Optimal General Offset Assignment

Sven Mallach
Institut für Informatik
Universität zu Köln
Cologne, Germany
mallach@informatik.uni-koeln.de

Roberto Castañeda Lozano
SCALE
SICS (Swedish Institute of Computer Science)
Sweden
rcas@sics.se

ABSTRACT

We present an exact approach to the General Offset Assignment problem arising in the domain of address code generation for application specific and digital signal processors. General Offset Assignment is composed of two subproblems, namely to find a permutation of variables in memory and to select a responsible address register for each access to one of these variables. Our method is a combination of established techniques to solve both subproblems using integer linear programming. To the best of our knowledge, it is the first approach capable of solving almost all instances of the established OffsetStone benchmark set to global optimality within reasonable time. We provide a first comprehensive evaluation of the quality of several state-of-the-art heuristics relative to the optimal solutions.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*code generation, compilers, Optimization*; G.1.6 [**Mathematics of Computing**]: Optimization—*Integer programming*

Keywords

Compiler optimization, digital signal processors, application specific processors, address code generation, offset assignment, integer programming, branch-and-cut

1. INTRODUCTION

Indirect addressing is a common way to reference memory operands of instructions where the address of an operand is not passed explicitly but using a register. One particular form of indirect addressing interprets the content of a register as a base address that may then be combined with a further immediate offset. This is particularly useful, e.g., for accesses to an array where the register may hold its starting address and the offset specifies an element to be referenced. However, addressing modes using an immediate offset require a wide instruction word since the offset operand needs to be passed directly. While this is usually no restriction for general purpose processors, specialized designs such as application-specific processors (ASIPs) and (older) digital signal processors (DSPs) frequently do not support arbitrary offsets in order to save silicon area. The respective address of an operand then needs to be explicitly present in one of the processors address registers (ARs). At the same time, to limit the drawbacks of this restriction, such processors usually provide an *address generation unit* (AGU). The AGU permits modifications to an AR to be performed in the same clock cycle as another instruction referencing the AR. For example, an address can be used to access an operand for a multiplication and be incremented afterwards at no extra cost. However, AGU instructions implicitly encode the respective modification so that only small ranges r of in- or decrements are possible. In fact, many ASIPs and DSPs have small instruction width and even only support autoin- or decrements by a single memory word, i.e., $r = 1$ [14, 11].

In a way, one can consider the complexity of indirect addressing to have been moved from hardware to software for processors with restricted offsets. In the absence of support for arbitrary offsets, address computation overhead may result from two main issues. An inappropriate storage layout may lead to a large number of additional explicit address arithmetic instructions for 'jumps' to addresses that have a distance larger than r. Further, if the processor provides multiple ARs, a suboptimal choice of the ARs responsible for particular accesses may result in avoidable immediate AR loads and, again, additional arithmetic is needed. However, compilers may freely choose the stack layout for local variables of a function and the AR responsible for a particular access to a variable. Since address calculations make up a significant part of machine instructions, optimizing these decisions may considerably reduce the code size and speed up the program at the same time. Indeed, various experimental studies [11, 12, 14] show that optimized configurations lead to significant savings in practice.

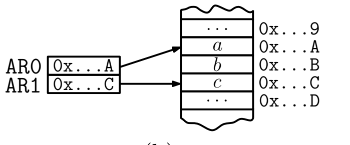
1.1 Motivating Example

During compilation, the instruction scheduling phase determines the *access sequence* to local stack variables. It can be extracted by simply concatenating the referenced variables of each instruction $c = a\ op\ b$ in the order $a\ b\ c$. For example, the program fragment in the left of Fig. 1 refers to the variables $\mathcal{V} = \{a, b, c, d, e, f, g\}$ that are accessed in the order $S = a\ b\ c\ g\ c\ f\ c\ e\ c\ c\ f\ d$. Tab. 1 shows pseudo machine code for this program fragment and three potential stack layouts A, B, and C of $\mathcal{V}$. Layout A refers to the order of first use (OFU) of the variables in S. On a pro-

Permission to make digital or hard copies of all or part 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 bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
SCOPES '14 June 10 - 11 St. Goar, Germany.
Copyright is held by the owner/author(s). Publication rights licensed to ACM. ACM 978-1-4503-2941-5/14/06 $15.00.

```
c = a + b;
f = g - c;
c = c - e;
d = c * f;
```

(a)



(b)

Figure 1: A sample code fragment (a), and an illustration of two ARs referencing memory locations of variables (b).

Instruction	AR0	**Instruction**	AR0	**Instruction**	AR0	AR1
LDAR AR0, &a	&a	LDAR AR0, &a	&a	LDAR AR0, &a	&a	
LOAD *(AR0)+	&b	LOAD *(AR0)+	&b	LOAD *(AR0)+	&b	
ADD *(AR0)+	&c	ADD *(AR0)		ADD *(AR0)+	&c	
STOR *(AR0)+	&g	ADAR AR0,2	&c	STOR *(AR0)		
LOAD *(AR0)-	&c	STOR *(AR0)-	&g	LDAR AR1, &g		&g
SUB *(AR0)		LOAD *(AR0)+	&c	LOAD *(AR1)-		&e
ADAR AR0,2	&f	SUB *(AR0)+	&f	SUB *(AR0)+	&f	
STOR *(AR0)		STOR *(AR0)-	&c	STOR *(AR0)-	&c	
SBAR AR0,2	&c	LOAD *(AR0)		LOAD *(AR0)		
LOAD *(AR0)		ADAR AR0,3	&e	SUB *(AR1)		
ADAR AR0,3	&e	SUB *(AR0)		STOR *(AR0)+	&f	
SUB *(AR0)		SBAR AR0,3	&c	MUL *(AR0)+	&d	
SBAR AR0,3	&c	STOR *(AR0)+	&f	STOR *(AR0)		
STOR *(AR0)		MUL *(AR0)+	&d			
ADAR AR0,2	&f	STOR *(AR0)				
MUL *(AR0)						
ADAR AR0,2	&d					
STOR *(AR0)						
$A = a\ b\ c\ g\ f\ e\ d$		$B = a\ b\ g\ c\ f\ d\ e$		$C = a\ b\ c\ f\ d\ e\ g$		

Table 1: Pseudo machine codes for the code fragment from Fig. 1 assuming different memory layouts A, B and C and either one (A and B) or two (C) available address registers.

cessor with only a single AR this layout would require six explicit address arithmetic instructions (ADAR and SBAR). An optimized layout (B) already reduces the necessary number of such instructions to three by increasing the use of autoin-/decrement instructions (*(ARx)+ and *(ARx)-). If the memory layout is optimized for a use of two ARs (C) and also an optimal AR assignment is computed, it becomes possible to cover the access sequence even without any explicit address arithmetic at all. Assuming the cost of an immediate AR load and the cost of an address arithmetic instruction to be both one, the optimal total cost with one AR would be four, with two ARs it would be two. Notably, layout A and B have no register assignment that leads to a total cost smaller than three with two or more ARs.

1.2 Problem Statement and Related Work

Given an access sequence of program variables, $k \geq 1$ ARs and an autoin-/decrement range of one, the problem to minimize the address computation overhead is called the *General Offset Assignment* (GOA) problem. As already indicated, GOA consists of two subtasks. On the one hand, a memory layout of the program variables needs to be determined and on the other, it must be decided which AR shall be used to perform each of the accesses to these variables. The first subproblem is the 'real' offset assignment since it determines the distances between program variables in memory. The second task is called *Address Register Assignment (ARA)*. If there is only a single AR, ARA is not necessary and the problem is then called the *Simple Offset Assignment* (SOA) problem which is already $\mathcal{NP}$-hard [17].

Offset Assignment was first considered by Bartley [3] in 1992. He proposed to model the variable relationships contained in an access sequence by an *access graph* $G = (V, E)$. The set of vertices V corresponds to the variables and there

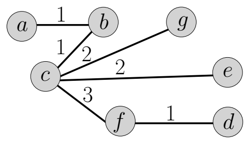


Figure 2: Access graph for the code fragment from Fig. 1.

is an edge $e = \{u, v\} \in E$ with weight $w(e)$ if the variables u and v appear subsequently in the access sequence for $w(e)$ times. Fig. 2 shows the access graph that corresponds to the sample code fragment shown in Fig. 1.

Bartley recognized a close relationship of SOA to the Maximum Weight Hamiltonian Path (MWHP) problem and developed a first greedy heuristic to solve it. In subsequent research, Liao [17] showed that SOA is rather equivalent to a Maximum Weight Path Cover (MWPC) problem instead and gave a formal proof of its strong $\mathcal{NP}$-hardness. Based on these results, he proposed a simpler and faster heuristic producing solutions with the same quality as Bartley's and also a first exact Branch-and-Bound procedure. A series of SOA heuristics were developed subsequently and experimentally evaluated by Leupers in 2003 [14]. Leupers introduced a standard reference benchmark set called *OffsetStone* that has been frequently used for experiments since then. Recently, the first exact approach to SOA capable to solve all instances from OffsetStone to optimality in reasonable time was presented in [12] and the heuristics known so far were broadly evaluated in relation to optimal solutions for the first time. However, to the best of our knowledge, this has never been carried out for GOA algorithms on a large benchmark set like OffsetStone prior to this article.

In practice, GOA is often solved by first partitioning the set of program variables w.r.t. the available ARs and then solving a SOA problem for each of the ARs. This strategy allows to reuse available SOA algorithms but inherently constrains all accesses to a particular variable to be performed by the same AR. This may preclude optimal results as is extensively discussed by Huynh et al. [11]. They also evaluate different combinations of ARA and SOA algorithms. The approach by Sugino et al. [25] performs best in their experiments. Their method partitions the variables iteratively by applying in each iteration a min-cut-based heuristic algorithm that repeatedly invokes a SOA algorithm to estimate the quality of the partition. The SOA algorithm iteratively removes edges from the access graph until a variable ordering can be trivially derived.

Besides his algorithms for SOA, Liao [17] also proposed a heuristic for GOA which however needed some manual parameter specification. In 1997, Leupers and Marwedel [16] proposed a different heuristic that outperforms Liao's on their random test instances. Their method was also used to generate initial populations for a genetic GOA-algorithm presented one year later by Leupers and David [15] that, however, also assigns all accesses to a particular variable to the same AR. Both articles further discuss the integration of modify registers (MRs) present in many ASIPs and DSPs into the optimization problem. MRs store offsets which may then be added to or subtracted from the address held by an AR at no extra costs. This can mitigate the address computation overhead especially if certain distances between variables occur frequently. Loading MRs with new offsets however also requires additional instructions. Ozturk et al. [20] provided a first ILP approach to GOA which can also be extended to deal with MRs. However, their model has flaws, e.g., it does not account for AR initialization costs and allows *any present AR* to be moved in parallel with *any instruction* regardless whether the instruction references the respective AR or not. This is hardly imaginable in hardware. We discuss all issues in detail in this paper's appendix. Apart from these, the formulation leads to critical numbers of variables and constraints and does neither exploit the combinatorial structure nor symmetries inherent to the problem. The authors reported relatively high running times that do not suggest to use their method in a production compiler.

In case that a memory layout of the variables is already *given*, an optimal assignment of accesses to registers can be computed in polynomial time. This was shown by Gebotys [9, 10] who provided a minimum-cost circulation algorithm that we will discuss in more detail in Sect. 3.1.

Specialized and integrated variants of the problem were also subject to research. Lorenz et al. [18] consider offset assignment in the context of specialized DSPs with wide memory and where accesses do not refer to single memory words but to all variables of a previously specified group simultaneously. Eriksson [8] proposed a dynamic programming algorithm to integrate scheduling, AR and offset assignment. However, the algorithm is highly time and memory consuming and could solve only small instances. A different idea named *variable coalescing* is to share storage locations among variables whose lifetimes do not overlap. Ottoni et al. [19] presented a first heuristic which was followed by another one and an ILP formulation by Salamy and Ramanujam [23, 24]. However, for the exact approach again the instances solved had to be restricted to sizes of about 30 variables due to the running time of the solver.

Further research deals with address code optimizations by computation or operand reordering. Rao and Pande [22] apply algebraic transformations to expression trees in order to find a least-cost access sequence. Similarly, Atri et al. [2] use commutative transformations of the access pattern to obtain better solutions with existing heuristics. Choi and Kim [4] perform code transformations and reschedule parts of the code as a preprocessing step to offset assignment.

1.3 Our Contribution

This paper presents new exact integer linear programming (ILP) formulations and algorithms to solve the GOA problem to optimality. They deliver optimal solutions for nearly all instances of the standard OffsetStone benchmark set [14] for varying numbers of ARs. In contrast to existing exact approaches, our algorithms solve the majority of instances within only a few milliseconds of CPU time and also model the autoin-/decrement capability of processors accurately. Hence, we are able to provide a first comprehensive evaluation of the quality of state-of-the-art GOA heuristics using the OffsetStone benchmark.

Due to the complexity of the problem, one cannot expect an algorithm that is capable to solve *any* instance to optimality in acceptable time. Nonetheless, based on the large set of instances we used in our experiments, our exact method is practical whenever a particular program is compiled infrequently and can be combined with a time limit and a fallback heuristic to deal with very large instances. This could be a promising strategy in cases where code size and runtime performance is critical like, e.g., for firmware to be stored in a small read-only memory.

As a side effect, we revised the ILP formulation by Ozturk et al. [20] and reimplemented several heuristics to incorporate them into our experiments. The results reveal significant quality discrepancies between heuristics that subdivide GOA into ARA and SOA and those approaches that derive a memory layout and solve ARA optimally for that layout.

The rest of this paper is organized as follows. We discuss the two arising subproblems of GOA and optimal solution techniques for them in Sect. 2 and Sect. 3. Based on this, we present novel integer programming formulations and algorithms to solve the General Offset Assignment Problem to optimality in Sect. 4. In Sect. 5, a short overview on our extensive experiments is given and Sect. 6 finally presents our results. We conclude on this contribution in Sect. 7.

2. OPTIMAL OFFSET ASSIGNMENT

The most important aspect regarding offset assignments is that memory layouts correspond to variable permutations which might be, e.g., modeled by Hamiltonian paths or path covers. For SOA it has been shown that the problem can be reduced to a Maximum Weight Path Cover (MWPC) [17], Maximum Weight Hamiltonian Path (MWHP) [3] or Maximum Weight Hamiltonian Cycle (MWHC) [12] problem. Interpretations of an optimal SOA memory layout for the example from Fig. 1 as MWPC, MWHP and MWHC are depicted in Fig. 3. Path covers are collections of disjoint paths and isolated vertices (considered as paths of length zero, like the vertex e in the left of Fig. 3) covering all vertices of the graph. Any concatenation of these paths using additional edges yields a Hamiltonian path that, interpreted as a memory layout, is then a feasible offset assignment. Using an artificial vertex as a unique splitting point, one may also identify a Hamiltonian path by computing a Hamiltonian cycle and removing the vertex again afterwards.

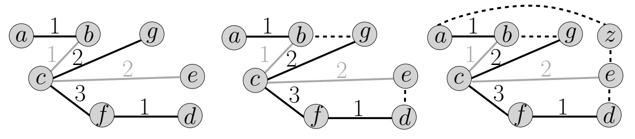


Figure 3: A path cover (left), and one possible corresponding Hamiltonian path (middle) created by appending additional edges (dashed). On the right, the path is identified as part of a Hamiltonian cycle using an artificial vertex z.

Due to the additional task of address register assignment, we could not find a way to model optimal solutions to GOA correctly by assigning static costs to the edges of an access graph. However, our approach still relies on the idea to model permutations by paths or tours.

2.1 Modeling Permutations

As discussed in [12] the main issue and commonality between Hamiltonian paths and path covers is that solutions to these problems must be free of cycles. This is also reflected in the corresponding integer programming formulations. At this point, we are just interested in how to model feasible solutions, so we omit edge weights and objective functions in the following ILP model descriptions.

Let $G = (V, E)$ be an access graph and let $x_{u,v} \in \{0, 1\}$ be a decision variable for each edge $\{u, v\} \in E$ expressing whether it is selected or not. For ease of notation, define $x(S) = \sum_{\{u,v\}\in S} x_{u,v}$, i.e., $x(S)$ is the sum of the variables associated to any edge-subset $S \subseteq E$. Then the set of feasible solutions to a path cover formulation can be stated as:

$$\begin{array}{lll} \sum\limits_{\{u,v\}\in E} x_{u,v} & \leq 2 & \forall\, v \in V \\ x(C) & \leq |C| - 1 & \forall \text{ cycles } C \subseteq E \\ x_{u,v} & \in \{0, 1\} & \forall\, \{u, v\} \in E \end{array}$$

The first inequalities force each vertex to have at most two incident edges in the cover. The cycle inequalities exclude any solutions that contain cycles from the feasible set.

In order to solve the problem by finding a Hamiltonian path instead, we create a complete graph $G_C = (V_C, E_C)$ from the access graph $G = (V, E)$ as follows. We set $V_C = V \cup \{z\}$ where z is an additional vertex like in Fig. 3 and $E_C = \{\{u, v\} \mid u, v \in V_C\}$. Computing a Hamiltonian cycle in G_C and removing the vertex z from the cycle yields the desired Hamiltonian path in G [12]. Let $E(W) = \{\{u, v\} \in E_C \mid u, v \in W\}$ for any $W \subseteq V_C$. Then the set of Hamiltonian cycles can be expressed exactly as when stating a Traveling Salesman Problem (TSP):

$$\begin{array}{lll} \sum\limits_{\{u,v\}\in E} x_{u,v} & = 2 & \forall\, v \in V_C \\ x(E(W)) & \leq |W| - 1 & \forall\, \emptyset \neq W \subsetneq V_C \\ x_{u,v} & \in \{0, 1\} & \forall\, \{u, v\} \in E_C \end{array}$$

Compared to the path cover formulation, the equations force any vertex to be adjacent to *exactly* two other vertices. The inequalities are called *subtour elimination constraints* (SECs) [5]. They exclude solutions containing *any* cycle w.r.t. the vertex sets W and $V_C \setminus W$ from the feasible set.

The numbers of possible cycles $C \subseteq E$ as well as vertex sets $W \subseteq V_C$ are exponential with respect to the size of the respective base set. One would therefore usually not add the cycle inequalities or the SECs to a ILP solver from the beginning. Instead, one could *separate* them, i.e., check whether computed linear program (LP) solutions violate these constraints and then add them only when necessary. When combined with traditional Branch-and-Bound, such a solution strategy is usually referred to as 'Branch-and-Cut' [7]. We do not want to go into detail here but shortly address one important aspect. If we consider a minimization problem, feasible solutions provide upper bounds on the optimal objective value, and LP solutions provide lower bounds. Strong inequalities are therefore important to prove optimality of a known solution. Similarly, the addition of (previously violated) inequalities successively strengthens the solved LPs which may lead to better lower bounds. Separation of the cycle inequalities and SECs can both be performed in polynomial time. However, in experiments with our GOA formulations (which we present in Sect. 4), the Hamiltonian Path formulation performed much better concerning the provided lower bounds. Further, it permits the application of several strong inequalities known for the TSP which in turn allow for a much better separation of fractional LP solutions as is discussed in [12]. However, the advantage over the path cover formulation comes at the cost of more variables induced by completing the access graph. Nevertheless, in our experiments this strategy is superior at least for instances with up to a few hundred vertices.

3. OPTIMAL ADDRESS REGISTER ASSIGNMENT

Suppose for now that a memory layout $\mathcal{L}$ of the program variables $\mathcal{V}$ has already been fixed and we are now asked to compute an optimal ARA for k address registers w.r.t. $\mathcal{L}$ and the input access sequence S.

3.1 Gebotys' Circulation Technique

An exact solution to this problem has been proposed by Gebotys [9, 10]. It is based on a minimum-cost circulation network that contains a vertex for each access in S and a directed arc for each pair of accesses u, v such that v succeeds u in S. As a small example, let $\mathcal{V} = \{a, b, c, d\}$, $S = a\ d\ c\ a\ b\ c$ and assume $\mathcal{L} = d$ - a - c - b (which is optimal for $k \geq 2$ ARs). After adding artificial 'source' (s) and 'sink' (t) vertices, the associated circulation network looks as depicted in Fig. 4. The cost of an arc between two accesses is zero if and only if the two associated variables are equal or adjacent in $\mathcal{L}$. Otherwise the cost c_A of an address arithmetic instruction is associated with the arc (drawn dashed in Fig. 4). All costs of arcs leaving s or entering t are zero and the arc (t, s) has the cost c_L of an immediate address register load.

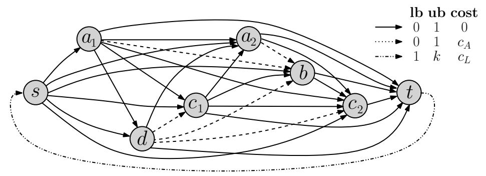


Figure 4: Circulation network assuming $\mathcal{L} = d$ - a - c - b.

Each vertex is constrained to receive and supply one unit of flow and the capacity of all arcs is one, except for the arc (t, s) that has capacity k. Hence, the maximum possible flow in this circulation network is k and each unit of flow leaving s essentially delivers a path of accesses before it proceeds to t. If the selection of these paths is based on a min-cost criterion then each of the resulting paths can be interpreted as an optimal series of accesses performed by an AR.

Minimum-cost circulations can be established in polynomial time using either combinatorial algorithms (e.g. [26]) or linear programming (solutions will be always integral) [13]. Let V_S be the set of vertices associated with the accesses in sequence S. The circulation network $N = (V_N, A)$ is obtained by setting $V_N = V_S \cup \{s, t\}$ and A to be the union of the arc sets $\{(s, v) \mid v \in V_S\}$, $\{(v, w) \mid v, w \in V_S, v < w\}$, $\{(v, t) \mid v \in V_S\}$ and the arc (t, s). Let $y_{u,v}$ be a flow variable for each arc $(u, v) \in A$ and $c_{u,v}$ its associated cost. Gebotys' LP formulation is then:

$$\begin{array}{llcll} \min & \sum\limits_{(u,v)\in A} c_{u,v} y_{u,v} & & & \\ s.t. & \sum\limits_{(v,w)\in A} y_{v,w} - \sum\limits_{(u,v)\in A} y_{u,v} & = & 0 & \forall\, v \in V_N \\ & \sum\limits_{(v,w)\in A} y_{v,w} & = & 1 & \forall\, v \in V_S \\ & \sum\limits_{(u,v)\in A} y_{u,v} & = & 1 & \forall\, v \in V_S \\ 0 \leq & y_{u,v} & \leq & 1 & \forall\, (u, v) \neq (t, s) \in A \\ 1 \leq & y_{t,s} & \leq & k & \end{array}$$

The restriction of the in- and out-degrees of all sequence-vertices to one by the second and third constraints highlights the aforementioned 'path selection' or 'assignment problem' property of this model. Actually, they make the preceding flow conservation constraints obsolete for all vertices except t since any unit of flow sent from s to satisfy the equations must finally arrive at t and will then be sent back using the circulation arc (t, s). Further, lower bounds on the flow on out- (in-) arcs of the source (target) are not necessary since the former (latter) must be satisfied due to the in- (out-) degree equation of the first (last) access vertex.

3.2 An Equivalent Min-Cost Flow Model

Further inspection of the problem and basic results from network flow theory allow for a simple transformation of this model into a usual min-cost flow problem where the arc (t, s) is removed. Its cost (the cost of an immediate AR load) can be instead installed on every s-leaving arc. This yields the same result that the cost is paid as soon a further register is used to cover the access sequence. The restriction to not use more than k registers (the former upper bound on the flow over arc (t, s)) can be applied over the sum of all s-leaving arcs instead. In this manner, the vertices s and t will be a real source and sink, respectively. An optimal solution (assuming $c_L = c_A$) to the example from Fig. 4 within the transformed network and for $k \geq 2$ ARs is depicted in Fig. 5. It also illustrates that it is not necessarily optimal to assign accesses to the same variable (here c) to the same AR.

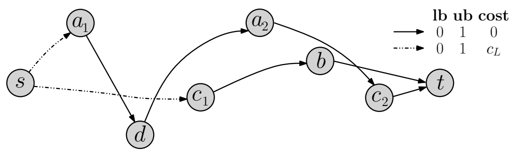


Figure 5: Optimal solution to the example from Fig. 4 shown in the min-cost flow representation of the problem.

After the transformation, all flow variables may be only zero or one and still combinatorial algorithms as well as linear programming can be used in order to obtain integral solutions in polynomial time. Assuming that the arc set A now does not contain the arc (t, s) anymore, the IP formulation of the transformed and reduced problem is:

$$
\begin{array}{llll}
\min & \sum\limits_{(u,v)\in A} c_{u,v} y_{u,v} & & \\
s.t. & \sum\limits_{(v,w)\in A} y_{v,w} & = 1 & \forall\, v \in V_S \\
 & \sum\limits_{(u,v)\in A} y_{u,v} & = 1 & \forall\, v \in V_S \\
 & \sum\limits_{v\in V_S} y_{s,v} & \leq k & \\
 0 \leq & y_{u,v} & \leq 1 & \forall\, (u,v) \in A
\end{array}
$$

4. OPTIMAL GENERAL OFFSET ASSIGNMENT

To solve GOA to global optimality, we need to find a memory layout that will allow us to create the best possible ARA.

The key observation that led to our exact approach is that the objective function is the only part where the memory layout influences the concrete ARA network problem to be solved. The cost of an access transition (u, v) in the network described in Sect. 3.2 is zero if and only if the variables associated with u and v are equal or neighbors in the memory layout. Otherwise, a positive cost c_A reflecting the overhead of an additional address arithmetic instruction is assigned.

We now combine this insight with the approach to model variable permutations from Sect. 2. Hence, we consider two graphs. Firstly, a complete graph $G = (V \cup \{z\}, E)$ where V is the set of program variables and z is an additional vertex as described in Sect. 2. Also, we again associate edge decision variables $x_{u,v} \in \{0, 1\}$ with the edges $\{u, v\} \in E$ that have no associated costs. Secondly, we have a network $N = (V_N, A)$ with $V_N = V_S \cup \{s, t\}$ where V_S is a vertex set related to the accesses contained in the input sequence S, just like in Sect. 3. Let $A_S = \{(v, w) \mid v, w \in V_S, v < w\}$ and $A = A_S \cup \{(s, v) \mid v \in V_S\} \cup \{(v, t) \mid v \in V_S\}$. We again use flow arc variables $y_{u,v}$ for each arc $(u, v) \in A$.

Since all access vertices in V_S are instances of the variables represented by the vertices V, we may define a corresponding unique mapping $\sigma : V_S \to V$. For ease of reference, we further split the set A_S into $A_S^{\neq} = \{(u, v) \in A_S,\ \sigma(u) \neq \sigma(v)\}$, i.e., the set of arcs between accesses that do not refer to the same associated variable and, analogously, the set $A_S^{=}$. Since G is undirected, the variables $x_{u,v}$ are only defined for $u < v$. Slightly disregarding mathematical precision, we write $x_{\sigma(u),\sigma(v)}$ when referring to the associated edge decision variable of $y_{u,v}, (u, v) \in A_S^{\neq}$ no matter whether $\sigma(u) < \sigma(v)$ or $\sigma(u) > \sigma(v)$.

4.1 Quadratic Formulation

When continuing straightforwardly with the just introduced relationships, we can now express the cost of an access transition $y_{u,v}$ with $(u, v) \in A_S^{\neq}$ as $(1 - x_{\sigma(u),\sigma(v)})c_A$. As already stated, the cost of each variable $y_{u,v}$ for $(u, v) \in A_S^{=}$ is zero. Putting everything into a single model yields the following formulation:

$$
\begin{array}{lll}
\min \sum\limits_{(u,v)\in A_S^{\neq}} (1 - x_{\sigma(u),\sigma(v)}) c_A y_{u,v} + \sum\limits_{v\in V_S} c_L y_{s,v} & & \\
\sum\limits_{\{u,v\}\in E} x_{u,v} & = 2 & \forall\, v \in V \cup \{z\} \\
x(E(W)) & \leq |W| - 1 & \forall\, \emptyset \neq W \subsetneq V \cup \{z\} \\
\sum\limits_{(u,v)\in A} y_{u,v} & = 1 & \forall\, u \in V_S \\
\sum\limits_{(u,v)\in A} y_{u,v} & = 1 & \forall\, v \in V_S \\
\sum\limits_{v\in V_S} y_{sv} & \leq k & \\
x_{u,v} & \in \{0, 1\} & \forall\, (u,v) \in E \\
y_{u,v} & \in \{0, 1\} & \forall\, (u,v) \in A
\end{array}
$$

This integer program is quadratic in its objective function. We may linearize it using the standard linearization approach. However, first we simplify. The term

$$\min \sum_{(u,v)\in A_S^{\neq}} (1 - x_{\sigma(u),\sigma(v)}) c_A y_{u,v}$$

can also be written as

$$\min \Big(\sum_{(u,v)\in A_S^{\neq}} y_{u,v} - \sum_{(u,v)\in A_S^{\neq}} x_{\sigma(u),\sigma(v)} y_{u,v}\Big) c_A.$$

We then need $|A_S^{\neq}|$ new variables $z_{u,v} = x_{\sigma(u),\sigma(v)} y_{u,v}$ and three linearization constraints for each of the new variables:

$$
\begin{array}{ll}
z_{u,v} & \leq x_{\sigma(u),\sigma(v)} \\
z_{u,v} & \leq y_{u,v} \\
z_{u,v} & \geq x_{\sigma(u),\sigma(v)} + y_{u,v} - 1
\end{array}
$$

After this transformation, the objective function becomes:

$$\min \sum_{(u,v)\in A_S^{\neq}} c_A y_{u,v} - \sum_{(u,v)\in A_S^{\neq}} c_A z_{u,v} + \sum_{v\in V_S} c_L y_{s,v}$$

4.2 Linear Formulation

Further inspection and the fact that there are only two cases for each arc $(u,v) \in A_S^{\neq}$, namely that it either has the assigned cost c_A or assigned cost zero, we found a way to linearize the problem inherently, that is without generating any products that need a subsequent linearization. The main idea is to replace every variable (arc) between two accesses $y_{u,v}, (u,v) \in A_S^{\neq}$ by two new variables (arcs) $y^0_{u,v}$ and $y^1_{u,v}$ reflecting the two mentioned cases. The set $A_S^{\neq}$ is therefore further split into the corresponding new arc sets A_S^0 and A_S^1. For every arc $(u,v) \in A_S^{=}$, we keep the former variable $y_{u,v}$ with zero cost as before. In the below IP formulation, we also skip the superscript when referring to flow variables disregarding their costs or if only one instance exists.

The new network N has now the arc set $A = A_S^0 \cup A_S^1 \cup A_S^{=} \cup \{(s,v) \mid v \in V_S\} \cup \{(v,t) \mid v \in V_S\}$. The new objective is of course to minimize the selected arcs with positive costs assigned. This is a linear expression in the set of variables. However, we now have to restrict the use of zero-cost arcs. As before in the quadratic model, it should only be possible to use them if the respective mapped variables are neighbors in the access sequence. With the newly introduced variables this can easily be enforced using the following constraints:

$$y^0_{u,v} \quad \leq x_{\sigma(u)\sigma(v)} \qquad \forall\ (u,v) \in A_S^0$$

Further, one may add also the following constraints to the model, since there is never a reason to use the cost-assigned arc in case that the variables are neighbors, i.e., the same access transition could be done without cost:

$$y^1_{u,v} \quad \leq 1 - x_{\sigma(u)\sigma(v)} \qquad \forall\ (u,v) \in A_S^1$$

However, this constraint will only have marginal impact on the solution process, since such a decision is either way not preferred due to the objective function.

The complete linear IP formulation is then:

$$\begin{array}{llll}
\min & \sum_{(u,v)\in A_S^1} c_A y^1_{u,v} + \sum_{v\in V_S} c_L y_{sv} & & \\
& \sum_{\{u,v\}\in E} x_{u,v} & = 2 & \forall\ v \in V \cup \{z\} \\
& x(E(W)) & \leq |W| - 1 & \forall\ \emptyset \neq W \subsetneq V \cup \{z\} \\
& \sum_{(u,v)\in A} y_{u,v} & = 1 & \forall\ u \in V_S \\
& \sum_{(u,v)\in A} y_{u,v} & = 1 & \forall\ v \in V_S \\
& \sum_{v\in V_S} y_{sv} & \leq k & \\
& y^0_{u,v} & \leq x_{\sigma(u)\sigma(v)} & \forall\ (u,v) \in A_S^0 \\
& y^1_{u,v} & \leq 1 - x_{\sigma(u)\sigma(v)} & \forall\ (u,v) \in A_S^1 \\
& x_{u,v} & \in \{0,1\} & \forall\ (u,v) \in E \\
& y_{u,v} & \in \{0,1\} & \forall\ (u,v) \in A
\end{array}$$

The number of variables is equal to the case of the quadratic formulation since essentially for each $(u,v) \in A_S^{\neq}$ the product variable $z_{u,v}$ is replaced by a second flow arc variable $y^1_{u,v}$. However, $|A_S^{\neq}|$ less constraints are needed.

4.3 The Branch-and-Cut Algorithms

We implemented Branch-and-Cut solvers for both the linear and quadratic formulation using CPLEX 12.1 [1] as the underlying ILP solver. We disabled its internal presolver but adopted all other default parameters. The optimization starts by relaxing integrality and taking all inequalities except the subtour elimination constraints (SECs) into account when solving the first LP. For the SECs and for 2-Matching-inequalities [21], we added exact separation procedures. Violation of the latter is checked whenever a fractional LP solution did not violate any SEC. If violated inequalities are found then they are added to the LP and it is solved again. Otherwise, if no violation is found, a *branching step* takes place, i.e., two new subproblems are created by fixing some fractional variable once to zero and once to one. The selection of the variable to branch on is left to CPLEX.

The LP solutions obtained at the respective Branch-and-Bound nodes will typically not be integral. Here, *primal heuristics* play an important role. Their purpose is to try to construct good feasible solutions by exploiting the current LP solution and therefore to improve the upper bound on the optimal objective function value. Our primal heuristic makes use of the condition that we can compute an optimal ARA for a given memory layout quickly. It follows the general idea that variables with an LP value close to one are likely to be part of a good or even optimal solution. In particular, we greedily construct a path cover by selecting edges $\{u,v\}$ iteratively based on the LP values of their corresponding variables $x_{u,v}$ as long as this is feasible. We use only edges from the set E' of edges where there is at least one corresponding access transition in the flow network $N = (V_N, A)$. The resulting paths are then concatenated to an offset assignment and then the network flow problem from Sect. 3.2 is solved to find an optimal ARA for the respective memory layout. We experimented with further weight criteria and tie-breaking concepts based on the flow variables but could not significantly improve over the results of this rather simple algorithm.

```
 1: function PRIMALHEURISTIC($G = (V, E')$, $x$, $N = (V_N, A)$)
 2:     SORT($E', x$)   # Sort edges non-increasingly w.r.t. $x$
 3:     INITIALIZEUNIONFIND($V$)
 4:     $n \leftarrow |V|$, $m \leftarrow |E'|$
 5:     $select \leftarrow \emptyset$, $count \leftarrow 0$
 6:     for $i = 1 \to n$ do
 7:         $deg(i) \leftarrow 0$                # Initialize degrees to zero
 8:     for $i = 1 \to m$ do
 9:         $e = \{u,v\} \leftarrow E'[i]$
10:         if $deg(u) < 2$ and $deg(v) < 2$ and $count < n$
11:         and (FIND($u$) $\neq$ FIND($v$) or $count = n - 1$) then
12:             $select \leftarrow select \cup \{e\}$, $count \leftarrow count + 1$
13:             $deg(u) \leftarrow deg(u) + 1$, $deg(v) \leftarrow deg(v) + 1$
14:             UNION($u, v$)
15:     $OA \leftarrow$ CONCATENATEPATHCOVER($select$)
16:     $ARA \leftarrow$ MINCOSTFLOW($N$, $OA$)
```

If $\mathcal{V}$ is the set of program variables and $|S|$ denotes the length of the access sequence, then both formulations have strictly less than $|\mathcal{V}|\cdot(|\mathcal{V}|-1)/2+|S|\cdot(|S|-1)$ variables. The number of constraints (after relaxing the SECs) is bounded by $|\mathcal{V}| + 2|S| + |S| \cdot (|S| - 1)$ Remarkably, all these numbers are independent from the number of ARs available. The more accesses in S coincide, the less (product or non-zero-cost arc) variables and associated constraints are needed. In contrast to that, the formulation by Ozturk et al. has $\mathcal{O}(k\cdot\mathcal{V}\cdot|S|+\mathcal{V}^2)$ variables and $\mathcal{O}(\mathcal{V}^3+k\cdot|S|\cdot\mathcal{V}^2)$ constraints (with larger constants) in the presence of k ARs.

5. EXPERIMENTAL SETUP

5.1 The OffsetStone Benchmark Set

The OffsetStone benchmark set consists of more than 3000 instances that have been extracted from 31 real-world application codes written in ANSI C. They comprise access sequences of lengths up to 3640 and referring to up to 1336 variables. Among them are computationally intensive programs (e.g., audio, video and image compression, Fourier transformation) as well as control-dominated applications (e.g., gzip). For details on how the instances were extracted, we refer to the original paper [14]. For our experiments, we only consider instances that consist of at least three program variables, these are 2785 in total. We perform the experiments once for 2, 4 and 8 ARs and assume the costs for address arithmetic instructions c_A and immediate AR loads c_R to be both equal to one.

5.2 Test System

Our experiments were run single-threaded with an Intel Core $i7$-3770T processor (2.5 GHz) on a Debian Linux system with 8 GB RAM, `g++` 4.7.2 and optimization level `-O2`. We measure the address computation overhead and average solution CPU times of five runs of the algorithms summarized in the following subsection.

5.3 Algorithms Included in the Evaluation

The following list summarizes the set of algorithms included in our experiments. A suffix `MCF` denotes that an optimal ARA is computed using the min-cost-flow technique (see Sect. 3.2) based on the respective memory layout. We implemented it using the network simplex algorithm provided by LEMON C++ library [6]. Since in `GOA-OFU-MCF` virtually no time is invested to create the memory layout, it can be used as a rough reference for its running time. We also reimplemented and fixed the ILP formulation of Ozturk et al. using the same CPLEX solver as for our new Branch-and-Cut algorithms.

- `GOA-OFU-MCF`: Creates a naive memory layout corresponding to the order of first use. This layout is then the basis for an optimal ARA. This can be seen as a reference for the impact of the ARA subproblem.
- `GOA-ITB-MCF`: Uses the most successful SOA heuristic `SOA-INC-TB` in [14, 12] to create a memory layout being then the basis for an optimal ARA.
- `GOA-SUGINO`: The heuristic by Sugino et al. with the *improved cost function* given in [25].
- `GOA-LM`: The GOA-heuristic presented in [16].
- `GOA-GA`: The genetic algorithm by Leupers and David [15] with an increased population size of 100 individuals.
- `GOA-OZTURK`: Our revised (details are given in the appendix) ILP formulation by Ozturk et al. [20].
- `GOA-QIP`: Our new Branch-and-Cut solver using the quadratic formulation from Sect. 4.1.
- `GOA-IP`: Like `GOA-QIP` but using the linear formulation from Sect. 4.2.

Each OffsetStone instance consists of one or multiple access sequences w.r.t. a particular set of variables. For algorithms that explicitly subdivide offset assignment and ARA, the latter case results in multiple ARA problems to be solved. Further, if the access sequences refer to disjoint subsets of program variables this is exploited and the exact solvers were started on the respective subsets of variables. The exact solvers were given a time limit of 10 seconds per call.

6. RESULTS

Fig. 6 shows the offset assignment costs relative to the optimal solution (left) and the running times (right) for all tested AR configurations. For each benchmark, we accumulated the running times and offset assignment costs for all instances of the benchmark that were solved to optimality by both `GOA-IP` and `GOA-QIP`. The number of these is given behind the benchmark names on the abscissa. The numbers of instances where one of the exact solvers timed out are listed in Tab. 2. With `GOA-IP` we could solve more instances than with `GOA-QIP` and, except for one single instance, `GOA-QIP` did not complete whenever `GOA-IP` did not either. However, if both algorithms timed out, there were cases where `GOA-QIP` provided better lower bounds, i.e., was closer to prove optimality of known solutions. Since the integer programs of `GOA-OZTURK` become very large (in the number of variables and, especially, constraints), it was able to solve less than half of the instances. Fig. 7-8 show the distribution of timeouts w.r.t. to the number of variables and access sequence lengths, respectively. We observe only a weak relationship between instance sizes and timeouts for our new solvers.

#ARs	GOA-IP	GOA-QIP	GOA-OZTURK
2	41 (1.48%)	52 (1.87%)	1610 (56.00%)
4	54 (1.94%)	69 (2.48%)	1742 (60.59%)
8	54 (1.94%)	78 (2.80%)	1887 (67.76%)

Table 2: Number of instances timed out after 10 seconds.

One major result of our experiments is that the gap between heuristic GOA solutions and optima is significant for many instances. This is in contrast to the results concerning SOA [12]. However, this is not surprising since the additional task of ARA largely increases the combinatorial complexity of the problem and adds a lot of symmetry. Another major result is that heuristics that try to find a good memory layout and perform an optimal ARA afterwards are clearly superior to those that perform ARA first or intertwined with memory layout computations. This becomes apparent since even a trivial layout with optimal ARA (`GOA-OFU-MCF`) performs often better. It is not clear whether this is mainly due to the inherent restriction to perform each access to a particular variable by the same AR since the heuristics evaluated have other weaknesses. For example, in the variable partitioning phase of `GOA-SUGINO`, there seems to be a significant discrepancy between the estimation delivered by the *improved cost function* [25] and the real cost of a solution. However, using the primary cost function given in the article, the heuristic would have timed out on many instances. `GOA-LM` uses all k available registers as soon as the access graph has at least k edges which might lead to avoidable immediate AR loads especially for small instances. Since OffsetStone comprises a lot of small instances, this problem is also apparent in the results for `GOA-GA` that uses `GOA-LM` for a starting solution and could not improve much on that with the parameters given in [15]. As a further matter, neither of these algorithms further subdivides the search space if an access graph has multiple connected components.

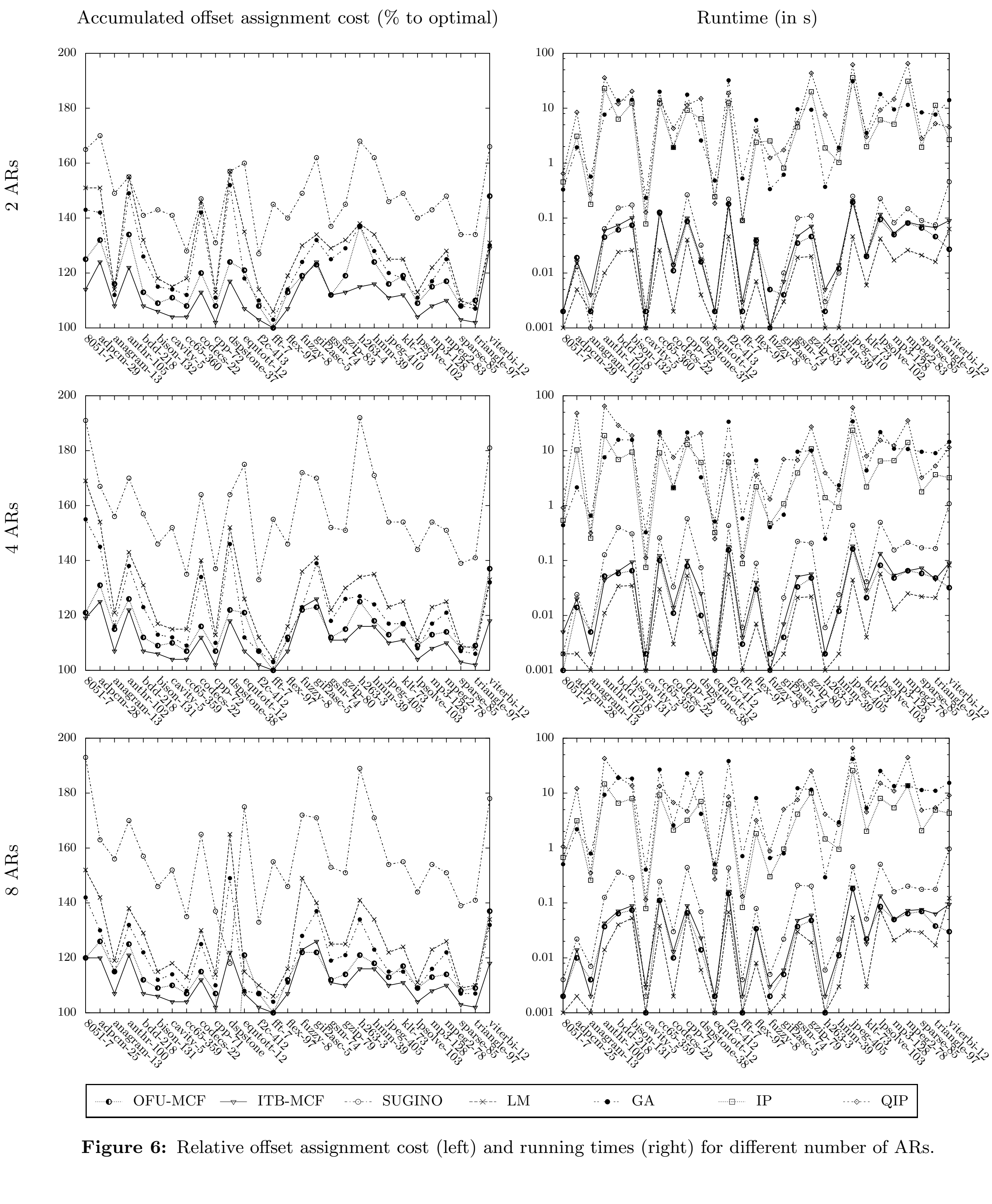


Figure 6: Relative offset assignment cost (left) and running times (right) for different number of ARs.

A further interesting result is that the sensitivity concerning the number of available ARs is low for almost all of the evaluated algorithms. Exceptions are `GOA-LM` which might encounter slight performance degradations due to the problem just mentioned and `GOA-OZTURK` that suffers from an exploding number of variables and constraints in the ILP formulation. The exact solvers presented in this paper encounter only slightly more difficulties with an increase in the number of ARs as can be seen from the number of timed out instances in Tab. 2. However, for most of the instances, their runtime overhead over the fast heuristics was acceptable even when considered for practical use.

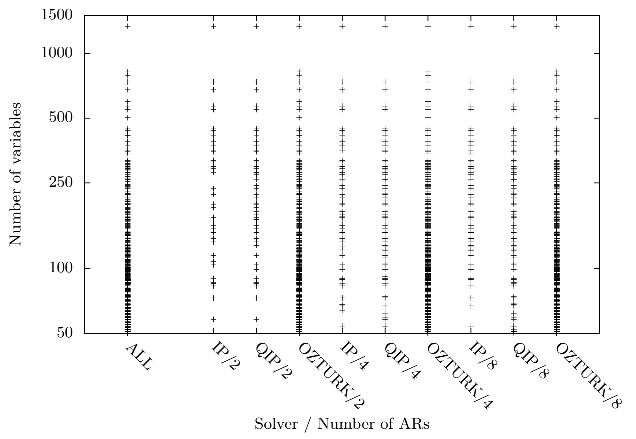


Figure 7: Timeouts per number of variables. The leftmost points reflect the distribution of all instances.

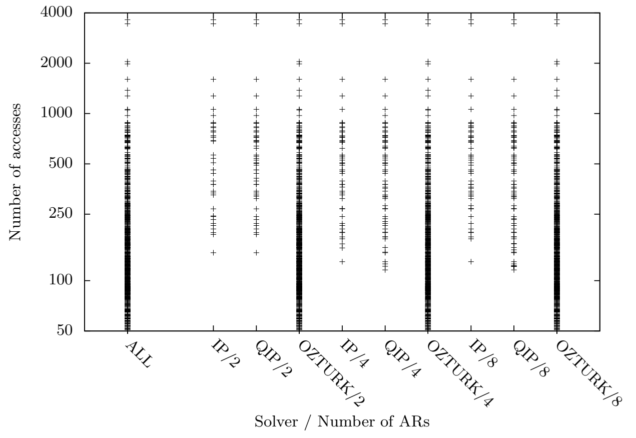


Figure 8: Timeouts per access sequence length. The leftmost points reflect the distribution of all instances.

7. CONCLUSION AND OUTLOOK

This paper has introduced a novel ILP approach to the General Offset Assignment problem that, for the first time, can solve 98% of the instances in OffsetStone to optimality, including non-trivial instances with a few hundred variables. Our experimental results unveil significant gaps between the solution quality of state-of-the-art General Offset Assignment heuristics and optimal solutions. For the large benchmark set considered, they are much more critical than in the case of Simple Offset Assignment. Further, the results underline the hypothesis that it is disadvantageous to perform address register assignment before offset assignment. The vast amount of instances that could be solved by our exact solver within a short time frame give hope that optimal or near-optimal solutions to the General Offset Assignment problem are realizable in practice. This is especially true since our primal heuristic often finds optimal or near-optimal solutions and much of the solving time is devoted to just prove that they are indeed optimal. In practice, exact techniques could lead to a much better exploitation of address generation units in application specific and digital signal processors. Based on the presented results, an exact approach could possibly be part of a real compilation process. An idea would be to combine it with a time limit and a fallback heuristic in order to protect against convergence problems for harder instances. Still there is a lot of improvement potential. The quadratic nature of the problem (the objective function) suggests to consider further solution techniques, i.e., to transform the problem into a maxcut problem or to formulate it as a semidefinite program. These approaches might result in better lower bounds and therefore be able to prove the optimality of found solutions more often than our solvers could. Since the subproblem of address register assignment is polynomial-time solvable even more sophisticated approaches like Bender's decomposition are candidate solution techniques. Another branch for further research is the question how to incorporate larger autoin-/decrement ranges and modify registers into a formulation that can handle a large set of instances.

8. ACKNOWLEDGMENTS

This research has been partially funded by the Swedish Research Council (VR 621-2011-6229). We thank Christian Schulte for helpful comments on the paper presentation.

9. REFERENCES

[1] CPLEX callable library version 12.1 C API. Reference manual, IBM ILOG, 2009.

[2] S. Atri, J. Ramanujam, and M. T. Kandemir. Improving offset assignment on embedded processors using transformations. In *Proc. 7th Int. Conf. on High Perf. Comput.*, HiPC '00, pages 367–374. Springer, 2000.

[3] D. H. Bartley. Optimizing stack frame accesses for processors with restricted addressing modes. *Softw. Pract. Exper.*, 22(2):101–110, 1992.

[4] Y. Choi and T. Kim. Address assignment combined with scheduling in DSP code generation. In *Proc. 39th Design Autom. Conf.*, DAC '02, pages 225–230. ACM, 2002.

[5] G. B. Dantzig, D. R. Fulkerson, and S. M. Johnson. Solution of a large-scale traveling-salesman problem. *Operations Research*, 3:393–410, 1954.

[6] B. Dezső, A. Jüttner, and P. Kovács. LEMON - an open source C++ graph template library. *Electron. Notes in Theor. Comp. Sc.*, 264(5):23–45, 2011.

[7] M. Elf, C. Gutwenger, M. Jünger, and G. Rinaldi. Branch-and-cut algorithms for combinatorial optimization and their implementation in ABACUS. In *Computational Combinatorial Optimization,*

Optimal or Provably Near-Optimal Solutions, volume 2241 of *LNCS*, pages 157–222. Springer, 2001.

[8] M. Eriksson. *Integrated Code Generation*. PhD thesis, Linköping University, 2011.

[9] C. H. Gebotys. DSP address optimization using a minimum cost circulation technique. In *Proc. 1997 IEEE/ACM Int. Conf. on Computer-Aided Design*, ICCAD '97, pages 100–103, 1997.

[10] C. H. Gebotys. A minimum-cost circulation approach to DSP address-code generation. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 18(6):726–741, 1999.

[11] J. Huynh, J. N. Amaral, P. Berube, and S. A. A. Touati. Evaluating address register assignment and offset assignment algorithms. *ACM Trans. Embedded Comput. Syst.*, 10(3):37, 2011.

[12] M. Jünger and S. Mallach. Solving the simple offset assignment problem as a traveling salesman. In *Proc. 16th Int. W. on Softw. and Compilers for Embed. Syst.*, M-SCOPES '13, pages 31–39. ACM, 2013.

[13] E. Lawler. *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart and Winston, 1976.

[14] R. Leupers. Offset assignment showdown: Evaluation of DSP address code optimization algorithms. In *Proc. 12th Int. Conf. on Compiler Constr.*, CC'03, pages 290–302. Springer, 2003.

[15] R. Leupers and F. David. A uniform optimization technique for offset assignment problems. In *Proc. 11th Int. Symp. on Syst. Synth.*, ISSS '98, pages 3–8. IEEE Comput. Soc., 1998.

[16] R. Leupers and P. Marwedel. Algorithms for address assignment in DSP code generation. In *Proc. 1996 IEEE/ACM Int. Conf. on Computer-Aided Design*, ICCAD '96, pages 109–112. IEEE Comput. Soc., 1996.

[17] S. Liao. *Code generation and optimization for embedded digital signal processors*. PhD thesis, 1996.

[18] M. Lorenz, D. Kottmann, S. Bashford, R. Leupers, and P. Marwedel. Optimized address assignment for DSPs with SIMD memory accesses. In *Proc. 2001 Asia and South Pacific Design Autom. Conf.*, ASP-DAC '01, pages 415–420. ACM, 2001.

[19] D. Ottoni, G. Ottoni, G. Araujo, and R. Leupers. Offset assignment using simultaneous variable coalescing. *ACM Trans. Embed. Comput. Syst.*, 5(4):864–883, Nov 2006.

[20] O. Ozturk, M. T. Kandemir, and S. Tosun. An ILP based approach to address code generation for digital signal processors. In G. Qu, Y. I. Ismail, N. Vijaykrishnan, and H. Zhou, editors, *ACM Great Lakes Symp. on VLSI*, pages 37–42. ACM, April 2006.

[21] M. W. Padberg and M. R. Rao. Odd minimum cut-sets and b-matchings. *Mathematics of Operations Research*, 7(1):67–80, 1982.

[22] A. Rao and S. Pande. Storage assignment optimizations to generate compact and efficient code on embedded DSPs. *SIGPLAN Not.*, 34(5):128–138, May 1999.

[23] H. Salamy and J. Ramanujam. An effective heuristic for simple offset assignment with variable coalescing. In *Proc. 19th Int. W. on Lang. and Compilers for Par. Comput.*, LCPC'06, pages 158–172. Springer, 2007.

[24] H. Salamy and J. Ramanujam. An ILP solution to address code generation for embedded applications on digital signal processors. *ACM Trans. Design Autom. Electron. Syst.*, 17(3):28:1–28:23, June 2012.

[25] N. Sugino, S. Iimuro, A. Nishihara, and N. Fujii. DSP code optimization utilizing memory addressing operation. *IEICE Transactions on Fundam. of Electr., Commu. and Comp. Sc.*, 79(8):1217–1224, 1996.

[26] É. Tardos. A strongly polynomial minimum cost circulation algorithm. *Combinatorica*, 5(3):247–255, 1985.

APPENDIX

A. REVISED MODEL OF OZTURK ET. AL

This appendix summarizes our changes made to the ILP model by Ozturk et al. [20] to capture GOA accurately. The asymptotic order of variables and constraints is not altered. The original model does not account for AR initialization costs. We repair this by adding a special variable v_M to the set of input program variables $\mathcal{V} = \{v_0, \ldots, v_{M-1}\}$ that is fixed to the artificial memory location M ($L_{v_M,M} = 1$). Instead of pointing 'nowhere' at program point $s = 0$ ($P_{r,v,0} = 0$, for all $r \in R$ and $v \in \mathcal{V}$), all variables now point to v_M ($P_{r,v_M,0} = 1$, for all $r \in R$) with the meaning that they are uninitialized. Initialization costs will be paid as soon as a register is moved away from v_M, since we let v_M be only within its own autoin-/decrement range, i.e., $R_{v_M,v_M} = 1$ and $R_{v,v_M} = 0$ ($R_{v_M,v} = 0$) for all $v \neq v_M$. For validity, v_M is also excluded from constraint (4) of the original model.

Constraint (4) forces transitions between adjacent variables v_1 and v_2 to have zero costs ($R_{v_1,v_2} = 1$). However, a constraint to let non-adjacent transitions have non-zero costs ($R_{v_1,v_2} = 0$) is missing. Hence, we add the constraints

$$R_{v_1,v_2} \leq 2 - L_{v_1,a} - L_{v_2,b}$$

for all $v_1, v_2 \in \mathcal{V} \setminus v_M$ and positions a, b such that $|a-b| > 1$.

At each program point $s \in S$, the variable *Sequence*(s) is accessed. Constraint (10) of the original model restricts the AR used for this access to be the *only* AR pointing to this variable at s. However, there is no reason why further ARs should not be able to point to the same address and it has not been proven whether such a restriction preserves optimality in each case. Thus we relax the constraint to

$$\textstyle\sum_{r \in R} P_{r,\mathit{Sequence}(s),s} \geq 1, \quad \forall\, s \in S.$$

To identify the AR responsible for a particular access we add variables $A_{r,s} \in \{0,1\}$ for each register $r \in R$ and program point $s \in S$ and the equations

$$\textstyle\sum_{r \in R} A_{r,s} = 1, \quad \forall\, s \in S.$$

Also, if AR r is used for the access at program point s, then r must hold the respective address. This is enforced by

$$P_{r,\mathit{Sequence}(s),s} \geq A_{r,s}, \quad \forall\, r \in R,\ \forall\, s \in S.$$

Unlike in the original model, an AR can only change the variable it points to if it is used in an access:

$$P_{r,v,s} \geq P_{r,v,s-1} - A_{r,s}, \quad \forall\, r \in R,\ \forall v \in \mathcal{V},\ \forall\, s \in S$$

To be precise, this constraint allows each AR to move *before* each performed access, not *after*, but it yields the same solutions and the interpretation is unambiguous.