type GR, no unrolling



(d) type GR, unrolling = 4×

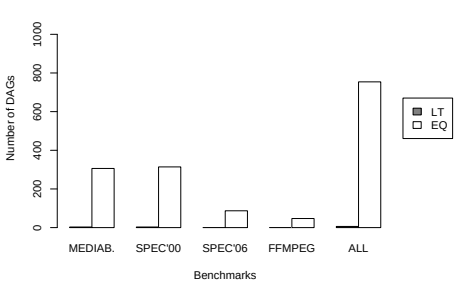


(e) type BR, no unrolling

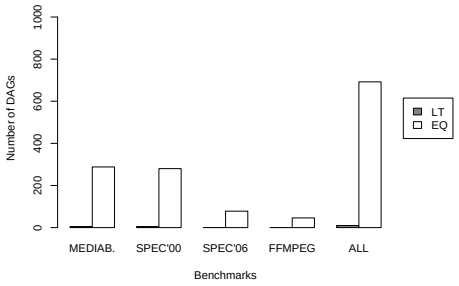


(f) type BR, unrolling = 4×

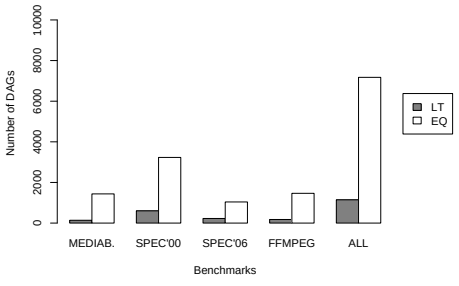
Figure 3.4: Accuracy of the precise heuristic (variant 1)



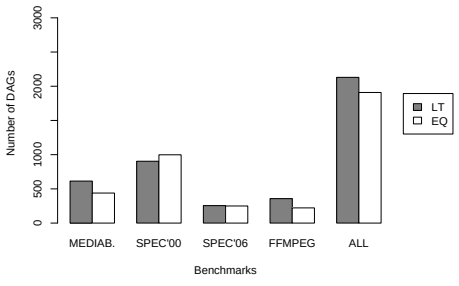
(a) type FP, no unrolling



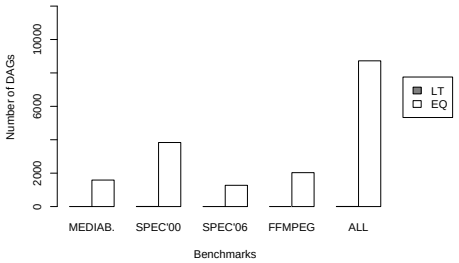
(b) type FP, unrolling = 4×



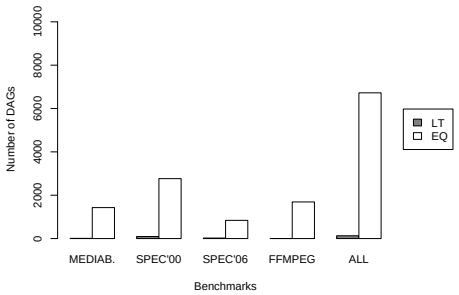
(c) type GR, no unrolling



(d) type GR, unrolling = 4×

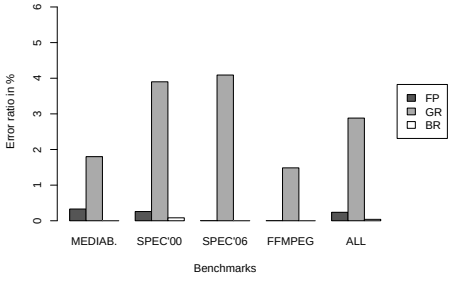


(e) type BR, no unrolling

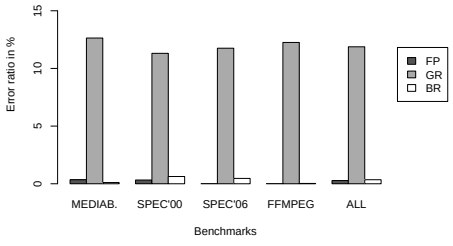


(f) type BR, unrolling = 4×

Figure 3.5: Accuracy of the approximated heuristic (variant 2)

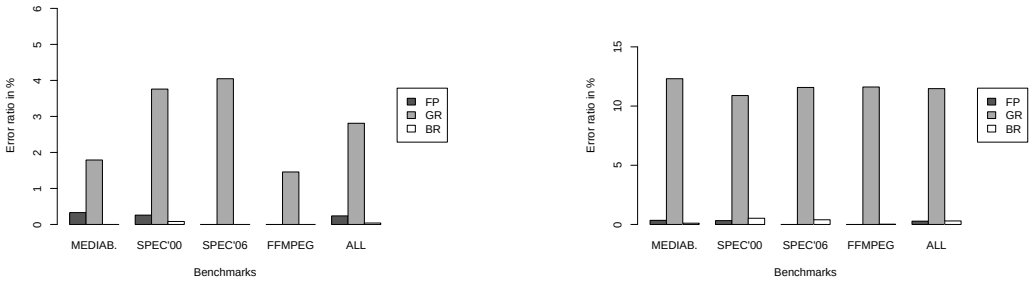


(a) no unrolling



(b) unrolling = 4×

Figure 3.6: Error ratio of the precise heuristic (variant 1)



(a) no unrolling

(b) unrolling = 4×

Figure 3.7: Error ratio of the approximated heuristic (variant 2)

Finally, quite surprisingly, we note that the two versions of the heuristics give similar results. Actually, approximated heuristic produces even better results than the precise one.

3.5 Comparison of the heuristics execution times

3.5.1 Experiments results

Since the two versions of the heuristics give equally good results in practice, the discriminating criteria becomes execution times.

Figure 3.8 and Figure 3.9 show the distribution of the execution times of the two heuristics using boxplots.

We also measure the gain factor of the approximated (supposedly faster) heuristic compared to the precise heuristic, given by the formula $\frac{\sum time_1^t(G)}{\sum time_2^t(G)}$ where $time_i^t(G)$ is the time spent by the heuristic i to estimate register saturation of type t of DAG G.

These ratios are shown on Figure 3.10.

3.5.2 Comments and analysis

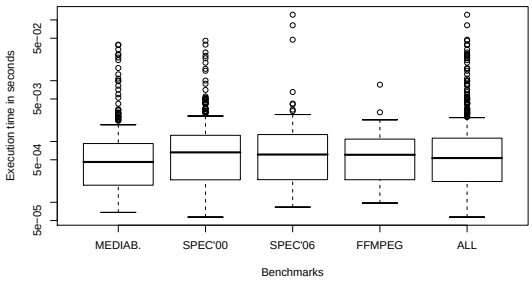
Approximated heuristic (variant 1) is clearly faster than precise one. It is more lucrative on big DAGs. The gain factor indeed raises from 25% for the original bodies to more than 90% for the bodies of the loops unrolled four times.

Regarding the execution times, we note that both heuristics are reasonably fast. Indeed, approximated heuristic only took at most 1 ms to compute saturation of 75% of the original loop bodies. For bodies of loops unrolled four times, it took not more than 20 ms in half of the cases.

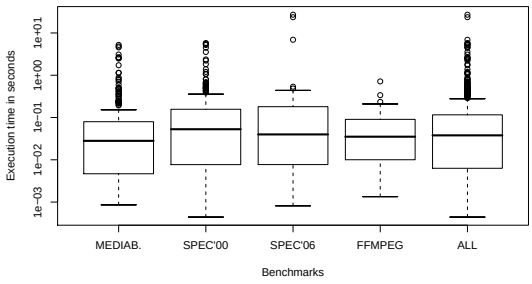
3.6 Analysing the impact of saturating schedules on the longest paths of the DAGs

This section analyses the relationship between saturating schedules and best (shortest) schedules. Understanding this relationship give us intuition about the relationship between maximal register requirement and instruction level parallelism. The best schedules are measured as the longest paths of the DAG.

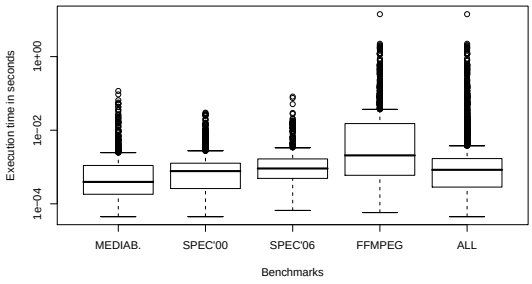
We measure the increase of the longest paths due to saturating schedules. This increase is given by the formula $\frac{\sum_G lp(G_s^t)}{\sum_G lp(G)} - 1$ (for $t \in \mathcal{T}$) where $lp(G)$ is the length of the longest path in G. G_s^t is the DAG constructed from G such as any schedule of G_s^t is a saturating schedule: Algorithm 1 from [4]



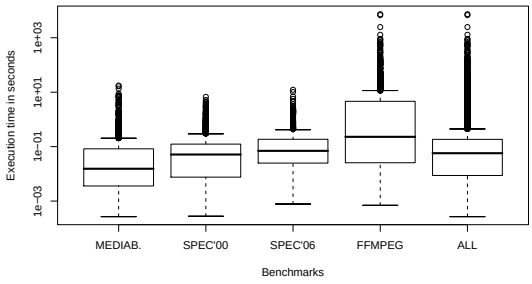
(a) type FP, no unrolling



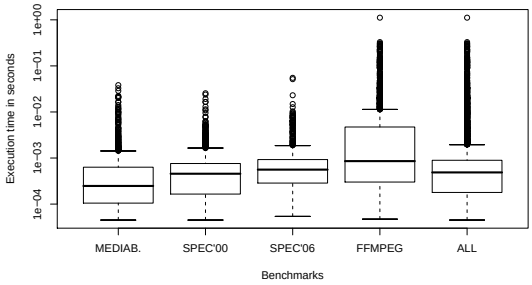
(b) type FP, unrolling = 4×



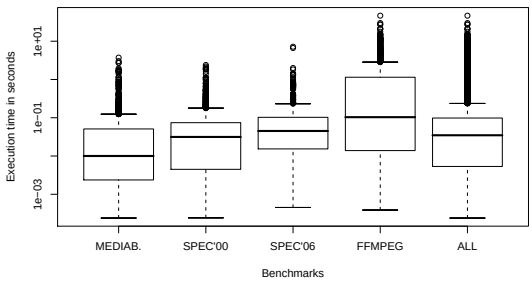
(c) type GR, no unrolling



(d) type GR, unrolling = 4×

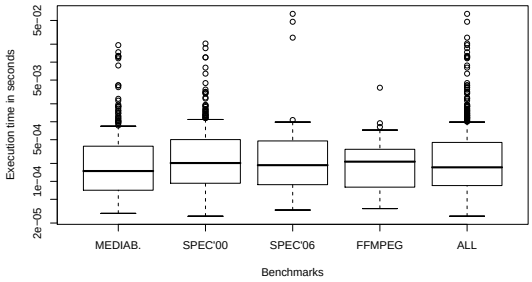


(e) type BR, no unrolling

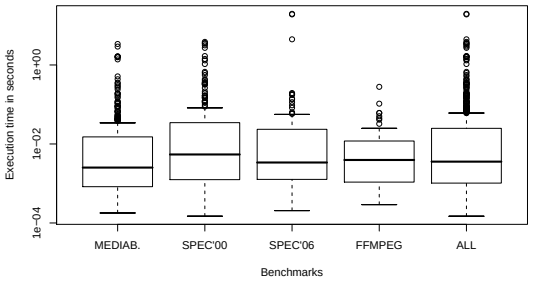


(f) type BR, unrolling = 4×

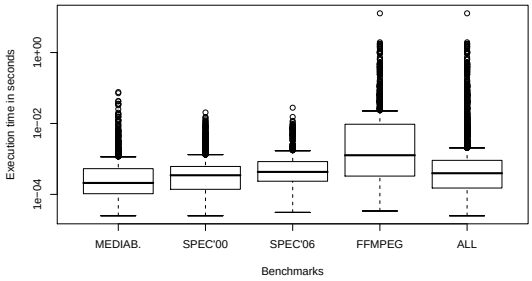
Figure 3.8: Execution times of the precise heuristic (variant 1)



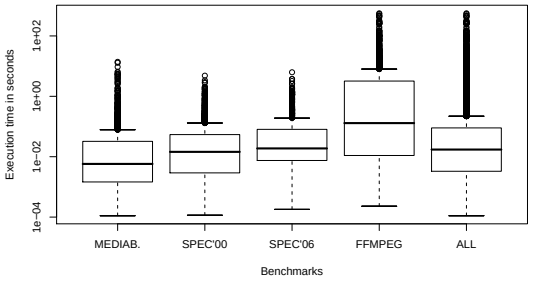
(a) type FP, no unrolling



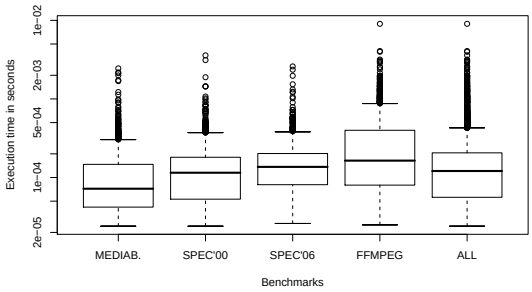
(b) type FP, unrolling = 4×



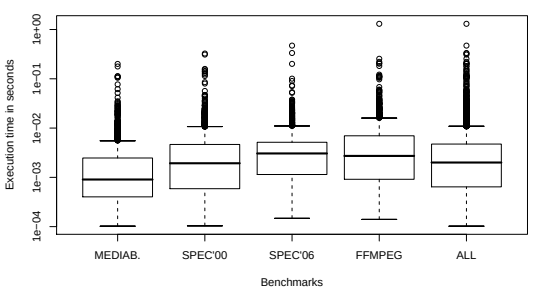
(c) type GR, no unrolling



(d) type GR, unrolling = 4×

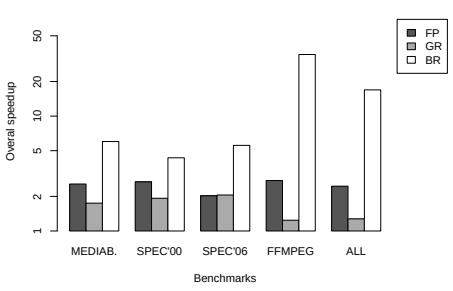


(e) type BR, no unrolling

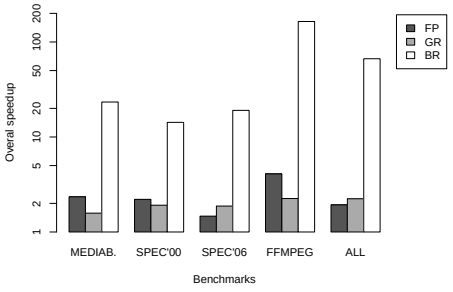


(f) type BR, unrolling = 4×

Figure 3.9: Execution times of the approximated heuristic (variant 1)



(a) no unrolling

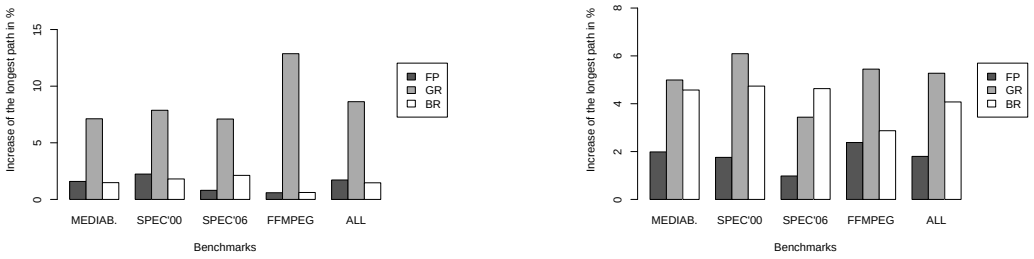


(b) unrolling = 4×

Figure 3.10: Overall speedup of the approximated heuristic vs. the precise one

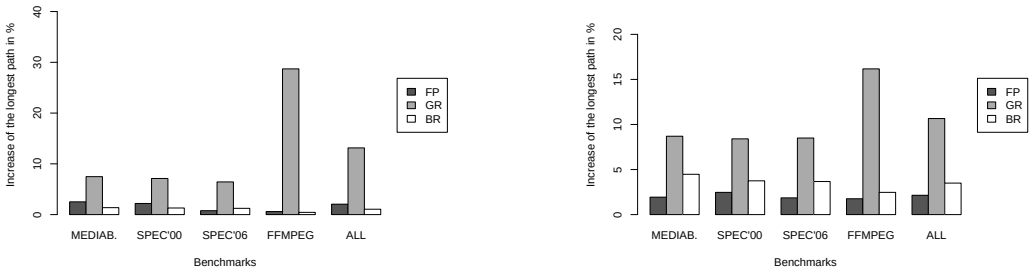
computes G_s^t. Depending on the algorithm used for computing the saturating killing function, G_s^t may be associated to the optimal register saturation or to an approximate register saturation.

Results are shown on Figure 3.11 for the optimal exponential algorithm and on Figure 3.12 for the precise heuristic (variant 1). Note that the approximated heuristic (variant 2) does not compute saturating values but only the register saturation only, so it cannot be used in this context.



(a) no unrolling (b) unrolling = 4×

Figure 3.11: Increase of the longest path (RS computed by optimal exponential algorithm)



(a) no unrolling (b) unrolling = 4×

Figure 3.12: Increase of the longest path (RS computed by variant 1 heuristic)

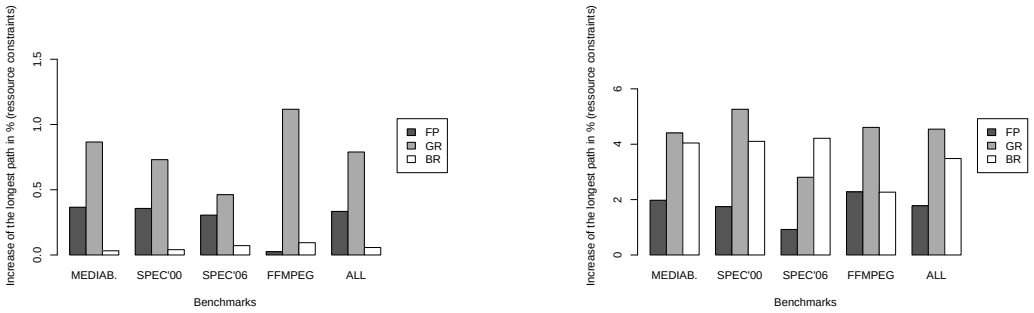
We see that the critical path increase is almost negligible for register types FP and BR. On the contrary, for type GR, the mean increase go up to about 30%, which is quite high. From Figures 3.11 and 3.12, we can say that maximising the register requirement does not necessarily minimise critical paths. In other words, this means that an agressive instruction scheduling strategy does not necessarily maximise the register requirement.

3.7 Analysing the impact of saturating schedules on the shortest possible schedules

We also measured the shortest possible schedules by considering not only the critical paths, but also by considering the resources constraint of the target architecture defined in Section 3.1.1 (number of simultaneously available functional units and instruction issue width, etc). In this case, the shortest possible schedule is the classical maximum value between the longest path of the DAG and the resource capacities[3]. Figure 3.13 and 3.14 show that the increase of the best possible schedule does not exceed

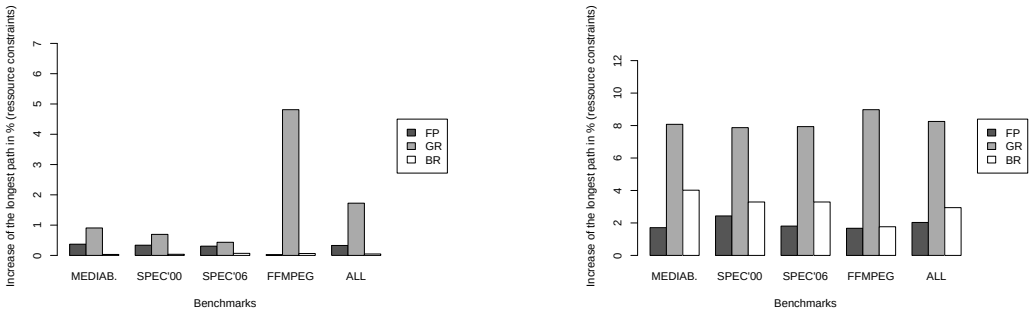
[3]The shortest possible schedules are not necessarily equal to the optimal schedules. Computing optimal schedules satisfying both resource and data dependence constraints in an NP-problem.

12%. As deduced from the previous section, these figures clearly show that maximal register requirement is not necessarily coupled with best possible schedules, and vice-versa.



(a) no unrolling (b) unrolling = 4×

Figure 3.13: Increase of the best possible schedules (RS computed by optimal exponential algorithm)



(a) no unrolling (b) unrolling = 4×

Figure 3.14: Increase of the best possible schedule (RS computed by variant 1 heuristic)

Conclusion

In this report, we have experimentally studied register saturation of directed acyclic graphs (basic blocks and super-blocks DDG).

To make these experimentations, we have implemented two new heuristics improving on the greedy-k heuristic of [4]. To estimate accuracy of these heuristics, we have also implemented an optimal exponential algorithm that solves the exact problem which has been proven to be NP complete.

Experiments, led over a large set of public and industrial benchmarks (MEDIABENCH, FFMPEG, SPEC2000, SPEC2006), have shown that both heuristics are nearly optimal. Indeed, in most of the cases, register saturation was estimated correctly for any register type (FP, GR or BR). We have measured the mean error ratio to be under 4%. If we enlarge the codes by loop unrolling ($\times 4$, multiplying the size by a factor of 5), then the mean error ratio of register saturation estimation reaches 13% at.

These experiments have also permitted us to establish that our approximated version (variant 2) of the greedy-k heuristic gives good results in practice. Thus it estimates register saturation as well as the original greedy-k heuristic while being much faster. Depending on the benchmark and on the register type, the new heuristic is up to $50\times$ faster. We should notice that the variant 2 heuristic has the drawback of not computing the set of saturating values, contrary to variant 1 heuristic. However, this is not problematic when the most important information to be exploited by the compiler is the register saturation itself. The fast nature of the variant 2 heuristic compared to the variant 1 may open the opportunity to use it inside just-in-time compilers.

Another set of experimentations concerns the relationship between register saturation and shortest possible instruction schedules. We find that, experimentally, minimal instruction scheduling does not necessarily correlate with maximal register requirement, and vice-versa. This relationship is not a surprise, since we already know that aggressive instruction scheduling strategies does not necessarily increase the register requirement.

Register saturation is indeed an interesting information that can be exploited by optimising compilers before enabling aggressive instruction scheduling algorithms. The fact that our heuristics do not compute optimal RS values is not problematic, because we have shown that best instruction scheduling does not necessarily maximise the register requirement. Consequently, if an optimal RS is under-evaluated by one of our heuristics, and if the compilation flow allow an aggressive instruction scheduling without worrying about register pressure, than it is highly improbable that the register requirement would be maximised. This may compensate the error made by the RS evaluation heuristic.

Bibliography

[1] 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.

[2] 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.

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

[4] Sid-Ahmed-Ali Touati. Register Saturation in Instruction Level Parallelism. *International Journal of Parallel Programming*, 33(4), August 2005.

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