Solving the Simple Offset Assignment Problem as a Traveling Salesman

Michael Jünger
Institut für Informatik
Universität zu Köln
Cologne, Germany
mjuenger@informatik.uni-koeln.de

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

ABSTRACT

In this paper, we present an exact approach to the Simple Offset Assignment problem arising in the domain of address code generation for digital signal processors. It is based on transformations to weighted Hamiltonian cycle problems and integer linear programming. To the best of our knowledge, it is the first approach capable to solve all instances of the established OffsetStone benchmark set to optimality within reasonable time. Therefore, it enables to evaluate the quality of several heuristics relative to the optimum solutions for the first time. Further, using the same transformations, we present a simple and effective improvement heuristic. In addition, we include an existing heuristic into our experiments that has so far not been evaluated with OffsetStone.

Categories and Subject Descriptors

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

General Terms

compiler optimization

Keywords

address code generation, compiler optimization, simple offset assignment, integer programming

1. INTRODUCTION

Address code generation is an important field for compiler optimizations since address calculations make up a significant part of machine codes. This is especially true for digital signal processors (DSPs) which frequently do not support implicit indirect addressing modes with arbitrary offsets. Instead they usually provide an address generation unit (AGU) with address registers (ARs) that need to be explicitly modified in order to compute memory access operands. Typically, modifications of an AR can be encoded into and performed with another instruction on that AR at no extra cost if the offset to the new address is within a certain processor-specific autoin-/decrement range. Otherwise, an explicit address arithmetic instruction is needed. Compilers may freely choose the stack layout for local variables of a function. It is therefore natural to ask for an optimization of the order of variable storage locations, such that subsequent memory accesses apply to locations that are within this range. Ideally, this may significantly reduce the code size and speed up the program at the same time. However, the range r for the mentioned autoin- or decrements is typically small. In fact, since many DSPs have a small instruction word length (e.g. 16 bits), they only allow autoin- or decrements by a single word, i.e., $r = 1$ [16, 14].

Given an access sequence of program variables, a range $r = 1$ and $k \geq 1$ ARs, the problem to find a memory layout that minimizes the address computation overhead is called the *General Offset Assignment* (GOA) problem. For $k = 1$, i.e. only a single AR, it is called the *Simple Offset Assignment* (SOA) problem and already NP-hard. Although SOA appears to be oversimplified, it reflects a real-world problem since GOA is typically solved by first assigning variables to ARs and then independently performing a SOA for each of the ARs. A comprehensive and up-to-date overview on how SOA and GOA correlate in practice and a discussion of the large impact of memory layouts on code size and performance was recently published by Huynh et al. [14]. In the paper at hand, we restrict our attention to SOA.

1.1 Motivating example

The input to the SOA problem is an access sequence of program variables. An access sequence is constructed from a scheduled order of three-address-code instructions $c = a\ op\ b$ by concatenating the accessed variables of each instruction in the order $a\ b\ c$.

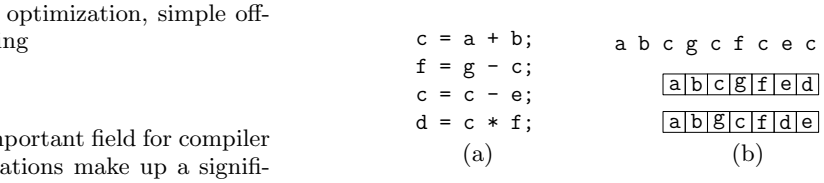


Figure 1: A sample code fragment (a), its access sequence and two memory layouts (b).

Performing this for the code fragment in Fig. 1 results in the access sequence shown on the right. Below, two example memory layouts are shown. The first one corresponds to the order of first use (OFU) of the variables and the second is an optimized one. Once the stack layout has been fixed, an address generator may add autoin-/decrement instructions whenever this is applicable. Look at Tab. 1 for a pseudo machine code of our exemplary code fragment. Increasing the use of the autoin-/decrement instructions (`*(AR)+` and `*(AR)-`) results in fewer explicit address arithmetic instructions (`ADAR` and `SBAR`). In this very small example, the optimized memory layout needs three less of them.

```
LDAR   AR, &a   AR = a
LOAD   *(AR)+   ACC = a,AR = b
ADD    *(AR)+   ACC += b,AR = c
STOR   *(AR)+   c = ACC,AR = g
LOAD   *(AR)-   ACC = g,AR = c
SUB    *(AR)    ACC -= c
ADAR   AR,2     AR = f
STOR   *(AR)    f = ACC
SBAR   AR,2     AR = c
LOAD   *(AR)    ACC = c
ADAR   AR,3     AR = e
SUB    *(AR)    ACC -= e
SBAR   AR,3     AR = c
STOR   *(AR)    c = ACC
ADAR   AR,2     AR = f
MUL    *(AR)    ACC *= f
ADAR   AR,2     AR = d
STOR   *(AR)    d = ACC
```

```
LDAR   AR, &a   AR = a
LOAD   *(AR)+   ACC = a,AR = b
ADD    *(AR)    ACC += b
ADAR   AR,2     AR = c
STOR   *(AR)-   c = ACC,AR = g
LOAD   *(AR)+   ACC = g,AR = c
SUB    *(AR)+   ACC -= c,AR = f
STOR   *(AR)-   f = ACC,AR = c
LOAD   *(AR)    ACC = c
ADAR   AR,3     AR = e
SUB    *(AR)    ACC -= e
SBAR   AR,3     AR = c
STOR   *(AR)+   c = ACC,AR = f
MUL    *(AR)+   ACC *= f,AR = d
STOR   *(AR)    d = ACC
```

Table 1: Pseudo machine codes for the OFU and optimized memory layouts for the code fragment from Fig. 1.

1.2 Our Contribution

In this paper, we present both - an integer linear programming (ILP) algorithm to solve the SOA problem to optimality and a simple and effective heuristic.

The main purpose of the exact algorithm is to deliver optimum solutions for the standard OffsetStone benchmark set [16] for the first time. This allows for a first real evaluation of the quality of existing heuristics. However, in contrast to existing ILP approaches for SOA and GOA, our algorithm solves the majority of instances within milliseconds of CPU time even though it has a quite simple design. Most of the time, it is much faster than the genetic algorithm from [17] which cannot guarantee an optimum solution. Still, an ILP solver that is capable to solve *any* instance in acceptable time cannot be expected. Nonetheless, we believe that an exact solver could be interesting in practice whenever the compilation of a particular program is carried out seldom or even only once. This is the case, e.g., for small devices' firmware to be stored in a small read-only memory. Another strategy could be to combine an exact solver with a time limit and a fallback heuristic.

As a second approach, we build a new improvement heuristic which provides a well-tunable trade-off between running time and solution quality. On the OffsetStone instances, it is capable to provide near-optimum solutions and can, e.g., produce better solutions than a previously proposed genetic algorithm [17] in less time.

In addition, we include the tie-break heuristic proposed by Ali et al. [2] in 2008 into our experiments. Before, it has only been tested on instances randomly generated by the authors.

The sequel of this paper is organized as follows. In Sect. 2, we report on related work and in Sect. 3, we recall the standard approach to solve the Simple Offset Assignment problem and its relations to path and path cover problems. Sect. 4 deals with the transformations necessary to apply our new algorithms, which we present in Sect. 5. We report on our experiments in Sect. 6 and 7 and, finally, the paper closes with concluding remarks in Sect. 8.

2. RELATED WORK

Simple Offset Assignment was first considered by Bartley [6] in 1992. He 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 [19] gave a formal proof for the strong NP-hardness of SOA, a simpler and faster heuristic (yielding the same results as Bartley's [16]) and also a first exact Branch-and-Bound procedure. In 1997, Leupers and Marwedel [18] proposed to use a tie-break function for edges with equal weight within Liao's heuristic. One year later, Leupers and David presented a genetic algorithm for GOA [17]. Atri et al. [5] developed an incremental algorithm that tries to successively improve a known feasible solution. These yet mentioned algorithms were subject to an exhaustive experimental comparison by Leupers [16] in 2003. It revealed only small differences in the quality of their solutions. However, the performance of the heuristics relative to the optimum solutions could only be verified for some small instances using Liao's Branch-and-Bound procedure. The corresponding benchmark set, called *OffsetStone*, which is freely available, is since then a standard reference for performance measures which we will also use in this paper.

Several research papers deal with integrated approaches or variants of the offset assignment problem. Ozturk et al. [24] provide an ILP approach for GOA with modify registers. Unfortunately, they did not evaluate their approach with OffsetStone. However, their ILP formulation is very generic, i.e., does not exploit polyhedral structure, has a very large number of variables and its running times do not allow for a direct use within a compiler. Similarly, Eriksson [12] proposed a dynamic programming algorithm to integrate scheduling, AR and offset assignment. However, the algorithm is highly time and memory consuming and far away from being capable to solve the SOA problem for all instances from OffsetStone.

Another research branch reflects the fact that storage locations can be shared by different program variables whose lifetimes do not overlap. The approach to group such variables is called *variable coalescing*. Ottoni et al. [23] presented a first heuristic which was followed by another one by Salamy and Ramanujam [26]. Recently, the latter authors also developed an ILP formulation for offset assignment with variable coalescing [27]. However, 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 [25] apply algebraic transformations to expression trees in order to find a least cost access sequence. Similarly, Atri et al. [4] use commutative transformations of the access pattern to obtain better solutions with existing heuristics. Choi and Kim [7] perform code transformations and reschedule parts of the code as a preprocessing step to offset assignment.

3. HAMILTONIAN PATHS, PATH COVERS AND SIMPLE OFFSET ASSIGNMENT

Typically, an instance of the SOA problem is modeled by an access graph $G = (V, E)$. The set of vertices V corresponds to the variables and there is an edge $e = \{u, v\} \in E$ with weight $w(e)$ if the variables u and v are neighbors within the access sequence for $w(e)$ times. Fig. 2 shows the access graph that corresponds to the sample code fragment shown in Fig. 1.

Let $G' = (V, E')$ be the complete graph that results by adding zero-weight edges between vertices that are not adja-

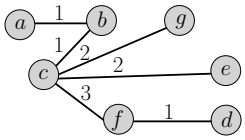


Figure 2: Access graph for the code fragment from Fig. 1.

cent in G. In the seminal paper [6] dealing with SOA, Bartley already noted (without proof) that the problem is equivalent to the Maximum Weight Hamiltonian Path (MWHP) problem in G'. Liao [19] showed the NP-hardness of SOA by reducing the decision variant of the Hamiltonian path problem to that of SOA. In view of the fact that the reduction to the MWHP problem requires the addition of many zero-weight edges, he reduced SOA to the Maximum Weight Path Cover (MWPC) problem instead. In this light, the majority of algorithms developed for SOA are heuristics to solve the MWPC problem. For our algorithms, we stay closer to the initial MWHP-oriented idea of Bartley. We recall formal definitions of the two problems.

DEFINITION 1. *(Maximum Weight Path Cover Problem) Given a graph $G = (V, E)$ and a weight function $w : E \to \mathbb{Z}$, compute a set of disjoint paths $\mathcal{P}$ such that each vertex is visited by exactly one path $P \in \mathcal{P}$ and the sum of the weights of the selected edges, $\sum_{P \in \mathcal{P}} \sum_{e \in P} w(e)$, is maximum.*

DEFINITION 2. *(Maximum Weight Hamiltonian Path Pr.) Given a complete graph $G = (V, E)$ and a weight function $w : E \to \mathbb{Z}$, compute a path of maximum weight that visits each vertex $v \in V$ exactly once.*

Def. **??** allows for isolated vertices considered as paths of length zero. In fact, the only difference between the two problems is that a collection of disjoint paths covering all vertices is feasible for the MWPC whereas it is not for the MWHP. Clearly, any concatenation of these paths to a memory layout is also a feasible solution for SOA.

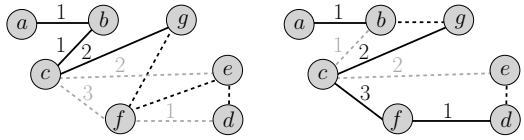


Figure 3: The path covers (solidly drawn), paths (with dashed black) and uncovered edges (dashed gray) corresponding to the two solutions from Fig. 1.

The cost of a solution expressed by a path cover is the sum of the edge weights that are *not* in the cover. It corresponds to the number of address computations that cannot be done by autoin- or decrements. Given a MWPC consisting of multiple paths, permuting them has no effect on this value. They can be concatenated to a memory layout in an arbitrary order. This is an important result that we should formalize.

LEMMA 1. *Let $G = (V, E)$ be an access graph and $\mathcal{P}$ be an optimal disjoint path cover for G, $|\mathcal{P}| > 1$. For any two end-vertices p, q of different paths $P, Q \in \mathcal{P}$, the number of access transitions between p and q is zero.*

This result can be easily verified. If there were access transitions between p and q, then there must be an edge $e = \{p, q\}$ in the access graph with a weight $w(e) > 0$. Adding this edge to the solution of the MWPC (and therefore simply connecting the two paths) would improve the solution which was assumed to be optimal.

It is therefore viable for our purposes to transform an access graph $G = (V, E)$ into a complete graph $G' = (V, E')$ by adding zero-weight edges between vertices that are not adjacent in G. Using this simple extension it is easy to show the following important relationship between the MWPC and the MWHP.

THEOREM 1. *Let $G = (V, E)$ be an undirected graph and $G' = (V', E')$ the complete graph that results by adding a zero-weight edge for every edge that is not in G. Then there exists a maximum-weight path cover P of weight $w(P)$ in G if and only if there exists a maximum-weight Hamiltonian path P' of weight $w(P') = w(P)$ in G'.*

PROOF. Let P be a maximum-weighted path cover in G. Clearly, if P consists only of a single path, then P is also a maximum-weight Hamiltonian path. So let P consist of $k > 1$ disjoint paths. By Lemma 1, there exists an Hamiltonian path P' in G' that consists of P and $k - 1$ additional edges with zero weight. Hence, P' has the same weight as P. So suppose now that P' is not a maximum-weighted Hamiltonian path in G', i.e., there exists a different path Q with weight $w(Q) > w(P')$. However, then Q, without its zero-weight edges, is also a maximum-weight path cover in G with weight greater than $w(P)$. This is a contradiction to the assumption that P is maximum. Conversely, a maximum-weight Hamiltonian path P' in G' yields (by removing zero-weight edges) directly a path cover P of the same weight, and there cannot be a better one, because this would contradict the optimality of P'. □

4. FROM OFFSET ASSIGNMENTS TO TRAVELING SALESMEN

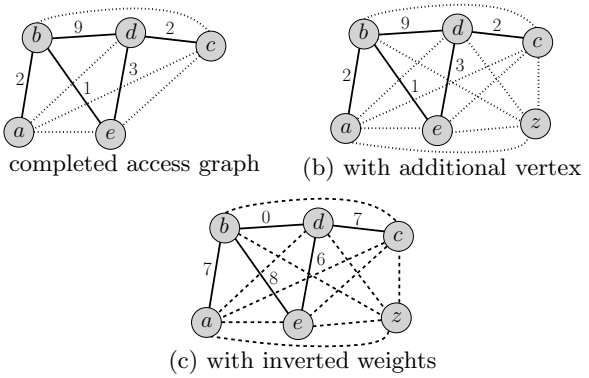
Theorem 1 allows us to calculate an optimum solution to the SOA problem by finding a maximum-weight Hamiltonian path in a graph created from the access graph as described above. Once formulated like this, it can be easily transformed into a *Maximum Weight Hamiltonian Cycle* (MHWC) problem and, by turning the maximization objective into a minimization objective, into the Traveling Salesman Problem (TSP).

DEFINITION 3. *(Maximum Weight Hamiltonian Cycle Pr.) Given a complete graph $G = (V, E)$ and a weight function $w : E \to \mathbb{Z}$, compute an Hamiltonian cycle (a tour) $T \subseteq E$ of maximum weight visiting each vertex $v \in V$ exactly once.*

We will now describe how to transform an instance of the MWHP (SOA) problem into an instance of the MWHC problem and into an instance of the TSP, respectively.

4.1 Problem Transformations

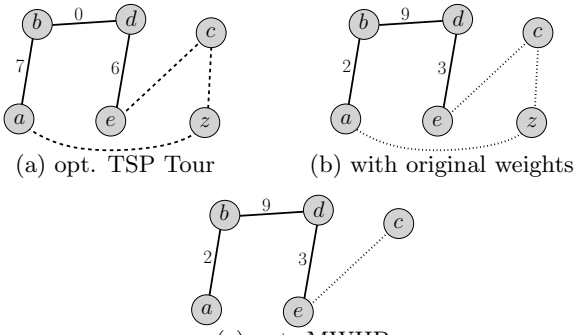
Let $G = (V, E)$ be the completed access graph for which we wish to find an Hamiltonian path of maximum weight. We create another graph $G_C = (V_C, E_C)$ as follows. We set $V_C = V \cup \{z\}$ where z is an additional vertex. E_C consists of E and additional edges $\{v, z\}$ with zero weight for every vertex $v \in V$. Computing a (maximum-) minimum-weighted



(a) completed access graph (b) with additional vertex

(c) with inverted weights

Figure 4: Transforming an instance of the MWHP problem into an instance of the TSP (dotted edges have zero weight, their dashed counterparts maximum weight nine).



(a) opt. TSP Tour (b) with original weights

(c) opt. MWHP

Figure 5: Interpreting the optimum TSP or MWHC tour to obtain a maximum-weight Hamiltonian path.

tour in G_C and removing the vertex z from the cycle yields a (maximum-) minimum-weighted Hamiltonian path in G. In order to solve the problem as a TSP instance, we need to account for the objective function. By computing $w_{max} = \max\{w(e) \mid e \in E_C\}$, we obtain the correct input graph for the TSP by reassigning all weights such that $w(e) = w_{max} - w(e)$. In a slight abuse of language, we refer to the last step as 'inversion' of edge weights.

Clearly, the transformations can be done in linear time with respect to the size of the complete graph, i.e., in $\mathcal{O}(|V|^2)$ time. They yield much more than just another strongly NP-hard problem. Due to the popularity and importance of the TSP, there is a vast research effort in the area of combinatorial optimization that has been invested to solve it. Instead of formulating some generic ILP, we can profit from the profound knowledge that has been published about the polytope corresponding to Hamiltonian cycles, e.g., known (facet-defining) inequalities and separation algorithms. However, even when linear programming is not applicable for implementation in practice, the transformation to the MWHC or TSP can help to obtain better solutions. For instance, it is then possible to apply well-performing TSP heuristics, such as, e.g., the algorithm proposed by Lin and Kernighan [21].

4.2 An ILP for the MWHC problem

Let $G = (V, E)$ be a complete graph on n vertices and let $x_e \in \{0, 1\}$ be a decision variable for each edge $e \in E$ expressing whether it is selected for the tour or not. For each edge (variable), let c_e denote the associated edge weight (cost). Further, we denote the set of edges adjacent to a vertex $v \in V$ with $\delta(v)$ (the star graph of v) and similarly, for any set of vertices W, the set of edges between the vertices in W with $E(W)$. Let $x(S) = \sum_{e \in S} x_e$ for any subset $S \subseteq E$. Then, an integer programming formulation for the MWHC can be stated as follows:

$$
\begin{array}{lllll}
\max & \sum\limits_{e \in E} c_e x_e & & & \\
& x(\delta(v)) & = 2 & & \forall\, v \in V \\
& x(E(W)) & \leq |W| - 1 & & \forall\, \emptyset \neq W \subsetneq V \\
& x_e & \in \{0, 1\} & & \forall\, e \in E
\end{array}
$$

The objective function is to maximize the total cost of the selected edges. The equations force any vertex to be adjacent to exactly two other vertices in the tour. Their number is linear in $|V|$. The inequalities are the well-known subtour elimination constraints (SECs) [9] which exclude solutions with multiple cycles from the feasible set. Their number is exponential in $|V|$, so they are usually not added to the problem formulation from the beginning but separated instead. Finally, the last row enforces integrality of the decision variables. Again, changing the objective into a minimization one yields an ILP formulation of the TSP. Many more valid (and facet-defining) inequalities for the polytope of Hamiltonian cycles are known and used in sophisticated solvers, such as, e.g., the comb inequalities [8]. For a comprehensive (polyhedral) study of the TSP, we refer to [3].

4.3 The Lin-Kernighan TSP heuristic

The TSP heuristic proposed by Lin and Kernighan [21] tries to successively improve a given initial tour by exchanging edges. The basic procedure maintains a set of marked vertices and tries to find improving sequences of edge flips starting from one of the marked vertices. A usual approach is to iteratively start the heuristic on different starting tours as long as some limit of computation time is not exceeded [3]. A more sophisticated approach proposed by Martin and Otto [22], called *Chained Lin-Kernighan*, is to perturb the tours obtained by one run of the basic procedure instead of creating new tours from scratch. The perturbations are called *kicks*, each time selecting k edges to be swapped with k being typically within the range $[2, 4]$. Another successful modification of the initial basic procedure has been proposed by Helsgaun [13] and implemented in his *LKH*-algorithm. An empirically [3] observed property of Lin-Kernighan heuristics is that it is often possible to trade quality for running time, i.e., if the number of allowed kick attempts is large then solutions are often close to optimal.

5. NEW ALGORITHMS

In principle, the presented transformation immediately enables the use of sophisticated TSP solvers to solve the SOA problem. However, in this paper, we rather want to give evidence that real-world instances can be quickly solved to optimality with a much simpler solver that could indeed be part of a compiler. Further, we construct a new improvement heuristic by combining a greedy MWPC heuristic with the Lin-Kernighan TSP heuristic.

5.1 An exact Branch-and-Cut algorithm

`SOA-MWHC` is a rudimentary ILP solver for the MWHC problem that we implemented using the Branch-and-Cut-Framework ABACUS [11] and CPLEX 12.1 [1] as linear program (LP) solver. It basically solves the ILP formulation introduced in Sect. 4.2 and uses the transformation with the additional vertex as described in Sect. 4.1. We give a high-level description of the procedure, for a detailed discussion of Branch-and-Cut algorithms we refer to [11].

The algorithm starts by relaxing the integrality and subtour elimination constraints, resulting in an LP consisting of variables $x_e \in [0,1]$ for each edge e and only the degree equations. Then the following iterative process is applied to the solution x^* after solving each LP: First, it is determined whether x^* violates any of the yet neglected SECs by computing a minimum cut (yielding a most violated SEC if x^* has fractional components) or by finding connected components (if x^* is integral). If no SEC can be found and x^* has fractional components, we check for violated 2-Matching-inequalities [10] using the algorithm proposed in [15]. If violated inequalities are found like this then they are added to the LP (as 'cutting planes') and it is solved again. Otherwise, if no violation is found or 50 iterations passed like this, a *branch* takes place, i.e., two new subproblems are created by fixing some $0 < x^*_e < 1$ once to zero and once to one. If, at any point of the procedure, a solution x^* is integral and does not violate any SEC, it corresponds to a tour. So if its weight cx^* is better than the best previously known feasible solution, we update it accordingly. The algorithm terminates if provably no better solution than the currently best known can be found in any open subproblem. The successive addition of cutting planes strengthens the LP relaxation which leads to better upper bounds on the optimum objective function value. In order to improve the lower bound and to obtain good tours quickly, a *primal heuristic* is run after the solution of each LP.

```
1: function primalHeuristic
2:     heap1 ← ∅, heap2 ← ∅, select ← ∅, count ← 0
3:     for e = 1 → m do
4:         if x_e > 0 and w_e > 0 then
5:             heap1.insert(e, w_e · x_e)
6:         else if w_e > 0 then
7:             heap2.insert(e, w_e)
8:     while ¬heap1.empty() and count < n do
9:         Edge e = (u, v) ← heap1.extractMax()
10:        if selection of e is feasible then
11:            select ← select ∪ {e}, count ← count + 1
12:    while ¬heap2.empty() and count < n do
13:        Edge e ← heap2.extractMax()
14:        if selection of e is feasible then
15:            select ← select ∪ {e}, count ← count + 1
16:    Extend select to a tour by adding zero-weight edges
17:    return select
```

It is likely that edges with a high LP value are part of a good or even optimum solution. Hence, our primal heuristic works as follows: We use two heaps. The first one contains the edges e with $w_e > 0$ and $x_e > 0$ in non-increasing order of $w_e \cdot x_e$. The second one contains those edges e that have LP value $x_e = 0$ but strictly positive weight (in non-increasing order, too). We then try to select edges one-by-one from both heaps (prioritizing the first one) as long they do not close a cycle and their end-vertices have degree less than two (feasibility check in lines 10 and 14). This process might lead to a partial solution, in fact a path cover, which we then arbitrarily concatenate to a tour using zero-weight edges. If its weight is greater than the currently best known one, we update the best feasible solution and the lower bound.

For reproducibility, we list those ABACUS parameters that we did not leave on their default values.

Parameter	value
`NBranchingVariableCandidates`	10
`NStrongBranchingIterations`	10
`ObjInteger`	`true`
`MaxIterations`	50

Table 2: Manually set ABACUS parameters.

5.2 A new improvement heuristic

As an alternative to profit from the transformation to the TSP without the need for linear programming, we combine two combinatorial algorithms with each other. In particular, we use Liao's algorithm with Leupers' and Marwedel's tie-break function (called `SOA-TB` in the experiments) to find an initial MWPC. The resulting offset assignment is a concatenation P of disjoint paths. Then, a complete graph $G' = (V', E')$ with $V' = V \cup \{z\}$ is created as described in Sect. 4.1. We invert the edge weights and append z to P interpreting the result as a tour starting and ending in z. This tour then serves as a starting solution for the Chained-Lin-Kernighan algorithm, as described in Sect. 4.3. After obtaining a new solution, z is removed, yielding an Hamiltonian path again from which we can easily construct the respective offset assignment. We call this procedure `SOA-TBLK` in our experiments. It can be considered a new improvement heuristic similar to the combination of the incremental algorithm by Atri et. al [5] with `SOA-TB` (called `SOA-INC-TB`) [16]. The following is a high-level description of the algorithm:

```
1: function SOA-TBLK(AccessGraph G = (V, E))
2:     path ← SOA-TB(G)
3:     z ← new Vertex
4:     G' = (V', E') ← new CompleteGraph(G, z)
5:     invert the weights of E'
6:     tour ← path ∪ {z}
7:     tour ← ChainedLinKernighan(G', tour)
8:     path ← tour \ {z}
9:     return offset assignment corresponding to path
```

For our experiments, we use the implementation of Chained-Lin-Kernighan within the Concorde TSP solver library [3]. In this version, four edges are switched per kick (so-called double-bridge kicks). It can be called with a few parameters besides the initial tour, namely a list of edges considered 'good' (which we do not provide) as well as limits on the number of kicks to perform without finding a better tour and in total. Further, one may specify the type of kicks to use. We allow at most $m \log m$ kicks in total (with $m = \binom{|V'|}{2}$) and set the kick type to 'random' using 4711 as seed.

6. EXPERIMENTAL SETUP

Since we are now able to evaluate the performance of heuristics relative to optimum solutions for all instances of the OffsetStone benchmark set, we basically repeat the ex-

periments of Leupers [16]. However, besides replacing Liao's Branch-and-Bound algorithm by our optimal Branch-and-Cut algorithm and adding the SOA-TBLK heuristic, we also implemented the tie-break heuristic of Ali et al. [2] that has so far only been tested on random instances generated by the authors. We did not repeat the experiments for Bartley's heuristic, since Leupers already showed that Liao's heuristic produces the same results faster [16].

6.1 The OffsetStone benchmark set

The OffsetStone benchmark set consists of more than 3000 realistic SOA instances that have been extracted from 31 real-world application codes written in ANSI C. They comprise access sequences with up to 1336 variables and lengths from 10 to 3640. 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 more details on how the instances were extracted, we refer to the original paper [16]. We also give the results for the random instances that have been used in this publication.

Each instance consists of one or multiple access sequences which may refer to disjoint sets of program variables. Thus, the access graphs of some of the instances have multiple connected components. In this case, our exact solver was started on each of the components and the resulting offset assignments were concatenated. For our benchmarks, we only considered instances that consist of at least 10 variables.

6.2 Test system

Our experiments were run with an Intel Core i7 960 processor (3.2 GHz) on a Debian Linux system with 6 GB RAM. We measure the offset assignment costs and average solution CPU times of five runs of the algorithms summarized in the following subsection. The time to create the access graphs and all necessary transformations within the Hamiltonian cycle oriented codes are considered part of the algorithms and contribute to the measured CPU times.

6.3 Algorithms included in the evaluation

- SOA-Liao: The heuristic presented in [19, 20].
- SOA-TB: Liao's heuristic extended by the tie-break heuristic of Leupers and Marwedel [18].
- SOA-TB2: Like SOA-TB but with the new tie-break heuristic of Ali et al. [2].
- SOA-INC: The incremental SOA-algorithm of Atri et al. [5] using SOA-Liao for an initial solution.
- SOA-INC-TB: Like SOA-INC but using SOA-TB for an initial solution.
- SOA-TBLK: The combination of SOA-TB with the Lin-Kernighan heuristic for TSPs presented in Sect. 5.2.
- SOA-GA: The genetic GOA-algorithm of Leupers and David [17].
- SOA-MWHC: The optimal MWHC-based solver presented in Sect. 5.1.

7. RESULTS

The most important result is that, if we sum up the offset assignment costs for all access sequences of each benchmark, all tested heuristics deliver solutions that are within 8.5% of the optimum value. This relative average performance is visualized in Fig. 6. A comprehensive list that summarizes the average solution quality as well as running times (rounded to milliseconds) can be found in Table 4. One reason for the small average deviation is that there are a lot of instances with only a few variables where all algorithms find optimum assignments. Still, considering single instances and absolute numbers, the differences can be quite large. This becomes visible especially in the anthr, cavity, gsm, mp3 and mpeg2 benchmarks. We list some interesting single access sequences that had a larger deviation or make up significant parts of the overall benchmark running time in Table 3.

Concerning the previously known heuristics, Leupers [16] already found that SOA-INC-TB performs best on the OffsetStone instances. With respect to the accumulated costs, its deviation from the optimum value is never more than 3%. Considering single access sequences with more than 50 variables, we recognized instances with up to 10% deviation for SOA-INC-TB and also SOA-TBLK. Since SOA-TBLK and SOA-INC-TB are both initialized with the solution of SOA-TB, they can never produce worse solutions and are especially interesting to be compared to each other. As could be expected, the large number of allowed kicks lets SOA-TBLK usually improve more starting solutions than SOA-INC-TB, but this is paid for with higher running times as can be seen in Table 4. Typically, the differences between the costs produced by both algorithms are small, however for several benchmarks, SOA-TBLK is the only heuristic with optimal or close-to-optimal solutions. Furthermore, it seldom performs worse than SOA-GA that sometimes produces slightly varying results in different runs due to the use of randomly generated numbers. Another interesting comparison is between the two tie-break-heuristics. In our experiments, the more recent tie-break heuristic of Ali et al. [2] (SOA-TB2) is nearly always inferior to the one by Leupers and Marwedel [18] (SOA-TB). This is also true for the randomly generated instances, whereas Ali et al. reported better results on their own randomly generated ones [2]. In fact, only for anagram and dct_unrolled it performed slightly better.

Let us now focus on the running times. First of all, those of all previously known heuristics and also SOA-TB2 are negligible and could sometimes even not be measured. We should mention here that we implemented SOA-TB2 directly into the OffsetStone code which does not provide a real notion of adjacency lists. Hence, SOA-TB2 could be implemented more efficiently. Although the heuristics are very fast, the overhead of the exact solver is, in most of the cases, small and usually acceptable. It is often much faster than SOA-GA and SOA-TBLK. In these cases, the bounds of the LP relaxation are strong and the primal heuristic helps to quickly find (optimum) integral solutions. For some large instances (especially mp3 86), the running time of the exact solver was dominated by the construction of the constraint matrix of the linear program and its solution.

As expected, also SOA-TBLK is faster than SOA-GA on most instances. However, it is much slower than all other heuristics. It provides an alternative to SOA-GA and an exact solver if genetic algorithms or linear programming are not applicable since its running times never peak in an extreme manner. Reducing the number of kicks to further reduce the running time of SOA-TBLK, however, would cause it to improve far less of the starting solutions obtained from SOA-TB.

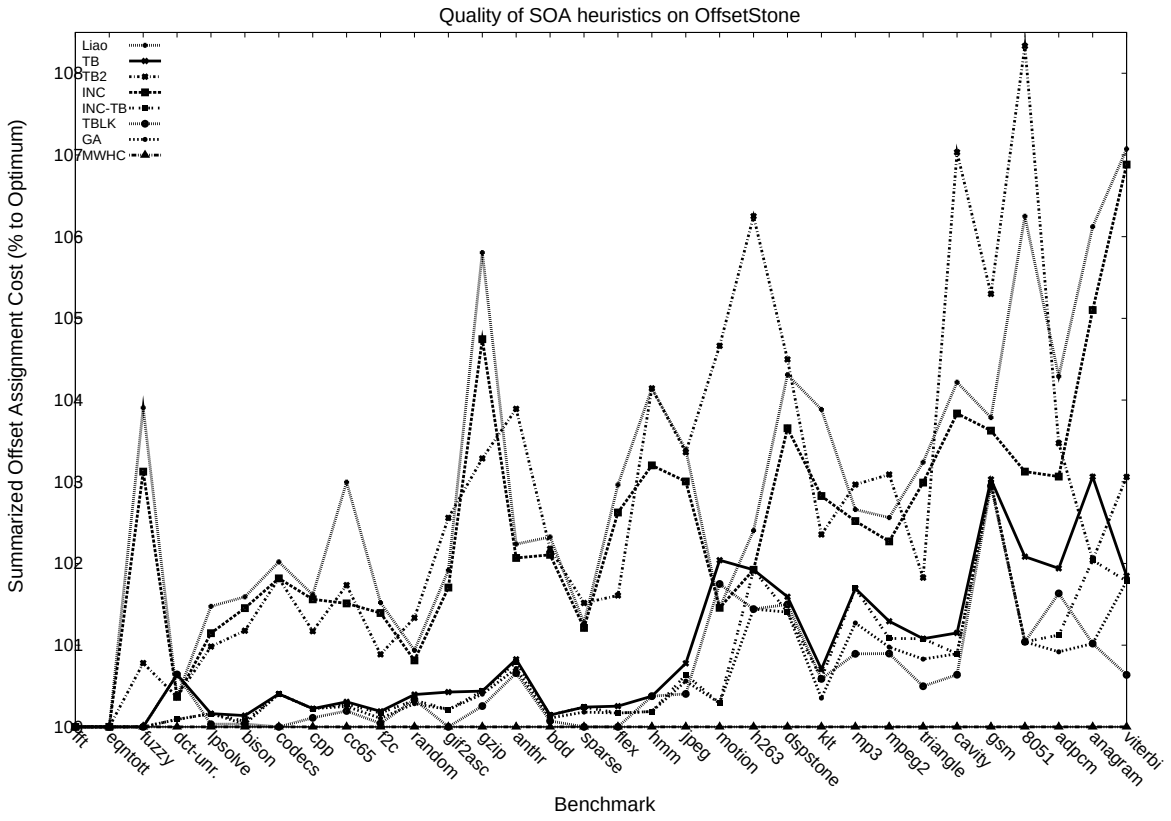


Figure 6: Summarized relative offset assignment cost of all tested algorithms on the OffsetStone instances.

instance	#vars	OFU	Liao	TB	TB2	INC	INC-TB	TBLK	GA	MWHC	time
`gsm` 22	437	260	260	260	260	260	260	260	260	237	0.496
`gsm` 28	446	260	260	260	260	260	260	260	260	237	0.436
`mp3` 86	1336	2893	1966	1983	2003	1966	1983	1953	1966	1918	40.044
`mpeg2` 80	313	415	344	343	344	344	343	343	343	336	0.146
`viterbi` 6	788	783	709	671	677	709	671	661	671	657	0.074
`viterbi` 8	820	786	718	678	683	718	678	670	678	666	0.060

Table 3: Some single instances with larger gaps (the last column displays the average CPU time needed by `SOA-MWHC`).

8. CONCLUSION

We presented a solution strategy for the Simple Offset Assignment problem by transforming it into an Hamiltonian cycle problem. In our experiments it became evident that the instances of the OffsetStone benchmark can be solved with a rudimentary Branch-and-Cut solver in acceptable time. We also made some experiments with the Concorde TSP solver [3] and compared the results to ours. As could be expected, on some of the larger and denser instances, the sophisticated solver finished even faster. However, the overall performance of our simple solver was comparable since the typical structure of the access graphs contained in the benchmark set seems to be not too difficult. Although OffsetStone is considered to reflect real world instances, it would be interesting to know whether it really covers typical challenges for today's address code generators or whether different types of instances occur in practice. This is also interesting with respect to the results that we could obtain for the relative quality of existing heuristic algorithms. On the OffsetStone instances they provided solutions that are within 8.5% of the optimum offset assignment cost. Apart from that, we can mainly confirm the results obtained by Leupers [16] in that `SOA-INC-TB` performs best on most instances. If higher running times are acceptable, the results can be slightly improved using the presented `SOA-TBLK` algorithm that combines Leupers' and Marwedel's tie-break version [18] of Liao's heuristic with the Lin-Kernighan TSP heuristic. Further, the tie-break-function proposed by Ali et al. [2] could not be verified to produce better results than the mentioned one.

Our results give hope that optimal or near-optimal solutions to the Simple Offset Assignment problem are realizable in practice. This is especially true since solving the first LP and running the primal heuristic within `SOA-MWHC` on the (usually fractional) LP solution (as described in Sect. 5.1) already produces competitive offset assignments. Considered as a building block for more complicated and more realistic General Offset Assignment scenarios, this could lead to a much better exploitation of address generation units in embedded digital signal processors. Based on the pre-

sented 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. Concerning TSP-oriented heuristics, there is also room for improvements, e.g., by experimenting with the LKH implementation of Helsgaun [13].

9. REFERENCES

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

[2] H. S. Ali, H. M. El-Boghdadi, and S. I. Shaheen. A new heuristic for SOA problem based on effective tie break function. In *Proc. of the 11th Intern. Workshop on Softw. and Compilers for Embed. Syst.*, SCOPES '08, pages 53–59, New York, NY, USA, 2008. ACM.

[3] D. L. Applegate, R. E. Bixby, V. Chvátal, and W. J. Cook. *The Traveling Salesman Problem: A Computational Study*. Princeton University Press, Princeton, NJ, USA, 2006.

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

[5] S. Atri, J. Ramanujam, and M. T. Kandemir. Improving offset assignment for embedded processors. In *Proc. of the 13th Intern. Workshop on Lang. and Compilers for Parallel Comput.*, LCPC '00, pages 158–172, London, UK, 2001. Springer.

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

[7] Y. Choi and T. Kim. Address assignment combined with scheduling in DSP code generation. In *Proc. of the 39th Design Automation Conf.*, DAC '02, pages 225–230, New York, NY, USA, 2002. ACM.

[8] V. Chvátal. Edmonds polytopes and weakly hamiltonian graphs. *Math. Progr.*, 5:29–40, 1973.

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

[10] J. Edmonds. Maximum matching and a polyhedron with 0,1-vertices. *J. of Research of the National Bureau of Standards*, 69B(1-2):125–130, 1965.

[11] M. Elf, C. Gutwenger, M. Jünger, and G. Rinaldi. Branch-and-cut algorithms for combinatorial optimization and their implementation in ABACUS. In *Comput. Comb. Opt., Optimal or Provably Near-Optimal Solutions*, volume 2241 of *LNCS*, pages 157–222, London, UK, 2001. Springer.

[12] M. Eriksson. *Integrated Code Generation*. PhD thesis, Linköping University, Dept. of Computer and Inform. Science, The Institute of Technology, 2011.

[13] K. Helsgaun. An effective implementation of the lin-kernighan traveling salesman heuristic. *European Journal of Operational Research*, 126:106–130, 2000.

[14] 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.

[15] A. N. Letchford, G. Reinelt, and D. O. Theis. A faster exact separation algorithm for blossom inequalities. In *Proc. of the 10th Intern. Conf. on Integer Programming and Comb. Opt. IPCO'04*, volume 3064 of *LNCS*, pages 196–205. Springer, 2004.

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

[17] R. Leupers and F. David. A uniform optimization technique for offset assignment problems. In *Proc. of the 11th Intern. Symp. on Syst. Synth.*, ISSS '98, pages 3–8, Washington, DC, USA, 1998. IEEE Computer Society.

[18] R. Leupers and P. Marwedel. Algorithms for address assignment in DSP code generation. In *Proc. of the 1996 IEEE/ACM Intern. Conf. on Computer-Aided Design*, ICCAD '96, pages 109–112, Washington, DC, USA, 1996. IEEE Computer Society.

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

[20] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, and A. Wang. Storage assignment to decrease code size. *ACM Trans. Program. Lang. Syst.*, 18(3):235–253, 5 1996.

[21] S. Lin and B. W. Kernighan. An effective heuristic algorithm for the traveling-salesman problem. *Operations Research*, 21(2):498–516, 1973.

[22] O. Martin and S. Otto. Combining simulated annealing with local search heuristics. *Annals of Operations Research*, 63:57–75, 1996.

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

[24] 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, Philadelphia, PA, USA, April 2006. ACM.

[25] 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.

[26] H. Salamy and J. Ramanujam. An effective heuristic for simple offset assignment with variable coalescing. In *Proc. of the 19th Intern. Conf. on Lang. and Compilers for Parallel Comput.*, LCPC'06, pages 158–172, Berlin, Heidelberg, 2007. Springer.

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

benchmark	OFU	Liao	TB	TB2	INC	INC-TB	TBLK	GA	MWHC
8051	123	102	98	104	99	97	97	97	96
5 instances		0.000	0.000	0.000	0.002	0.000	0.210	2.564	0.018
adpcm	1260	1021	998	1013	1009	990	995	988	979
29 instances		0.000	0.002	0.006	0.046	0.016	4.218	22.494	0.228
anagram	148	104	101	100	103	100	99	99	98
9 instacnes		0.000	0.000	0.000	0.000	0.000	0.146	3.588	0.036
anthr	6686	5437	5362	5525	5428	5360	5353	5356	5318
89 instances		0.028	0.024	0.050	0.430	0.080	44.601	148.712	3.262
bdd	3551	2817	2757	2813	2811	2757	2755	2756	2753
148 instances		0.018	0.014	0.026	0.048	0.028	25.548	112.242	0.858
bison	3720	2934	2892	2922	2930	2890	2889	2889	2888
99 instances		0.010	0.026	0.024	0.038	0.026	23.626	111.074	0.778
cavity	958	815	791	837	812	789	787	789	782
5 instances		0.010	0.010	0.020	0.262	0.710	17.960	32.220	2.214
cc65	4665	3679	3583	3634	3626	3582	3579	3581	3572
208 instances		0.010	0.006	0.024	0.064	0.026	16.378	127.986	0.926
codecs	618	505	497	504	504	497	495	497	495
12 instances		0.002	0.004	0.004	0.006	0.002	1.918	12.854	0.104
cpp	2349	1822	1797	1814	1821	1797	1795	1797	1793
54 instances		0.024	0.014	0.030	0.040	0.048	38.850	114.078	0.538
dct_unr.	2818	2187	2193	2187	2187	2181	2193	2181	2179
1 instance		0.010	0.014	0.022	0.092	8.934	24.978	52.920	1.268
dspstone	1442	1113	1084	1115	1106	1082	1083	1082	1067
23 instances		0.012	0.004	0.022	0.182	0.014	10.376	34.856	0.300
eqntott	132	85	85	85	85	85	85	85	85
8 instances		0.000	0.000	0.000	0.000	0.000	0.036	2.400	0.020
f2c	6473	4804	4741	4774	4798	4740	4734	4737	4732
287 instances		0.022	0.034	0.038	0.056	0.038	33.572	200.866	1.438
fft	17	17	17	17	17	17	17	17	17
5 instances		0.000	0.000	0.000	0.000	0.000	0.212	3.154	0.024
flex	1701	1216	1184	1200	1212	1183	1181	1183	1181
67 instances		0.002	0.004	0.008	0.012	0.010	6.506	43.662	0.290
fuzzy	166	133	128	129	132	128	128	128	128
6 instances		0.000	0.000	0.000	0.002	0.000	0.072	2.380	0.010
gif2asc	575	478	471	481	477	470	469	470	469
6 instances		0.002	0.000	0.012	0.034	0.018	6.780	19.068	0.250
gsm	3187	2604	2585	2642	2600	2584	2583	2584	2509
58 instances		0.026	0.012	0.066	0.062	0.058	58.140	137.052	2.456
gzip	3755	2898	2751	2829	2869	2751	2746	2751	2739
68 instances		0.022	0.020	0.026	0.132	0.034	20.378	90.650	0.738
h263	302	213	212	221	212	212	211	211	208
4 instances		0.000	0.000	0.000	0.002	0.000	0.678	4.678	0.032
hmm	786	553	533	553	548	532	533	532	531
28 instances		0.000	0.002	0.000	0.000	0.002	0.612	13.170	0.086
jpeg	9004	6642	6474	6640	6617	6465	6450	6460	6424
317 instances		0.012	0.014	0.028	0.054	0.050	21.546	195.928	1.530
klt	1288	882	855	869	873	854	854	852	849
50 instances		0.000	0.000	0.002	0.002	0.006	1.496	24.096	0.200
lpsolve	3951	3094	3054	3079	3084	3054	3050	3054	3049
83 instances		0.020	0.024	0.030	0.074	0.036	36.908	136.756	0.728
motion	384	348	350	359	348	344	349	344	343
1 instance		0.002	0.004	0.002	0.008	0.200	3.514	9.000	0.544
mp3	5988	4360	4319	4373	4354	4319	4285	4301	4247
79 instances		0.060	0.050	0.132	0.170	0.180	138.374	234.042	41.012
mpeg2	5030	3884	3836	3904	3873	3828	3821	3824	3787
72 instances		0.020	0.016	0.030	0.104	0.090	30.370	114.412	0.970
sparse	2196	1670	1653	1674	1669	1653	1649	1653	1649
73 instances		0.002	0.006	0.018	0.020	0.026	12.246	66.586	0.424
triangle	1878	1243	1217	1226	1240	1217	1210	1214	1204
79 instances		0.006	0.014	0.038	0.034	0.010	42.417	61.842	16.184
viterbi	1881	1680	1598	1617	1677	1597	1579	1597	1569
10 instances		0.032	0.024	0.076	0.086	0.082	79.002	119.854	0.170
random	379559	353269	351375	354663	352839	351130	351158	350987	349988
2014 instances		0.132	0.090	0.492	4.246	3.782	96.736	1805.557	12.137

Table 4: Offset assignment cost and measured CPU times in seconds (average of five runs) of the evaluated algorithms.