Studying Optimal Spilling in the Light of SSA

QUENTIN COLOMBET, LIP, CNRS, INRIA, ENS-Lyon, UCB-Lyon

FLORIAN BRANDNER, ENSTA ParisTech

ALAIN DARTE, LIP, CNRS, INRIA, ENS-Lyon, UCB-Lyon

Recent developments in register allocation, mostly linked to static single assignment (SSA) form, have shown the benefits of decoupling the problem in two phases: a first *spilling* phase places load and store instructions so that the register pressure at all program points is small enough, and a second *assignment* and *coalescing* phase maps the variables to physical registers and reduces the number of move instructions among registers. This article focuses on the first phase, for which many open questions remain: in particular, we study the notion of optimal spilling (what can be expressed?) and the impact of SSA form (does it help?).

To identify the important features for optimal spilling on load-store architectures, we develop a new integer linear programming formulation, more accurate and expressive than previous approaches. Among other features, we can express SSA ϕ -functions, memory-to-memory copies, and the fact that a value can be stored simultaneously in a register and in memory. Based on this formulation, we present a thorough analysis of the results obtained for the SPECINT 2000 and EEMBC 1.1 benchmarks, from which we draw, among others, the following conclusions: (1) rematerialization is extremely important; (2) SSA complicates the formulation of optimal spilling, especially because of memory coalescing when the code is not in conventional SSA (CSSA); (3) microarchitectural features are significant and thus have to be accounted for; and (4) significant savings can be obtained in terms of static spill costs, cache miss rates, and dynamic instruction counts.

Categories and Subject Descriptors: D.3.4 [Programming Languages]: Processors—Code generation

General Terms: Algorithms, Experimentation, Performance, Theory

Additional Key Words and Phrases: Register allocation, spilling, static single assignment form, performance models

ACM Reference Format:

Quentin Colombet, Florian Brandner, and Alain Darté. 2014. Studying optimal spilling in the light of SSA. ACM Trans. Architect. Code Optim. 11, 4, Article 47 (December 2014), 26 pages.
DOI: <http://dx.doi.org/10.1145/2685392>

1. INTRODUCTION

Register allocation, a key optimization in the back end of a compiler [Torczon and Cooper 2011], consists of mapping the unlimited set of variables used in a low-level

Parts of this work were published at CASES 2011. This article contains more detailed discussions, more examples illustrating new concepts and existing approaches, and additional experiments covering the observed worst-case behavior, a new postlatency heuristic, and empiric evidence showing why static spill costs are a poor metric. Three configurations were added: Appel and George under SSA, Koes and Goldstein, and the heuristic of Braun and Hack. This work was partly supported by a contract with STMicroelectronics.

Authors' addresses: Q. Colombet, Compsys, LIP, UMR 5668 CNRS, INRIA, ENS-Lyon, UCB-Lyon; email: quentin.colombet@ens-lyon.org; F. Brandner, Computer Science and System Engineering Department, U2IS, ENSTA ParisTech; email: brandner@ensta.fr; A. Darté, Compsys, LIP, UMR 5668 CNRS, INRIA, ENS-Lyon, UCB-Lyon; email: alain.darte@ens-lyon.fr.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2014 ACM 1544-3566/2014/12-ART47 \$15.00

DOI: <http://dx.doi.org/10.1145/2685392>

program representation to the limited number of registers available in the target architecture. When not all variables can be mapped to registers, some are stored in memory to reduce register demand. This eviction to memory is called *spilling*. Memory transfers are costly in execution time, power dissipation, and code size; a good register allocator consequently should reduce spilling in order to preserve the gains of previous optimizations. For load-store architectures, memory transfers are performed through dedicated load and store instructions. The spilling phase of a register allocator consequently seeks the *optimal* placement of these instructions.

Until recently, compilers performed register allocation using variants of graph coloring, as developed by Chaitin et al. [1981]. This method gives fairly good results in practice but has three main weaknesses. First, it relies on an NP-complete problem (graph coloring) to decide which variables are spilled. Second, unless *live-range splitting* is used, evicted variables are *spilled everywhere*: store (resp. load) instructions are inserted just after (resp. before) all variable definitions (resp. uses). Third, a variable that is not spilled is kept in the same register throughout its whole lifetime. These spilling and register assignment restrictions induce suboptimal placements.

In the past few years, some researchers proposed to decompose register allocation in two phases [Appel and George 2001; Hack and Goos 2006; Bouchez et al. 2006]. The first phase decides where to place load and store instructions so that, at every program point, the register pressure (the number of live variables) is at most k , the number of available registers. A second phase assigns the variables to registers, with no additional spilling. For that to be possible, register-to-register copies (move instructions) may need to be inserted. The underlying assumption that makes such a decoupling efficient is that move instructions are more likely to be cheaper than memory transfers. Decoupled register allocation is often associated with static single assignment (SSA) form as, in strict SSA, live ranges are explicitly split in such a way that the second phase is always feasible. Koes and Goldstein [2009] empirically confirmed, using an optimal register allocation framework, the effectiveness of this decoupled strategy. We push this analysis even further by exploring the first spilling phase in more detail.

Our initial motivation was to analyze whether SSA is helpful or not to achieve good spilling. SSA may indeed appear attractive for the design of spilling algorithms, because the underlying dominance tree often simplifies algorithms [Braun and Hack 2009]. Also, a spill-everywhere strategy can be realized by finding a maximal k -colorable subgraph in the interference graph, which is chordal in SSA. Although NP-complete [Yannakakis and Gavril 1987; Bouchez et al. 2007b] (if k is not fixed), this problem may, in practice, appear simpler than for a general graph. However, considering the different SSA live ranges, obtained from a given non-SSA variable, as unrelated means that stores may be needed for each spilled live range, while only one might be enough for the original variable. This may increase the spill cost considerably, unless the moves hidden in the SSA ϕ -functions are exploited.

To analyze these choices, and not just through heuristics, we needed an exact formulation of the spilling problem, as complete as possible, that exploits the structure of decoupled register allocation. As we show in Section 2, previous formulations either express the whole register allocation problem and are very expensive, or cannot express all solutions due to some simplifying assumptions, in particular the fact that a variable cannot be stored simultaneously in a register and in memory. We developed a new integer linear programming (ILP) formulation to approach optimality even closer and to better understand the mechanisms involved during spilling. Section 3 first presents a simplified version of this formulation, which already subsumes most previous approaches, and then shows extensions that incorporate more advanced features. Section 4 gives a thorough analysis of the results obtained for the SPECINT 2000 and

EEMBC 1.1 benchmarks and discusses the important features for optimal spilling on load-store architectures.

To summarize, our contributions are:

- A flexible and expressive ILP formulation for spilling on load-store architectures, which accurately models variable liveness, rematerialization, SSA and move instructions, memory coalescing, and so forth.
- A detailed analysis that shows (1) the importance of rematerialization; (2) the complexity induced by modeling SSA and memory coalescing to exploit ϕ -functions and move instructions; (3) the importance of microarchitectural features, which render static spill costs unreliable indicators of actual runtime performance; and (4) that still considerable gains are possible wrt. instruction counts and cache miss rates.

2. FORMULATING “OPTIMAL” SPILLING

“Optimal” spilling formulations are based on the notion of *program points* and *local* register pressure, capturing the number of live variables and their assignment to either memory or registers at a given point. Since spilling is a global problem, program points are connected according to the control-flow graph (CFG) so that decisions at one point impose constraints at its neighbors in the CFG.

2.1. Existing “Exact” Formulations

The ILP formulation of Goodwin and Wilken [1996] models the complete register allocation problem, including the assignment of registers, using *live-range graphs* (LRGs). An LRG models the assignment of a variable to a specific hardware register and needs to be instantiated for *every* register. The initial LRGs are extended to capture spilling along the variable’s live range, that is, stores, loads, register copies, and rematerialization [Briggs et al. 1992]. A major drawback is the large size of the ILP instances, due to the duplication of the LRGs and redundancies arising from the LRG extensions. The approach appears rather expensive in practice. However, an optimized variant later addressed some of those issues [Fu and Wilken 2002]. Falk et al. [2011] account for spill code in the cost function modeling the worst-case execution time of real-time programs, without changing the formulation’s expressiveness.

Appel and George [2001] were the first to exploit the decoupling between spilling and register assignment by replacing the latter by a simpler constraint on *register pressure*. Developed for CISC machines, they demonstrated that this strategy considerably reduces the size of the ILP instances. However, they made a fundamental (and surprising) assumption: a variable *cannot* be stored simultaneously in memory and in a register. The problem can then be simplified by expressing, for each program point, the possible movements of a variable between memory and the set of registers. This limitation leads to suboptimal code, in particular to redundant store instructions.

The approach of Koes and Goldstein [2006] is based on *multicommodity network flows*. All live ranges are expressed using a single network-flow problem, where variables are represented by source and sink nodes, while other nodes represent allocation classes, such as constants, registers, and memory, at program points and instructions. Network capacities constrain the number of variables that can be assigned to the storage classes. Initially designed to solve spilling and register assignment, the approach can also be used to express the spilling problem alone, constraining register pressure by merging nodes and summing their associated capacities [Koes and Goldstein 2009]. Using so-called *antivariabes*, the insertion of redundant stores is avoided. However, a variable may only be assigned to a single allocation class at any given program point.

Ebner et al. [2009] address the spilling problem for SSA programs using a series of network-flow problems, one for each variable. Nodes correspond to instructions and

<pre> a = ... b = a + 1 ⚡ ... = a </pre>	<pre> a = ... b = a + 1 ⚡ @a = store a ... a = load @a = a </pre>	<pre> a = ... @a = store a b = a + 1 ⚡ // a dies in register ... a = load @a = a </pre>
(a) Before spilling	(b) Ineffective spilling	(c) Effective spilling

Fig. 1. Spilling variable *a* does not help, unless it can be kept in a register and in memory at the same time.

<pre> a = ... while(...){ ⚡ = a } </pre>	<pre> a = ... @a = store a while(...){ ⚡ a = load @a = a @a = store a } </pre>	<pre> a = ... while(...){ @a = store a ⚡ a = load @a = a } </pre>	<pre> a = ... @a = store a while(...){ ⚡ a = load @a = a } </pre>
(a) Before spilling	(b) Appel 1	(c) Appel 2	(d) Optimal spilling

Fig. 2. A load causes the value in memory to die and thus induces spurious stores, in particular in loops.

edges to program points where loads can be placed. Every cut of such a network gives a solution to the spilling problem for that particular variable. Nodes representing the same instruction of the various flow problems are assembled into partitions with capacities to capture the register pressure. Stores are not optimized and inserted after the unique definition of a variable. Furthermore, the splitting of live ranges due to SSA is kept unchanged; that is, the implicit moves of ϕ -functions are not exploited.

2.2. Limitations of Existing Approaches

The approaches presented earlier were designed to solve register allocation and spilling under various constraints and assumptions stemming from complexity or modeling considerations. They often show slight limitations concerning “optimality,” expressiveness, and even correctness. In the following, the symbol ⚡ indicates a program point where the register pressure is too high and some variable needs to be spilled.

2.2.1. Liveness. The extent of live ranges is a surprising source of problems. If a variable cannot be simultaneously in register and memory, as for Appel and George and Koes and Goldstein, a variable has to remain in a register *after* a use until the value can be spilled, unless it dies at that use. Variable *a* in Figure 1(a) cannot be spilled before its first use and has to remain in a register until after the definition of *b*. It always interferes with *b* as shown in Figure 1(b). The problem remains, regardless of the spilling decision. A better solution (Figure 1(c)) is to store *a* right after its definition and to keep it in a register and in memory until its first use. In the worst case, these artificial interferences between uses and definitions may render the spilling problem unfeasible, for example, when the number of variables defined and used is larger than the number of available registers.

A similar problem can arise with the initial formulation of Goodwin and Wilken [1996], linked to the *block start* and *block end* transformations. Later results resolve this issue with additional ILP variables explicitly modeling *deallocation* [Fu and Wilken 2002].

2.2.2. Living in Memory and Register Simultaneously. A major limitation of the approach of Appel and George is the assumption that a variable may only be kept in memory or in a register, but never in both at the same time. Besides the extension of live ranges shown previously, this leads to spurious store operations as shown by Figure 2. If *a* is

Fig. 3. Even when redundant stores cost zero, suboptimal spilling solutions may be generated.

to be spilled, a load needs to be placed inside the loop. This load “destroys” the spilled value in memory and forces the placement of a (useless) store operation inside the loop (Figure 2(b)). The “optimal” solution in their model is to spill only inside of the loop (Figure 2(c)). However, the best solution, shown in Figure 2(d), cannot be expressed: a is spilled before the loop while its value is kept alive in memory after the load.

A similar example for the model of Koes and Goldstein is shown in Figure 3, despite the fact that redundant stores cost zero (shown in gray). A load is clearly needed in the code block of the `if` statement. This load induces either a store in the same `if` (Figure 3(a)), a free store and a spurious load within the loop (Figure 3(c)), or a free store within the loop and a spurious load before the loop. However, a useless load operation is required before the loop due to the use of a on the `else`-branch within the loop.

2.2.3. Rematerialization. It is well known that rematerialization [Briggs et al. 1992] has a great potential to reduce spill costs by recomputing values instead of storing and reloading them from memory. However, in the context of “optimal” spilling, rematerialization and its impact on code quality and solving times are hardly studied.

The approach of Appel and George does not address rematerialization. Koes and Goldstein model rematerialization of simple constants using a dedicated allocation class. Since variables cannot live in multiple allocation classes at the same time, rematerialization prevents values from staying available in memory. This may be needed for further usage after a CFG join point if the variable is not rematerializable on the other path. Goodwin and Wilken restrict their model to variables holding a constant value throughout their entire live range. Moreover, rematerializable variables *cannot* be spilled to memory. This is a severe limitation for non-SSA programs, since variables are often rematerializable on parts of their live ranges only [Briggs et al. 1992].

2.2.4. Memory Coalescing Under SSA Form. SSA simplifies the register assignment phase, but its benefits for spilling are less clear. An important aspect, not covered so far, is ϕ -functions—in particular the effect of spilling their result and/or arguments. Ebner et al. treat them as completely independent variables and do not exploit the implicit copy relations in their cost model. Instead, they place loads and stores a posteriori, once spilling on the SSA variables is done. The operands of the ϕ -function in Figure 4(a) are partially in registers and memory. During ϕ -function elimination, copies are inserted to ensure that the values are available in the ϕ -function’s memory location ($@a$). One operand is only available in memory and thus requires a memory-to-memory copy (a load and a store in Figure 4(b)). The register pressure is locally increased, which requires additional repair code to spill and restore some temporary register *cross* (Figure 4(c)).

Fig. 4. Removing the ϕ -function in (a) induces a load and a store (b), which locally increases register pressure and causes additional spill code (c).

Spilling heuristics [Braun and Hack 2009] usually avoid the problem by requiring the program to be in conventional SSA (CSSA), where the operands and the definition of a ϕ -function do not interfere. In this case, the related variables can be stored at the same memory location, without the need for additional memory operations. For programs that do not exhibit this property, the copy relations and coalescing of memory locations among ϕ -operands have to be modeled to derive an accurate cost.

3. A “MORE OPTIMAL” FORMULATION

This section presents a new ILP spilling formulation for load-store architectures, more accurate than previous solutions and flexible enough to evaluate tradeoffs when designing spilling strategies. It can also *emulate* the formulations from Section 2 with a few additional constraints. We first present a simplified version for non-SSA programs, then describe extensions to handle moves, in particular those implicit in SSA.

Given a program represented by a CFG, with weights indicating the execution frequency of basic blocks, we seek the cost-optimal placement of stores and loads. No other modifications are performed (e.g., no rescheduling). The spill operations can be placed on *program points* before and after instructions. Additional program points are available at CFG joins and splits, depending on whether CFG edges are split. An optimal solution might require multiple spill operations at a program point. Without loss of optimality, we choose to perform all stores first, then all loads. The relative order of the individual stores (respectively loads) is not relevant and not modeled.

3.1. Basic Formulation

For every variable live at a program point, we record the variable’s assignment to a register and/or memory. Constraints ensure that the register pressure never exceeds the number of available registers and the needs of instructions reading/writing variables are respected. Note that a variable can die in register or memory at any moment. For a variable v live at a program point p , we introduce the following 0-1 variables.

$$\begin{aligned}
 \rho_{1,p,v} &= 1 \text{ iff } v \text{ is available in a register at the beginning of } p \\
 \rho_{2,p,v} &= 1 \text{ iff } v \text{ is available in a register at the end of } p \\
 \mu_{1,p,v} &= 1 \text{ iff } v \text{ is available in memory at the beginning of } p \\
 \mu_{2,p,v} &= 1 \text{ iff } v \text{ is available in memory at the end of } p \\
 s_{p,v} &= 1 \text{ iff } v \text{ is stored to memory at (the beginning of) } p \\
 l_{p,v} &= 1 \text{ iff } v \text{ is reloaded from memory at (the end of) } p
 \end{aligned}$$

The variables $s_{p,v}$ and $l_{p,v}$ could be deduced from the other ILP variables. Nevertheless, we keep them for readability and to simplify the specification of the ILP cost function.

3.1.1. Constraints.

Definitions and Uses. On a load-store architecture, a variable v must be in a register immediately after its definitions and before its uses. For a program point p immediately preceding an instruction using v , v must be in a register at the end of p :

$$(\text{Use}) \rho_{2,p,v} = 1.$$

Similarly, for a program point p that immediately follows an instruction defining v , v must be in a register at the beginning of p but is not available in memory:

$$(\text{Def}_R) \rho_{1,p,v} = 1 \quad (\text{Def}_M) \mu_{1,p,v} = 0.$$

Loads and Stores. To perform a load (resp. store) of v at program point p , v has to be available in memory (resp. register) at the beginning of p :

$$(\text{Load}) l_{p,v} \leq \mu_{1,p,v} \quad (\text{Store}) s_{p,v} \leq \rho_{1,p,v}.$$

The following constraints are added for convenience while preserving optimality. A load (resp. store) does assign a register (resp. memory location):

$$(\text{Load}^*) l_{p,v} \leq \rho_{2,p,v} \quad (\text{Store}^*) s_{p,v} \leq \mu_{2,p,v}.$$

Propagation. A variable v is available in a register at the end of a program point p if it was in a register (resp. memory) at the beginning of p or it has just been read from (resp. written to) memory using a load (resp. store):

$$(\text{Reg}_p) \rho_{1,p,v} + l_{p,v} \geq \rho_{2,p,v} \quad (\text{Mem}_p) \mu_{1,p,v} + s_{p,v} \geq \mu_{2,p,v}.$$

It remains to ensure the consistency between two successive program points p and q . If a variable v is not defined by the instruction between p and q , v is in register (memory) at the beginning of q if it is in register (memory) at the end of all immediately preceding program points p . Using inequalities ($\geq$) instead of equalities ($=$) allows us to release register and memory assignments at any time within and between program points:

$$(\text{Reg}_{p,q}) \rho_{2,p,v} \geq \rho_{1,q,v} \quad (\text{Mem}_{p,q}) \mu_{2,p,v} \geq \mu_{1,q,v}.$$

Register Pressure. There should be at most k variables in a register at the beginning and at the end of each program point p , where k is the number of available registers:

$$(\text{Pres}_b) \sum_v \rho_{1,p,v} \leq k \quad (\text{Pres}_e) \sum_v \rho_{2,p,v} \leq k.$$

3.1.2. Objective Function. Our goal is to minimize the expected cost of spill code at runtime (code size could also be modeled). We denote by F_p the expected execution frequency of program point p and by $C_{\text{store},v}$ and $C_{\text{load},v}$ the costs of a store and a load for variable v at p . The parameterization of the costs with p and v gives additional freedom for our advanced formulations presented later. We then aim at minimizing:

$$\sum_p \sum_v F_p (C_{\text{store},v} s_{p,v} + C_{\text{load},v} l_{p,v}).$$

3.1.3. Illustrative Example. Consider the two successive program points p and q surrounding the instruction $b = a + 1$ (Figure 5). Two constraints are generated due to the definition of variable b : $\rho_{1,q,b} = 1$ (Def_R) and $\mu_{1,q,b} = 0$ (Def_M). The use of a induces $\rho_{2,p,a} = 1$ (Use). Variable a can also be spilled or reloaded at p as indicated by the constraints $l_{p,a} \leq \mu_{1,p,a}$ (Load) and $s_{p,a} \leq \rho_{1,p,a}$ (Store). The register and memory assignment of a is propagated at q using $\rho_{1,p,a} + l_{p,a} \geq \rho_{2,p,a}$ (Reg_p) and $\mu_{1,p,a} + s_{p,a} \geq \mu_{2,p,a}$ (Mem_p), while the constraints $\rho_{1,q,a} \leq \rho_{2,p,a}$ ($\text{Reg}_{p,q}$) and $\mu_{1,q,a} \leq \mu_{2,p,a}$ ($\text{Mem}_{p,q}$) propagate the assignment between the two program points. Register pressure constraints are added and $l_{p,a}$ and $s_{p,a}$ are appended to the objective function.

	$\rho_{1,p,a} = ?$		$\mu_{1,p,a} = ?$
	+	(Load) (Store)	+
$\bullet p$	$l_{p,a} = ?$	$\swarrow \quad \searrow$	$s_{p,a} = ?$
	$\geq (\text{Reg}_p)$		$\geq (\text{Mem}_p)$
	$\rho_{2,p,a} = 1$ (Use)		$\mu_{2,p,a} = ?$
$b = a + 1$	$\geq (\text{Reg}_{p,q})$		$\geq (\text{Mem}_{p,q})$
$\rho_{1,p,b} = 1$ (Def _R)	$\mu_{1,q,b} = 0$ (Def _M)	$\rho_{1,q,a} = ?$	$\mu_{1,q,a} = ?$
	$\bullet q$		

Fig. 5. ILP variables and constraints on program points surrounding an instruction. Interrogation marks denote values that need to be derived by the ILP solver, that is, give rise to the computed spilling solution.

3.2. Emulating Other Formulations

With a few additional constraints, we can emulate other ILP approaches as well as heuristic strategies such as *spill everywhere*. We only detail here the emulation of the approaches of Appel and George and of Koes and Goldstein. These emulations are derived by overconstraining our basic formulation, which expresses more solutions.

To emulate the first one, we just need to forbid a variable from being in register and memory at the same time. This can be done with the constraint $\mu_{2,p,v} + \rho_{2,p,v} = 1$ for every program point p and variable v live at p . As for $\mu_{1,p,v} + \rho_{1,p,v} = 1$, it is implied by the propagation constraints $\text{Reg}_{p,q}$ and $\text{Mem}_{p,q}$. An alternative formulation is to force a store (resp. load) to release the corresponding register (resp. memory location):

$$(\text{Appel}_l) \ l_{p,v} + \mu_{2,p,v} \leq 1 \quad (\text{Appel}_s) \ s_{p,v} + \rho_{2,p,v} \leq 1.$$

It is interesting to note that, if we remove the Appel_l constraints and only keep the Appel_s constraints (i.e., a load does not force the variable to die in memory), we retrieve the model of Koes and Goldstein, in which the cost of a store is zero when the variable has already been stored. Actually, to get a faithful emulation, we should slightly weaken the model to express the limitation exposed in Section 2.2.1 by adding $\rho_{1,p,v} = 1$ for every program point p after a use of variable v that is not the last use.

3.3. Handling SSA and ϕ -Functions

We now explain how to extend the previous basic formulation to deal with SSA programs. Several approaches are possible depending on whether live ranges of SSA variables are considered to be unrelated or whether copy relations implicit in ϕ -functions are exploited. We refer to the former approach as *basic SSA* (see Section 3.3.1). In the latter case, the fact that arguments of a ϕ -function can interfere complicates the formulation: we then propose two solutions, an *optimistic approach* that may require repair code and optimizes an underestimation of the spill costs, and a *pessimistic approach* that conservatively exploits memory-to-memory copies. The way we handle ϕ -functions can also be used to exploit regular move operations, thanks to the notion of *local equivalence classes* that will be explained later on. Finally, we also present support for a sophisticated model for rematerialization.

3.3.1. Basic SSA. The easiest way to handle SSA programs is to consider live ranges of SSA variables as unrelated and to interpret ϕ -functions as copies between variables. The basic formulation of Section 3.1 can then be applied on the code resulting from a direct out-of-SSA translation [Sreedhar et al. 1999]. In this process, ϕ -functions are represented by parallel move operations that are implicitly placed at the program point representing a ϕ -function and its predecessors as illustrated by Figure 6. These parallel copies are then sequentialized, which may require an additional variable.

<pre> if (...) ... else ... a = $\phi(b, c)$ e = $\phi(b, d)$ </pre>	<pre> if (...) ... (a_ϕ, e_ϕ) = (b, b) else ... (a_ϕ, e_ϕ) = (c, d) (a, e) = (a_ϕ, e_ϕ) </pre>
(a) Before	(b) After

Fig. 6. Replacement of ϕ -functions.

This approach, although correct, has several weaknesses. For load-store architectures, it requires every argument of a ϕ -function to pass through a register at the corresponding copy. This may increase spilling. Also, each ϕ -function potentially induces a store if the corresponding variable is spilled. Finally, the fact that a particular sequentialization is chosen a priori may preclude opportunities. When considering SSA variables, it is thus preferable to combine spilling with a form of copy coalescing.

3.3.2. Optimistic Coalescing. It is more natural to consider a ϕ -function as a special case of the propagation rule between program points ($\text{Reg}_{p,q}$ and $\text{Mem}_{p,q}$). The registers/memory assignments of the ϕ -function arguments then need to be combined and propagated onward. The result of a ϕ -function is available in register (resp. memory) if all other arguments are in register (resp. memory). More formally, for every program point p_i , $1 \leq i \leq n$ preceding a program point q that represents a ϕ -function $a_0 = \phi(a_1, \dots, a_n)$, we add the following two constraints:

$$(\text{Phi}_R) \rho_{1,q,a_0} \leq \rho_{2,p_i,a_i} \quad (\text{Phi}_M) \mu_{1,q,a_0} \leq \mu_{2,p_i,a_i}.$$

In this approach, implicit memory-to-memory copies, expressed by the constraints Phi_M , are allowed at no cost. This model is used in the heuristic of Braun and Hack [2009] under the precondition that the program is in CSSA, which guarantees that no actual memory copies are required (how Ebner et al. [2009] capture ϕ -functions is not explained). Indeed, in CSSA, variables connected by ϕ -functions do not interfere and can be spilled to the same memory location.

The same approach can be used *optimistically* for programs that are not in CSSA by observing that memory live ranges are shorter than the original live ranges and, after spilling, are less likely to interfere than the original live ranges. After ILP solving, ϕ -functions whose results are not in a register at their definition point are converted to ϕ -functions with memory operands. The live ranges of all memory locations are then computed and coalesced using aggressive coalescing [Chaitin et al. 1981]. Finally, repair code is inserted that transfers from the memory location of a ϕ -function argument to the appropriate destination whenever the argument was not coalesced with the destination. These additional costs are not reflected in the ILP objective function, which may lead to suboptimal solutions. The copies can also increase the register pressure locally, which then requires additional spilling in the repair code.

3.3.3. Pessimistic Coalescing. The *pessimistic* approach proceeds in the opposite manner. Parallel move operations are implicitly placed at the program point representing a ϕ -function and its predecessors, as illustrated in Figure 6(a). Next, liveness is computed, an interference graph of all live ranges is built, and aggressive coalescing is used to define sets of coalescable variables. These sets are then used during the construction of the ILP problem to express memory-to-memory duplications and take their costs into account in the ILP objective function. This is expressed using two new constraints Mem_{cpy} (copy at no cost) and Mem_{dup} (duplication) detailed hereafter.

a = ...	@ _c = store a	@ _c = store a
b = ...	@ _b = store b	@ _b = store b
⚡	⚡	⚡
if (b)	b ₁ = load @ _b	b ₁ = load @ _b
⚡	if (b ₁)	if (b ₁)
c = φ(a, b)	⚡	@ _c = store b ₁
⚡	@ _c = mem_dup @ _b	⚡
	@ _c = φ(@ _c , @ _c)	@ _c = φ(@ _c , @ _c)
(a) Original	(b) Optimistic/pessimistic	(c) Optimal

Fig. 7. Optimal memory duplication placement. Variables whose name starts with @ are memory slots.

This approach is *pessimistic*, because, whenever a variable interferes with another variable in the original program, it is assumed that the spill locations of these variables will also interfere in the final program. After ILP solving, however, we might encounter that these memory locations actually do not interfere, because the variables are not kept in memory throughout their complete live ranges. Using a postprocessing, we may eliminate useless memory duplications, again by coalescing memory locations, and lower the spill cost. In contrast to the *optimistic* variant, this postprocessing is optional and not required for correctness.

3.3.4. Optimal Coalescing. None of the approaches presented in the previous sections captures the optimal solution. Since the live ranges of a and b in Figure 7(a) always interfere in memory, coalescing cannot attain the optimum (Figure 7(b)). The best solution here is to simply store the value of b₁ that is available in a register (Figure 7(c)). Optimally solving the memory coalescing problem along with the spilling problem is intractable at the moment due to the subtle semantics of φ-functions and the complexity of capturing the actual live ranges in memory, which are not known before spilling is done. The problem of expressing optimal solutions for non-CSSA programs is left open. However, we can draw a hierarchy between the different approaches compared to an optimal solution. The basic SSA approach overconstrains the program by forcing the φ-functions' operands to be in register. Clearly, this might be suboptimal in certain cases. The pessimistic approach might also yield suboptimal solutions due to its conservative choice of coalescable memory locations and the resulting overestimation in the ILP objective function. The optimal solution can still be achieved in some cases during postprocessing, that is, when spurious memory duplications are eliminated by coalescing. The pessimistic approach is guaranteed to give better solutions than the basic SSA approach. The optimistic approach, in contrast, may find solutions whose objective functions are *better* than optimal. This happens when memory locations are falsely assumed to be coalescable. Repair code then corrects this underestimation, resulting in the final, potentially suboptimal, spill code. Since this cost underestimation is implicit, the final solution can be worse than that of the basic SSA model, even if this is unlikely.

3.4. Extended Formulation

We now present an extension of the basic ILP formulation described in Section 3.1, which can be customized to express the different approaches proposed earlier by pre-defining some variables or by omitting certain constraints.

3.4.1. Handling Regular Copy Operations. As noted before, the implicit moves of SSA's φ-functions need special attention. This is particularly true for memory-related copies,

Fig. 8. Exploiting moves as special operations.

Fig. 9. Local equivalence classes (in braces) of a short program fragment.

where the impact of coalescing memory locations has to be accounted for. The same applies for regular moves, which may appear in both SSA and non-SSA programs.

Moves. Figure 8 illustrates a situation where spill code can be avoided by exploiting move operations, with either code motion or renaming. To express such an optimization, we introduce the notion of *local equivalence classes* as the set of variables, denoted $EC_{p,v}$, that carry the same value as v at program point p (these sets can be statically precomputed). This allows us to express several additional features. For example, whenever a variable v is used, we may choose to read another variable u from its equivalence class, if u is in register, or to load from the memory location of u . We may also allow the insertion of an explicit register-to-register copy between u and v . To describe the constraints more easily, we treat the original move as a virtual operation using an artificial program point. Figure 9 illustrates the handling of program points (left) and their equivalence classes (in braces). At p_1 , only variable v_1 is live, resulting in a singleton equivalence class. The original copy is treated as a virtual program point p'_1 , where v_1 and v_2 are known to hold the same value. The corresponding equivalence class is $\{v_1, v_2\}$. Due to the redefinition of v_2 ($p_3 - p_4$), this equivalence class has to be split at p_4 . Note that live ranges *may* also be extended (here v_2 at p_3).

Crossing Variables. On load-store architectures, a memory-to-memory copy requires a register, which has to be accounted for at the respective program point, unless the copy can be eliminated through coalescing. We also want to express the newly introduced rematerialization and move operations, even if some have cost 0 in our objective function. A fixed order of these operations may lead to a suboptimal solution. Nevertheless, to keep our ILP formulation practical, we chose the following static ordering: (1) store operations, (2) memory-to-memory copies, (3) load operations, (4) rematerialization operations, and finally (5) register-to-register moves. The assignment of a variable to a register may be released either at the beginning of the program point by a store or in the middle by a register-to-register move. We account for *crossing* registers in this region to ensure that the register pressure never exceeds the number of registers.

3.4.2. ILP Variables. The extended formulation introduces six new 0-1 variables for each variable v live at a program point p :

$mem_cpy_{p,v} = 1$: memory copy into memory slot of v
 $mem_dup_{p,v} = 1$: duplication into memory slot of v
 $has_mem_dup_p = 1$: at least one memory duplication at p
 $move_{p,v} = 1$: register-to-register move to v at p
 $remat_{p,v} = 1$: rematerialize v at p
 $cross_{p,v} = 1$: v still in register after the stores at p

Both memory copies and memory duplications represent memory-to-memory transfers. The difference is that memory copies are only applicable to memory locations that are assumed to be coalescable and do not incur any cost. Memory duplications, on the other hand, cause a load followed by a store and, in addition, require a register.

3.4.3. Constraints. As the changes to exploit the local equivalence classes are straightforward, we summarize them quickly and focus on the new spill operations.

Crossing. Due to the additional spill operations and the fact that memory duplications might require temporary live ranges that are not visible at the beginning or the end of the program point, we track variables crossing through the program point in a register. The propagation constraint for a variable v at program point p then becomes

$$(\text{Cross}) \quad \rho_{1,p,v} \geq cross_{p,v}.$$

Using Equivalence Classes. Instead of requiring a given variable to be in a register at a use site, it is sufficient that some variable $u \in EC_{p,v}$ is in a register ($v \in EC_{p,v}$). At a program point p preceding a use of variable v , we apply the following constraint:

$$(\text{Use}) \quad \sum_{u \in EC_{p,v}} \rho_{2,p,u} \geq 1.$$

Similarly, loads (resp. stores) can use memory locations (register) of other variables:

$$(\text{Load}) \quad l_{p,v} \leq \sum_{u \in EC_{p,v}} \mu_{1,p,u} + s_{p,u} \quad (\text{Store}) \quad s_{p,v} \leq \sum_{u \in EC_{p,v}} \rho_{1,p,u}.$$

Moves. We do not represent an explicit move between variables as an instruction but as a program point with additional constraints. Instead of forcing the operands of the move into a register using the regular Use or Def_R constraints, we indicate that the result is neither in memory nor in register. For a program point p representing an explicit move defining a variable v , this is done by the constraint

$$(\text{Def}_{move}) \quad \rho_{1,p,v} = 0, \mu_{1,p,v} = 0.$$

Since the original instruction is removed, we have to provide a way to instantiate the move, if needed. A move $v = u$ can be performed anywhere along the original live range of v , or beyond if desired, as long as u belongs to the equivalence class of v . Given the equivalence class $EC_{p,v}$, a move can be instantiated at p under the following conditions:

$$(\text{Move}) \quad move_{p,v} \leq \sum_{u \in EC_{p,v}} cross_{p,u} + l_{p,u}.$$

In contrast to other constraints, we use $cross_{p,u}$ instead of $\rho_{1,p,u}$. This expresses that moves appear after the store and memory duplication operations of the program point.

Memory Copies. Memory-to-memory copies have different implications depending on whether the related memory slots are coalescable or not. A truly optimal spilling approach would require one to solve the memory coalescing problem along with the spill code placement. This is hardly an option since coalescing, even aggressive, is NP-complete [Bouchez et al. 2007a]. The constraints presented hereafter can be used with different coalescing strategies, including integrated approaches.

Let v be a variable live on a program point p and let $CC_{p,v} \subseteq EC_{p,v}$ be the set of variables whose memory slots can be coalesced with the memory slot of v . A *memory copy* can be performed at no cost under the following condition:

$$(\text{Mem}_{\text{cpy}}) \text{mem_cpy}_{p,v} \leq \sum_{u \in CC_{p,v}} \mu_{1,p,u} + s_{p,u}.$$

A *memory duplication* can be done regardless of whether v can be coalesced with the source of the duplication as long as both are in the same equivalence class:

$$(\text{Mem}_{\text{dup}}) \text{mem_dup}_{p,v} \leq \sum_{u \in EC_{p,v}} \mu_{1,p,u} + s_{p,u}.$$

Register Pressure. As memory duplications require an additional register, we need to know whether any duplications are performed to limit the register pressure. This can be expressed through the set of variables with a potential memory duplication at p , denoted by dup_var_p . The size of this set is known statically and constant, which can be used to express whether a memory duplication is performed at p as follows:

$$|\text{dup_var}_p| \text{has_mem_dup}_p - \left(\sum_{v \in \text{dup_var}_p} \text{mem_dup}_{p,v} \right) \geq 0.$$

Using has_mem_dup_p can ensure that the register pressure is not exceeded within a program point p , even when *memory duplications* are performed:

$$(\text{Pres}_c) \sum_v \text{cross}_{p,v} + \text{has_mem_dup}_p \leq K.$$

Rematerialization. We explicitly model rematerialization using dedicated ILP variables in order to have a clear model of which values are actually live in memory (also see Section 2.2.3). This allows us to keep rematerializable values live in memory, which is particularly important when memory-to-memory copies read a rematerializable variable. Rematerializing a variable is always possible when the respective code does not require any arguments in a register (Remat). Similarly, arguments, represented as the set A to follow, can be forced into a register (Remat_A). The respective constraints for a program point p are as follows:

$$(\text{Remat}) \text{remat}_{p,v} \leq 1 \quad (\text{Remat}_A) |A| \text{remat}_{p,v} \leq \sum_{u \in A} \text{cross}_{p,u}.$$

More complex compositions of rematerialized expressions are straightforward to express for SSA programs. Since we limit our evaluation to simple rematerialization, we do not discuss these capabilities any further at this point.

Propagation. A variable v can be in register at the end of program point p if it is the result of a move, a load, or a rematerialization or if the variable crosses p in register:

$$\text{move}_{p,v} + l_{p,v} + \text{remat}_{p,v} + \text{cross}_{p,v} \geq \rho_{2,p,v}.$$

Likewise, v can only be in memory at the end of p , if its memory location was defined by a store, *memory duplication*, or *memory copy* on p , or was available before:

$$s_{p,v} + mem_dup_{p,v} + mem_cpy_{p,v} + \mu_{1,p,v} \geq \mu_{2,p,v}.$$

Finally, the constraints to propagate between two program points are unchanged. We do not use local equivalence classes there to capture the cost of register-to-register and memory-to-memory copies. It is also important to note that the constraints of the extended formulation can easily emulate our basic formulation by presetting the value of the 0-1 variables representing the new spill operations and by restricting equivalence classes. The same is true for the proposed approaches to coalesce memory locations of copy-related variables, either coming from ϕ -functions or regular copies.

4. EXPERIMENTS

We made our experiments on the *ST231* embedded processor for media applications. This is a four-way parallel VLIW architecture, supporting one memory operation per instruction bundle. It features a direct mapped cache of 32KB for instructions (64b lines) and a four-way set associative cache of 32KB for data (32b lines). Both caches are connected to a shared bus memory controller with an average latency of 120 cycles to access off-cache data. For in-cache data, the latency between the load and a use is three cycles. The pipeline is stalled automatically if this latency is violated; that is, at least three instruction bundles have to follow a load to hide the cache latency. The data cache follows a write-through strategy. A store buffer for memory writes allows grouping up to four store requests into a single bus transaction. In case of a store/load conflict in the store buffer, the store must be processed down to the memory before being reloaded.

We implemented our ILP spiller in the Open64¹ compiler of STMicroelectronics. Register allocation, and spilling, is performed in a separate back-end optimizer that comes with the production compiler. The register allocation uses a decoupled approach, where spilling is by default performed using a heuristic and assignment using graph coloring. In the following, we compare several exact and heuristic spilling approaches:

AG	Appel and George's ILP Formulation [2001]
Coloring	Heuristic using iterated register coalescing [George and Appel 1996]
Basic	Our basic formulation; see Section 3.1
SpEv	Basic formulation emulating optimal spill everywhere ²
KG	Emulation of Koes and Goldstein's ILP Formulation [2006]
BasicSSA	Naive handling of SSA; see Section 3.3.1
SpEvSSA	Emulation of optimal spill everywhere under SSA
Optimistic	Extended formulation; see Section 3.3.2
Pessimistic	Extended formulation; see Section 3.3.3
Hack	Hack's SSA-based spilling heuristic [2007]
BH	Braun and Hack's SSA-based spilling heuristic [2009]

The first five configurations were evaluated using non-SSA programs, while the others were applied to SSA programs. In both cases, critical edges were split prior to spilling. We also show results for configurations with equivalence classes enabled

¹<http://www.open64.net/>.

²Variables are spilled or in register everywhere, with stores (resp. loads) after (before) definitions (uses).

(marked by a suffix *_ec*) and with rematerialization enabled (suffix *_remat*)—both disabled by default. We used IBM CPLEX for academics, version 12.2, as ILP solver. All configurations were tested on the benchmark suites EEMBC 1.1 and SPECINT 2000, excluding the C++ program *eon*. The compiler was invoked using the *-O3* optimization level with the number of allocatable registers limited to eight (four callee-saved and four caller-saved registers). The cost model is based on basic block frequencies, which were derived either from profiling feedback or from estimates according to Ball and Larus' heuristic [1993].

The experiments investigate the solving time of our formulation, its impact on static spill costs over all benchmark programs, and the effects on the runtime behavior of the EEMBC benchmarks. Runtime measurements for SPEC are not shown, because the benchmarks are too large for our architecture's instruction cache. The programs spent up to 65% of the time waiting for the cache, rendering all runtime measurements irrelevant for spilling. We set a time limit of 1,000 seconds for all ILP configurations. To avoid the impact of random results when the optimal solution is not reached, all presented numbers refer to optimally solved instances only. To reduce the solving time, a heuristic supplies an initial solution for all ILP configurations. Hack's heuristic is used for SSA programs; otherwise, graph coloring is used. Therefore, when the solver reaches the time limit, the provided solution is at least as good as the related heuristic.

4.1. Solving Time

Our primary goal was to express the spilling problem in a simple and flexible way. Speed was not a major concern. We still restricted the placement of loads and stores, without losing optimality, to improve solving times. For all formulations where a variable can be simultaneously in register and in memory, it is sufficient to consider solutions where loads (resp. stores) are just before (resp. after) each use (resp. definition) and at the end (resp. beginning) of basic blocks [Goodwin and Wilken 1996].

After 20 seconds, whatever the formulation, 90% of the functions of EEMBC are solved optimally. For SPEC, whose functions are larger, this takes 65 seconds. SpEv is the fastest configuration to solve. After 5 seconds, 95% (resp. 90%) of the functions are solved optimally for EEMBC (resp. SPEC). As a comparison, AG solves 79% (resp. 85%) of the functions optimally in 5 seconds. After 1,000 seconds, all 656 functions of EEMBC were solved optimally, except for the Optimistic and Pessimistic configurations, which reached the time limit for two of them. For SPEC, 99% of the 5,060 functions are solved optimally when reaching the time limit, whatever the configuration. Note that we excluded from these numbers the functions that do not require spill code, that is, 221 for EEMBC and 719 for SPEC. In these cases, the solution of the heuristic is used and the ILP solver is not invoked.

Figure 10 gives an overview of the solving times for all EEMBC and SPEC benchmarks. The curves show the percentage of functions solved to optimality in a given amount of time. However, these timings may vary, depending on runtime conditions (e.g., machine load, solver options, etc.). The solving times of most configurations develop similarly, except for SpEv, which is solved the fastest for both benchmark suites. The solving time increases for larger instances having more program points.

4.2. Static Spill Cost

In this section, we compare the different approaches with respect to the *static* spill costs, following the cost model provided by Open64, where a store costs 1.25, a load 3.25, and a rematerialization 1 (multiplied by the expected execution frequency). The costs are computed from the actual spill code after clean-ups and repair code insertion. They can slightly differ from the ILP objective value. Figure 11 shows the geometric mean of the costs normalized to AG over all benchmarks by summing the costs of all functions

Fig. 10. Percentage of functions that can be solved optimally in the given amount of time for EEMBC (left) and SPEC (right). (Marks are indicative and do not represent measured points; higher is better.)

Fig. 11. Mean of static spill costs for EEMBC (left) and SPEC (right), using freq. estimates. (Lower is better.)

optimally solved by AG and the respective strategy. For the heuristic approaches, we consider all functions optimally solved by AG. In addition, the variation is depicted using the minimum and maximum.

Despite its restrictions, AG performs much better than optimal spill everywhere (SpEv), which is the worst, except for the graph coloring heuristic (Coloring), which is also spill everywhere. All other configurations based on our ILP formulation outperform AG by about 20% or more. This is mostly due to the elimination of spurious stores (recall Figure 2). For spill everywhere, the live-range splitting of SSA cannot counterbalance the costs induced by the naive modeling of ϕ -functions. SSA here increases the static spill costs by 13% for EEMBC (8% for SPEC). BasicSSA performs better than AG but increases spill costs in comparison to Basic (non-SSA), with quite a few bad cases due to the naive handling of ϕ -functions. This can also be observed with AG under SSA, which is 4% (resp. 5% for SPEC) worse than AG. Note that SSA also leads to some improvements in a few cases. This is surprising, since the naive handling of SSA constrains the solution—solutions reachable under SSA are also reachable without SSA (when rematerialization is disabled). This can be explained as follows. First, the spill code of a function may differ between the SSA and the non-SSA version, leading to differing register assignments. Since Open64 propagates information on register usage from subfunctions to call sites, this may lead to changes in register pressure around call sites and subsequent differences in spilling. Second, SSA construction performs a simple copy propagation that preserves CSSA. This can change the register pressure, and hence the global spill code. These observations are particularly obvious in Figure 12, showing the static spill costs per function normalized to AG. For readability, functions where the costs are the same for all ILP configurations (AG, SpEv, Pessimistic, BasicSSA, Basic) have been filtered ($\sim 37\%$ of the functions). The functions are sorted in increasing order by costs according to Basic. Functions having the same

Fig. 12. Static spill costs per function for both EEMBC and SPEC, using freq. estimates. (Lower is better.)

costs for Basic are then sorted according to Pessimistic, and finally SpEv, if the costs are again the same. Note that in some cases AG has to spill, while other formulations do not spill.

In contrast to the naive handling of SSA, exploiting ϕ -functions as proposed in Section 3.3 delivers good results without degradations (Optimistic/Pessimistic). Note that our current settings mostly result in CSSA programs. More aggressive transformations violating CSSA may have an adverse impact on the optimistic strategy.

Rematerialization gives remarkable improvements of more than 20% in comparison to the same configuration without, and generally more than 40% compared to AG. Good rematerialization is essential to reduce spill costs and ought to be considered accordingly. Moreover, configurations with rematerialization profit from SSA. This is particularly true for simple rematerialization strategies. SSA offers more opportunities as each rematerializable live range matches a single variable. Note that we do not exploit the full potential of our extended formulation, which could handle more general cases, to preserve comparability with the other approaches.

KG performs well and almost reaches our `Basic_ec_remat` formulation with simple rematerialization for non-SSA programs. The geometric mean of both approaches is within 4% for EEMBC (5% for SPEC). KG is able to remove most spurious stores. However, it applies rematerialization less often than `Basic_ec_remat`. For the latter, 32% of the static spill costs stem from rematerialization for EEMBC (34% for SPEC), whereas for KG it amounts only to 19% for EEMBC (24% for SPEC). This experimentally confirms the exposed limitation of Section 2.2.3.

The heuristics achieve better static spill costs for EEMBC than for SPEC. We extended Hack's heuristic to always rematerialize values instead of keeping them in registers (`Hack_remat`); that is, rematerialization is always assumed to be free. This gives good results, especially for SPEC. However, we observe some bad cases for EEMBC ($16.25\times$ without profiling feedback) because these rematerializations are counted 1 in the Open64 cost model. Braun and Hack [2009] already provide a heuristic to handle rematerialization, which is not documented in their paper. Here, we give the idea of this support. Their spilling algorithm is based on a farthest-first strategy using next-use distances. They extend the computation of next-use distances to reflect the cost of redefining a value. In other words, the more expensive a value is to reload or to rematerialize, the shorter is its next-use distance. Their next-use distance formula is shortest path to next use minus reload or rematerialization costs for that use. BH is the best among all the heuristics, also wrt. to outliers. Like Hack's original approach, it does not rely on a spill-everywhere strategy and, moreover, provides explicit control over rematerialization costs—which helps to avoid some of the bad cases.

Figure 12 allows us to relate the different approaches. Spill everywhere clearly performs the worst. Hack's heuristic performs better but follows a very similar trend. BasicSSA competes with Basic but suffers many bad cases (spikes). Pessimistic performs the best as it avoids the bad cases of BasicSSA through its dedicated handling of ϕ -functions. It additionally profits from the simple copy propagation performed during SSA construction (mentioned in the previous paragraph). This experimental ranking is slightly different than expected, since Basic should at least match the SSA-based configurations. This is explained by the simple copy propagation performed for SSA programs and the difference in register assignment propagated to callees (see earlier).

The simple restriction of either keeping a value in memory or in register gives good results as soon as spurious stores are eliminated (e.g., for KG). This gives considerably better results than spill-everywhere strategies. We conclude that spilling heuristics adopting this simplification will only experience minor degradations. The results also indicate that our strategies to handle ϕ -functions perform very well. Also note that the static spill costs follow the same trend when profiling information is used instead of frequency estimates—we do not show them here.

4.3. Dynamic Counts

Table I compares the impact of the spilling strategy on the number of dynamically executed instructions as reported by the *ST231* profiling tools when running the final assembly code. This corresponds to a sequential machine model, where every instruction completes in one cycle. The table provides the geometric means, the best and the worst cases, and the overall benchmarks normalized to AG.

The improvements seen for the static spill costs are still reflected by the dynamic execution counts and basically show the same trend with respect to the different configurations. However, the extent of the improvements is of course reduced. This is because the reported numbers include other code not related to spilling. Table I reports all loads/stores and not just those inserted during spilling. Overall, our formulations are very effective at eliminating dynamically executed instructions. The best, `Pessimistic_remat`, reduces the number of loads/stores by 20% (and up to 66%!). Even

Table I. Number of Dynamically Executed Instructions with Frequency Estimates and Profiling Feedback (Geom. Mean/Min/Max) (Lower Is Better)

Config.	Frequency Estimates						Profiling Feedback					
	# ld/st			# instr.			# ld/st			# instr.		
	G.m	Min	Max	G.m	Min	Max	G.m	Min	Max	G.m	Min	Max
AG	1	1	1	1	1	1	1	1	1	1	1	1
–_ssa	1.01	0.89	1.34	1	0.92	1.16	1.02	0.95	1.12	1.01	0.94	1.11
KG	0.84	0.34	1	0.95	0.84	1.04	0.85	0.37	1	0.94	0.85	1.01
SpEv	1.04	0.82	1.84	1	0.84	1.42	1	0.83	1.22	0.98	0.83	1.04
SpEvSSA	1.03	0.82	1.8	1	0.85	1.4	1	0.74	1.16	0.98	0.83	1.09
Basic	0.94	0.78	1.02	0.97	0.87	1.01	0.94	0.74	1	0.97	0.84	1.02
–_ec_remat	0.81	0.34	1	0.93	0.75	1	0.82	0.37	1	0.93	0.78	1.03
BasicSSA	0.95	0.78	1.12	0.98	0.86	1.1	0.95	0.74	1.1	0.98	0.84	1.17
–_remat	0.83	0.45	1.12	0.94	0.81	1.14	0.83	0.37	1.07	0.94	0.81	1.11
Pessimistic	0.93	0.78	1.02	0.98	0.87	1.1	0.94	0.73	1.02	0.96	0.83	1.02
–_remat	0.8	0.34	1	0.93	0.8	1.02	0.82	0.37	1	0.93	0.78	1.04
Optimistic	0.94	0.78	1.05	0.97	0.86	1.04	0.94	0.73	1.05	0.96	0.84	1.05
–_remat	0.81	0.34	1	0.93	0.77	1.02	0.82	0.37	1	0.92	0.8	1
Coloring	1.14	0.9	1.98	1.06	0.92	1.49	1.09	0.76	1.44	1.02	0.86	1.14
Hack	0.99	0.78	1.16	0.98	0.82	1.1	1.01	0.86	1.28	0.97	0.86	1.06
–_remat	0.83	0.34	1.1	0.92	0.75	1.03	0.85	0.37	1.1	0.92	0.77	1.06
BH	0.89	0.57	1.17	0.96	0.82	1.13	0.91	0.56	1.11	0.97	0.8	1.13

the total number of instructions is reduced by 7% (and up to 20%!). Nonetheless, Hack with our rematerialization support achieves the best result for the total number of instructions. This may seem surprising, since the static spill costs of Pessimistic were always better. However, most rematerializable values stem from constants representing base addresses of arrays. In the *ST231* family, most of these constants can be encoded directly with the operation or can be supplied as addressing mode. Consequently, most of the rematerializations emitted during spilling do not appear in the final assembly code. Note that the ILP formulation could easily be extended to handle these cases. This was not done to allow comparisons between the different formulations here. This demonstrates why static costs may be misleading and far from reality.

As a side effect, reducing the number of memory accesses reduces the traffic to and from memory. Indeed, we also measured a significant reduction of instruction and data cache misses, even though the objective functions of the ILP formulations do not model this and assume a perfect cache (i.e., no cache misses).

4.4. Execution Time Measurements

We now focus on measurable runtime effects of the different strategies, specifically for the *ST231* architecture, which, unlike the sequential model of Section 4.3, involves instruction and memory latencies, as well as instruction bundling.

The leftmost bars of Figure 13 report the geometric means of the runtimes (normalized to AG) when executing the actual programs (compiled with the various spilling strategies using frequency estimates). The gains for the static spill costs generally do not lead to equivalent runtime gains. For example, the 20% improvements in static spill costs of the configurations without rematerialization result in a moderate runtime mean gain of about 2%. Overall, we measured mean runtime improvements from 2% to 8%. Considering the best cases for individual benchmarks, we see impressive improvements that go up to about 30%. Note that the Hack_remat heuristic is again performing slightly better than our “optimal” configurations.

Analyzing the individual benchmarks, we found that the difference between the dynamic execution counts and the actual execution times are mostly due to architectural features that are not accounted for in the spilling model. As mentioned in

Fig. 13. Runtime results for EEMBC, with postprocessing and profiling feedback. (Lower is better.)

Section 4.3, this difference is not due to cache misses. However, memory latencies under a hit turned out to be highly relevant. So far, for both the static spill costs and the dynamic execution counts, we considered that load and store instructions induce a nonzero cost, irrespective of their placement within basic blocks. In practice, the runtime overhead of these instructions depends on the ability of the postpass scheduler to hide their latencies. If the scheduler succeeds, even bad spilling decisions in terms of the *number* of loads might actually perform well. This explains why the runtimes for different configurations are close to each other. The opposite effect is also possible. We observed that, due to the placement of loads and stores (see Section 4.1), the postpass scheduler of Open64, which runs after register assignment, fails to hide many latencies and fails to pack spill code nicely into bundles. This particularly impacts spill-everywhere strategies, since loads are systematically placed right before uses.

Latency Postprocessing. For purely sequential targets, the cache-hit latency could be roughly modeled using the parametric costs we presented in our formulation (see Section 3.1.2). For instance, consider point q just before a use of v . On that point, loading v would cause the maximum latency, for example, 3. Now, consider the point before the previous instruction of the use of v . At that position, loading v will reduce the load-use latency by 1, as the respective instruction will be executed before the use. This load-use distance can be computed for each program point to reflect the cache-hit latency. However, two inaccuracies lie in this model. First, the cost does not reflect the effect of other spill instructions inserted between a load and a use. Second, it assumes a fixed scheduling and does not consider that instructions and, in particular, spill instructions can be reordered during postpass scheduling. Both problems could, theoretically, be modeled in the ILP formulations, maybe even through approximations. However, this would inevitably increase the model complexity and solver times. The problem is even amplified by the instruction bundling of the *ST231* considered here.

As modeling the interaction with scheduling clearly goes beyond the scope of this article, we chose a simpler method that is applicable for all targets. Instead of accounting for the cache-hit latency in the ILP itself, we designed a heuristic, after spilling but before assignment, that moves loads up within basic blocks while respecting register pressure. The simplified pseudo-code of this heuristic is given in Algorithm 1. The heuristic traverses the basic block from top to bottom. Whenever a spill-related load

ALGORITHM 1: Postlatency optimization

```

1: procedure HIDELOADLATENCY(BasicBlock block, integer limit)
2:   for each instruction in block from top to bottom do
3:     // Check for spill-related loads
4:     if isSpillCode(instruction) and isLoad(instruction) then
5:       // Determine currently live values
6:       currentLive = getLiveVariablesAfter(block, instruction)
7:       // Estimate initial instruction latency
8:       remainingLatency = getLatencyCost(instruction) - getDynamicLoadCost (instruc-
          tion)
9:       while remainingLatency > 0 and isDefined(instruction.prev) do
10:        prevInst = instruction.prev
11:        // Check data dependencies
12:        if checkDependency(prevInst, instruction) then
13:          // Update live values assuming the instruction moves up
14:          currentLive = currentLive \ prevInst.defs
15:          currentLive = currentLive  $\cup$  prevInst.args
16:          // Check register pressure
17:          if currentLive.size()  $\leq$  limit then
18:            block.moveInstructionUp(instruction)
19:            remainingLatency = remainingLatency - eatenLatency (prevInst)
20:          else
21:            break

```

instruction is encountered (l. 4), the number of live values is computed (l. 6) and the instruction's latency is estimated (*remainingLatency*, l. 8). The algorithm then tries to move the instruction upward one by one (l. 9 – 21). At each step, data dependencies (l. 12) and register pressure (l. 17) are verified. When these checks succeed, the instruction moves upward and the *remainingLatency* estimate is updated (l. 18). This is repeated until the estimation of the instruction's latency is completely hidden, the instruction reaches the top of the basic block or is blocked by a data dependency, or the register pressure limit would be exceeded (*limit*).

The latency estimation *remainingLatency* (l. 8) is computed using *getLatencyCost*, usually a constant denoting the cache-hit or load-use latency, and *getDynamicLoadCost*. The latter is an estimation of the latency induced by stalling at the next use of the loaded value, ignoring intermittent spill-related loads. *remainingLatency* is updated using the function *eatenLatency*, which estimates the reduction of the instruction's latency when moving upward in the code. In our case, this function returns $\frac{1}{4}$ for regular instructions and 1 for memory accesses; that is, bundles are assumed dense.

The inner loop (l. 9 – 21) is executed at most once for each load; that is, each instruction moves once and then remains at its location. Moving spill-related loads may never incidentally increase the latency of the instructions processed so far. However, the insertion of loads may further reduce the latency.

Note that the algorithm presented here does not deal with precolored registers, multiple register files, or special instructions that cannot be moved. In practice, these constraints have to be respected.

For our target, the load-use latency is not the single source of regression. The bundle density is also relevant. To also improve that point, we perform a similar optimization on stores. The interest of such a store placement is to give more freedom to the final scheduler for placing stores into bundles. Without that, because of postcoloring constraints (write after read), the stores may be stuck at some place.

Fig. 14. Runtime per EEMBC benchmark using frequency estimate (left) and profiling feedback (right). In both cases, latency optimization is performed after spilling and before coloring. (Lower is better.)

These heuristics were applied to all spilling strategies as depicted by the middle bars in Figure 13, resulting in additional runtime improvements of about 4% on almost all configurations while preserving the same static spill costs! The spill-everywhere strategies, which place load operations right before uses, profit the most, showing mean speedups of more than 10%. Manual inspection of the resulting code in all configurations indicates that our heuristic is able to resolve almost all spill-code-related latency violations, without changing the spilling decisions.

The rightmost bars of Figure 13 give the results obtained using accurate profiling feedback combined with the latency heuristic. In other words, we check the performance of the model in a configuration where the static spill costs reflect much more closely the actual runtime behavior. In this setting, the runtime of our formulations (i.e., Basic, BasicSSA, Pessimistic, and Optimistic with and without rematerialization) is 8% to 13% better than AG in the original setup (AG leftmost bar compared to our formulation's rightmost bars) and 4% to 9% better than AG with profiling and latency optimization (rightmost bar). Note that, at least for the EEMBC benchmarks, Hack and BH perform very well, despite a few bad outliers.

Our formulations show clear improvements, but some outliers still remain. Figure 14 shows the runtime measurements per benchmark (freq. estimates left, profiling feedback right). With profiling enabled, the number of bad cases for Basic and Pessimistic, which are supposed to be better than AG in static spill costs, is reduced and the peaks are smaller. The remaining outliers stem from interactions between spill code, register copies, and bundling, which has a large impact on the runtime (as noted before). Also, our model assumes that critical edge splitting is for free. This model is only accurate when the blocks created during edge splitting remain empty, since the blocks along with the related branch instructions can be removed. When spill code or register copies remain in those blocks, even after register coalescing, the overhead of the branch instruction is unavoidable. All these situations can be exacerbated when spilling merely reduces register pressure to the bare minimum needed. This may lead to both uncoalescable register copies (permutations) and stronger constraints for the postpass scheduler. For such cases at hot regions (loops), spilling a bit more may be useful.

4.5. Simplifying Assumptions

In this section, we investigate the impact of several simplifying assumptions on runtime. Some of these simplifications were already used in previous work; others are introduced here. Clearly, these assumptions have a negative impact on static spill costs. However, no other study systematically investigated their actual impact. To do so, we added additional constraints to our ILP formulations and performed the same

Fig. 15. Runtime impact of simplifications for Basic (left) and Pessimistic (right), over all benchmarks. Numbers are normalized to the respective base configuration with frequency estimate. (Lower is better.)

Fig. 16. Runtime impact of simplifying assumptions for Basic (left) and Pessimistic (right) with postlatency optimization. Base represents the respective configuration without simplifications. Numbers are normalized to the respective base configuration with frequency estimate. (Lower is better.)

set of experiments as before. Due to space considerations, we limit our discussion to two configurations for the EEMBC benchmarks: basic (non-SSA) and pessimistic (SSA), optionally followed by our postlatency optimization and/or using profiling feedback.

4.6. The Instruction store

On most architectures, store instructions are an order of magnitude cheaper than load instructions. Heuristics thus often make simplifying assumptions about stores.

Free stores. The first simplifying assumption is to consider store instructions to be free. Consequently, only the placement of loads is optimized by the spill cost model. Note that, even when they are free, useless store instructions are of course not inserted in the different methods we evaluated. The red curve with square markers in Figures 15 and 16 shows the impact of this assumption on runtime.³

For the original Basic configuration with frequency estimate and without postlatency optimization (Figure 15, left), this assumption causes slowdown above 5% in only five out of 38 benchmarks. The impact is comparable for the Pessimistic configuration (right). This assumption coupled with our postlatency optimization (Figure 16) shows even slightly better results. For Basic, the runtime is degraded only for four benchmarks and for Pessimistic only for a single benchmark. In addition, the amplitude of the peaks is greatly reduced. The same trend also holds when accurate frequencies (i.e., with profiling feedback) are used (omitted here for brevity).

³In all these figures, benchmarks are sorted so that one of the curves (“store at definition,” then “Base”) increases. This is an arbitrary choice to make the figures more readable.

Overall, the assumption that stores are for free has little impact on the runtime. We conclude that this simplification is reasonable when developing spilling heuristics. However, to our knowledge, this assumption is never used alone but is usually coupled with other assumptions. It is particularly important to constrain the placement of stores in the code, even when they are considered free. Note also that these figures illustrate again the weakness of the static spill cost model (i.e., the model used in our ILP). Although this simplification is a restriction, the runtimes improve in many cases.

Stores at definitions. A common assumption in spilling heuristics is to place a store at each definition of the related variable. In particular, all heuristics that use a spill-everywhere approach, from linear scan to puzzle solving through graph coloring [Chaitin 1982; George and Appel 1996; Pereira and Palsberg 2008; Poletto and Sarkar 1999], use that simplification. This is also true for more recent SSA-based spilling approaches [Ebner et al. 2009; Hack 2007; Braun and Hack 2009]. Moreover, this simplification may be combined with the previous assumption, as in the progressive spill-code placement of Ebner et al. [2009].

The brown curve with triangle markers in Figures 15 and 16 shows the impact of this assumption on the runtime. The black curve with star markers demonstrates the impact when coupled with the previous assumption (Free store).

This assumption alone has very little impact on the runtime. This applies in particular to non-SSA programs, as shown by Figure 15 (left). Under SSA, there are some very bad cases (see Figure 15 (right)), with two benchmarks slowed down by more than 10%. When stores are additionally assumed to be free, the impact overall remains limited. Still, a few bad cases are observed. The results are slightly worse under SSA, mainly due to the increased number of definitions. ϕ -functions pose a particular problem, as they tend to cluster stores at the same point, which are then harder to schedule. When our postlatency optimization is enabled (Figure 16), the number of cases that are worse (i.e., above the blue Base curve) is reduced. The benefit of this latency optimization is immediately visible as it helps to schedule the (clustered) stores more freely. With accurate profiling feedback, the results are slightly better.

We conclude that these two assumptions make perfect sense for a heuristic. This is particularly true when a postlatency optimization is applied that increases the scheduler's freedom. Also, as stated before, the increased static spill costs are not reflected in corresponding runtime degradations. On the contrary, we often observed improvements.

4.7. The Instruction load

As load instructions are usually considered expensive, fewer simplifications are used in today's heuristics. Based on the static spill cost metric, Goodwin and Wilken [1996] demonstrated that the optimality of this metric is preserved if the insertion points of load instructions are limited to the end of basic blocks or just before the related uses (the static cost is the same for all points of a basic block). Note that the instruction latencies are not taken into account. Spill-everywhere-based heuristics use even more limited insertion points: loads are inserted just before *all* related uses, even if the variable has been already loaded earlier. It is possible, however, to eliminate these redundant loads afterward [Bodík et al. 1999]. The SSA-based spilling heuristic of Braun and Hack [2009] does not explicitly limit the insertion points of load instructions. However, by construction, it always inserts loads on edges (i.e., at the end of the previous basic blocks) or just before uses. But a previously loaded variable can be reused to avoid redundant loads.

Loads at uses. We decided to test a model simpler than those mentioned earlier but more general than a spill-everywhere strategy: the placement of loads is restricted to

points before uses, but, unlike spill-everywhere, redundant loads are avoided. In terms of the static cost model, this is equivalent to optimal spill-everywhere coupled with a redundant load elimination. Moreover, stores are assumed to be free and placed at definitions—as we showed they were valid simplifications with respect to runtime.

The impact of this approach on runtime is shown by the green curve with diamond markers in Figures 15 and 16. As shown in Section 3, forcing the placement of loads just before uses produces the worst possible latency. It is not surprising that the performance of this simplification is rather disappointing (Figure 15). This observation completely changes when our latency optimization is enabled (Figures 16). In this setting, the simplification is, overall, as good as the Base configuration (blue curve), with a few bad cases. As the runtime costs of loads are higher than the cost of stores, the accuracy of the frequency estimates in the cost model has a larger impact on the runtime. However, the general trend is similar when profiling feedback is enabled.

In conclusion, we believe that this model is quite accessible to heuristics. Moreover, we showed that its impact on runtime, as soon as it is coupled with a latency optimization, is overall little, compared to an optimal model based on static spill costs. It is a promising starting point for the development of new spilling heuristics.

5. CONCLUSION

We proposed a new decoupled spilling formulation based on ILP that is applicable to SSA and non-SSA programs. It is more expressive and “complete” than previous “optimal” approaches, allows us to model tradeoffs, and can be used to emulate spilling heuristics as well as previous optimal formulations. From our elaborate experiments evaluating and comparing alternative spilling strategies and previous optimal formulations, we draw the following conclusions to take away from this work:

- Most work on spilling focuses on the placement of loads and stores. Rematerialization is merely treated as an afterthought. Our experiments show that rematerialization is essential to good performance and needs to be accounted for accordingly.
- While SSA provides clear advantages during register assignment, it does not provide any benefits for spilling. On the contrary, the complicated semantics of ϕ -functions requires an equally complicated handling of memory-to-memory copies. Instead, it is preferable to perform live-range splitting and rematerialization on demand, as demonstrated by our non-SSA formulation.
- Using our new formulations, we achieved surprising gains wrt. standard objectives such as static spill costs, instruction counts, and cache-miss rates, even in comparison to previous optimal formulations. This shows that there is still headroom for improved spilling heuristics.
- However, static spill costs do not reliably predict runtime performance, as too many aspects of the underlying architecture are masked out. Thus, evaluating and reporting static spill costs alone, as is often done, is not sufficient to fully judge the benefits of a spilling strategy. An important observation, supported by experiments with our postlatency optimization and manual inspection, is that aggressive spilling with regard to static spill costs can be counterproductive. Sometimes it is advantageous to spill a bit more in order to relax constraints on subsequent optimizations, including register assignment and instruction scheduling. More research is needed to explore alternative cost models that reliably guide spilling.

REFERENCES

- A. W. Appel and L. George. 2001. Optimal spilling for CISC machines with few registers. In *Conference on Programming Language Design and Implementation (PLDI'01)*. ACM, 243–253.
- T. Ball and J. R. Larus. 1993. Branch prediction for free. In *Conference on Programming Language Design and Implementation (PLDI'93)*. ACM, 300–313.

- R. Bodík, R. Gupta, and M. L. Soffa. 1999. Load-reuse analysis: Design and evaluation. In *Conference on Programming Language Design and Implementation (PLDI'99)*. ACM, 64–76.
- F. Bouchez, A. Darté, C. Guillon, and F. Rastello. 2006. Register allocation: What does the NP-completeness proof of Chaitin et al. really prove? or revisiting register allocation: Why and how? In *Languages and Compilers for Parallel Computing (LCPC'06) (LNCS)*, Vol. 4382. Springer, 283–298.
- F. Bouchez, A. Darté, and F. Rastello. 2007a. On the complexity of register coalescing. In *International Symposium on Code Generation and Optimization (CGO'07)*. IEEE Computer Society.
- F. Bouchez, A. Darté, and F. Rastello. 2007b. On the complexity of spill everywhere under SSA form. In *Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'07)*. 103–112.
- M. Braun and S. Hack. 2009. Register spilling and live-range splitting for SSA-form programs. In *Compiler Construction (CC'09) (LNCS)*, Vol. 5501. Springer, 174–189.
- P. Briggs, K. D. Cooper, and L. Torczon. 1992. Rematerialization. In *ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation (PLDI'92)*. ACM, 311–321.
- G. J. Chaitin. 1982. Register allocation & spilling via graph coloring. In *Symposium on Compiler Construction (CC'82)*. ACM, 98–105.
- G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. 1981. Register allocation via coloring. *Computer Languages* 6 (Jan. 1981), 47–57.
- D. Ebner, B. Scholz, and A. Krall. 2009. Progressive spill code placement. In *Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'09)*. ACM, 77–86.
- H. Falk, N. Schmitz, and F. Schmoll. 2011. WCET-aware register allocation based on integer-linear programming. In *Proceedings of the Euromicro Conference on Real-Time Systems*. IEEE, 13–22.
- C. Fu and K. Wilken. 2002. A faster optimal register allocator. In *Symposium on Microarchitecture (MICRO'35)*. IEEE Computer, 245–256.
- L. George and A. W. Appel. 1996. Iterated register coalescing. *ACM Transactions on Programming Languages and Systems* 18, 3 (May 1996), 300–324.
- D. W. Goodwin and K. D. Wilken. 1996. Optimal and near-optimal global register allocation using 0-1 integer programming. *Software: Practice and Experience* 26, 8 (1996), 929–965.
- S. Hack. 2007. *Register Allocation for Programs in SSA Form*. Ph.D. Dissertation. Universität Karlsruhe. Retrieved from <http://digbib.ubka.uni-karlsruhe.de/volltexte/documents/6532>.
- S. Hack and G. Goos. 2006. Optimal register allocation for SSA-form programs in polynomial time. *Information Processing Letters* 98, 4 (May 2006), 150–155.
- D. R. Koes and S. C. Goldstein. 2006. A global progressive register allocator. In *Conference on Programming Language Design and Implementation (PLDI'06)*. 204–215.
- D. R. Koes and S. C. Goldstein. 2009. Register allocation deconstructed. In *Workshop on Software & Compilers for Embedded Systems (SCOPE'S'09)*. ACM, 21–30.
- F. M. Q. Pereira and J. Palsberg. 2008. Register allocation by puzzle solving. In *Conference on Programming Language Design and Implementation (PLDI'08)*. ACM, 216–226.
- M. Poletto and V. Sarkar. 1999. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems* 21, 5 (1999), 895–913.
- V. C. Sreedhar, R. D.-C. Ju, D. M. Gillies, and V. Santhanam. 1999. Translating out of static single assignment form. In *Symposium on Static Analysis (SAS'99) (LNCS)*, Vol. 1694. Springer, 194–210.
- L. Torczon and K. Cooper. 2011. *Engineering A Compiler* (2nd ed.). Morgan Kaufmann, San Francisco, CA.
- M. Yannakakis and F. Gavril. 1987. The maximum k -colorable subgraph problem for chordal graphs. *Information Processing Letters* 24, 2 (1987), 133–137.

Received June 2014; revised September 2014; accepted October 2014