

ULRR

On heuristic solutions to the simple offset assignment problem in address-code optimization

Item Type	Meetings and Proceedings
Authors	Shokry, Hesham;El-Boghdadi, Hatem M.
Citation	ACM Transactions on Embedded Systems Computing (TECS);11(3), article 63
Publisher	Association for Computing Machinery
Download date	2026-05-09 14:19:06
Item License	https://creativecommons.org/licenses/by-nc-sa/1.0/
Link to Item	https://hdl.handle.net/10344/2645

On Heuristic Solutions to the Simple Offset Assignment Problem in Address-Code Optimization[†]

HESHAM SHOKRY*

Lero – the Irish Software Engineering Research Centre
and

HATEM M. EL-BOGHDAI

Computer Engineering Department, Cairo University, Egypt

The increasing demand for more functionality in embedded systems applications nowadays requires efficient generation of compact code for embedded DSP processors. Because such processors have highly irregular data-paths, compilers targeting those processors are challenged with the automatic generation of optimized code with competent quality comparable to hand-crafted code. A major issue in code-generation is to optimize the placement of program variables in ROM relative to each other so as to reduce the overhead instructions dedicated for address computations. Modern DSP processors are typically shipped with a feature called Address Generation Unit (AGU) that provides efficient address-generation instructions for accessing program variables. Compilers targeting those processors are expected to exploit the AGU to optimize variables assignment. This paper focuses on one of the basic offset-assignment problems; the Simple Offset Assignment (SOA) problem, where the AGU has only one Address Register and no Modify Registers. The notion of *Tie-Break Function*, TBF, introduced by Leupers and Marwedel [1], has been used to guide the placement of variables in memory. In this paper, we introduce a more effective form of the TBF; the *Effective Tie-Breaking Function*, ETBF, and show that the ETBF is better at guiding the variables placement process. Underpinning ETBF is the fact that program variables are placed in memory in sequence, with each variable having only two neighbors. We applied our technique to randomly generated graphs as well as to real-world code from the OffsetStone testbench [13]). In previous work [12], our technique showed up to 7% reduction in overhead when applied to randomly-generated problem instances. We report in this paper on a further experiment of our technique on real-code from the Offsetstone testbench. Despite the substantial improvement our technique has achieved when applied to random problem instances, we found that it shows slight overhead reduction when applied to real-world instances in OffsetStone, which agrees with similar existing experiments. We analyze these results and show that the ETBF *defaults* to TBF.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors – *compilers; optimizations*.

General Terms: Algorithms, Experimentation.

Additional Key Words and Phrases: Code size, offset assignment, compilation.

[†]A preliminary version of this paper was published in 11th international workshop on Software & Compilers for Embedded Systems (SCOPE'S'08), Munich, Germany. This paper includes new work of benchmarking the algorithm using the benchmark suite at www.address-code-optimization.org.

*The author is supported in part by the Science Foundation Ireland grant 03/CE2/I303_1 to Lero www.lero.ie

Authors' addresses: H. Shokry, Lero, University of Limerick, Limerick, Ireland; email: hesham.shokry@lero.ie; H. EL-Boghdadi, Computer Engineering Department, Cairo University, Giza, Egypt; email: helboghdadi@eng.cu.edu.eg.

1. INTRODUCTION

As more features and functionality are added to embedded DSP devices, a corresponding need for smaller and faster embedded software is constantly increasing. Automatic generation of compact code is a key factor for success or failure when building such resource-constrained devices. Although it is error prone and time consuming, assembly programming is still an inevitable part of DSP software development, despite the recent advances in source-level optimizations [7]. This is mainly due to the high-quality of hand-crafted code compared to automatically generated versions. In spite of advances in compilation technology, code-generation has yet much to do to optimize generated code in terms of memory size requirements [4, 11, 14, 15]. Quality of the code generated by classical compilation technology has overheads of several hundred percent [4] when compared to the corresponding hand-crafted code, which is not acceptable in industry. Considering the current trend of model-driven engineering to overcome complexity in embedded systems, manual programming in low-level languages is not practical any more. Advanced code-generation techniques are necessary to take this tedious task off developers and, more importantly, to decrease the probability of coding-errors injected in hand-written code [4].

A significant overhead in generated code is due to those instructions used to compute necessary memory-addresses for future accesses of program variables. Within the code flow, each reference to a variable requires extra auxiliary instructions to compute its address to access it in the memory. Those instructions are commonly known as *address-code* and they constitute overhead in the overall application code. Minimizing this overhead code is a task known as *address-code optimization*. Address-code optimization relies primarily on proper placement of program variables relative to each other in memory; a problem formulated as the *offset assignment optimization* problem [2]. The offset assignment is performed during the code-generation phase as part of the traditional code-compilation process, in which the compiler arranges the memory layout. The space and time overhead incurred due to those address-code instructions are quantifying the *cost of the assignment*.

Fortunately, modern DSP processors have a special-purpose hardware feature called Address Generation Units, AGU. AGU is particularly dedicated to help with computing the memory addresses, using special zero-overhead AGU instructions, in parallel with program execution flow. Figure 1 shows the structure of a typical AGU (a detailed description is provided in Section 2). Solutions to the Offset Assignment problem are basically exploiting the AGU and developing efficient algorithms so as to maximize the

use of AGU special instructions. The Offset Assignment problem takes as input an *access sequence* of program variables, extracted from program statements, and suggests a placement of variables that maximizes the use of AGU special instructions and minimizes the unavoidable address-code instructions. An *access sequence* is a sequence of program variables ordered according to their references in the program statements. Figure 2(b) shows an access sequence composed of 8 variables. A variable may appear multiple times in the access sequence, corresponding to the number of times it is referenced within the program statements flow and in the same order of these references. The output of Offset Assignment is a sequence of the program variables, with each variable appearing once in this sequence, and the variables are placed in memory according to this sequence.

There are several variants of the Offset Assignment problem, depending on the capabilities of the AGU. The most basic one is the Simple Offset Assignment, SOA, problem. ‘Simple’ refers to the fact that the AGU under consideration has only one Address Register—a register that temporarily holds addresses during code flow. Another variant of the SOA problem, called SOA-MR, is concerned with AGUs involving one or more Modify Registers (see Section 2 for more details). In case the AGU is shipped with more than one address registers, the offset assignment problem is referred to as the General Offset Assignment, GOA. The GOA problem is typically approached by breaking it into several SOA problems. In this paper, we consider the basic SOA problem, with no modify registers.

Bartley [5] was the first to study the SOA problem and modeled it as a weighted Hamiltonian Path problem. Later, Liao [2] proved that the optimal weighted Hamiltonian path may not correspond to the optimal solution and he modeled the SOA problem as finding the so-called *Maximum Weight Path Cover*, MWPC, in an undirected edge-weighted *Access Graph* [2]. Informally, a MWPC is a set of nonintersecting paths in the access graph where each node is traversed once by exactly one path. The MWPC is a path cover with maximum weight [2]. Liao proved that finding an optimal MWPC is an NP-hard problem and he proposed a greedy algorithm to find it. The algorithm sorts the graph edges, in descending order of their weights, and iteratively selects the next highest weight edge which can be included in the MWPC until all edges are attempted.

Leupers and Marwedel [1] improved Liao’s greedy algorithm and proposed an algorithm based on their notion of *Tie Break Function*, TBF: a function calculated for an edge. The algorithm proceeds in the same greedy way as Liao [2]; however, it applies the concept of tie breaking as a selection criterion when it faces more than one edge with the same weight. According to [1], the TBF of an edge e is computed as the sum of weights

of all of e 's neighboring edges (i.e., all the edges that are incident on the two nodes connected by e). In this way, an edge e with higher TBF value indicates that there are high-weight edges in e 's neighborhood. Leupers and Marwedel postulated that the selection of such an edge for inclusion in the MWPC would restrict the probability of future selections of (high-weight) edges residing in e 's neighborhood. Accordingly, their algorithm selects the edge with least TBF value when there are multiple edges with same weights.

A similar approach for improving Liao's technique but using different selection criterion is that of Hong [3]. Hong has defined the preference-Interference ratio for an edge and proposed to select the edge with highest preference-Interference values, among a set of equally weighted edges. An edge with a higher preference-Interference value indicates that the edge is, relatively, the highest preferred one and has the lowest interference with other edges. The results obtained by Hong were very close to the results in [1] and reduced the offset assignment cost by less than 1%.

Other approaches have used extensions of Liao's SOA problem model. For example, the approach of Salamy and Ramanujam [17] depends on the so-called *coalescing of non-interfering variables* to the same memory location, provided that the live-ranges of these variables do not overlap. This problem is formulated as Coalesced SOA (CSOA). As mentioned, this work depends on the feature of variable coalescing which is not the case in this paper.

Other directions for approaching the SOA problem are based on Algebraic Transformations. Rao and Pande [8] defined the Least Cost Access Sequence (LCAS) problem, and proposed a heuristic to solve it. Recently, Choi and Kim [11] presented a technique that generalizes the work of Rao and Pande [8]. Other related work on transformations includes those of Atri et al. [9] and Ramanujam et al. [10]. These proposals are considered as a pre-step before applying Liao's algorithm, and hence, they are complementary to the work introduced here.

In this paper, we assume an AGU that has only *one address-register* and *no modify-registers*. We build on the notion of TBF [1] and propose a more effective criterion, the *Effective Tie Breaking Function* ETBF, for selecting between equally weighted edges when applying Liao's algorithm [2]. We prove that the TBF function does not represent an accurate criterion for selecting between equally-weighted edges, compared to ETBF, and propose an algorithm for finding the MWPC based on the ETBF selection criterion. We show that our algorithm runs in $O(E^2)$. We apply the algorithm to randomly generated problem instances as well as real-world problem instances from the

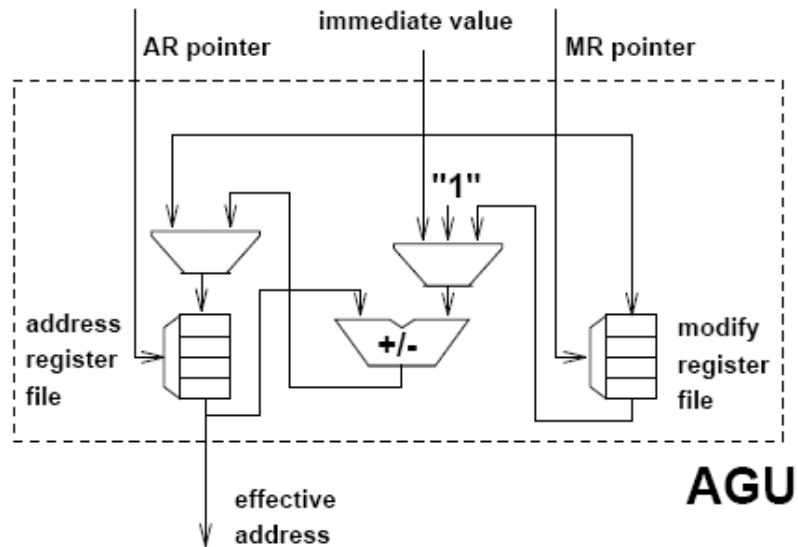


Figure-1: Typical hardware architecture of AGU

OffsetStone testbench [13]. The results demonstrate significant reduction in average offset assignment cost of up to 7% over well-known offset assignment algorithms [1,2,3], when applied to random instances. In the case of real-world problem instances, our approach still shows improvement but less than in the case of random instances. We discuss those results along with observations that agree with the ones reported in similar experiments. The results in this paper apply to a wide range of DSP architectures that employ AGU as a basic feature.

The rest of the paper is organized as follows. Section 2 provides background on address code optimization. Section 3 discusses the concept of the Tie Break Function and introduces our Effective Tie Break Function. Section 4 presents an algorithm for solving the SOA problem based on Effective Tie Break function, along with complexity analysis. Section 5 shows the experimental results and a comparison to other techniques. Finally, we make some concluding remarks in Section 6.

2. BACKGROUND

Many computing architectures provide register-indirect addressing modes with auto-increment and auto-decrement arithmetic. These addressing modes allow for sequential access of memory and increase code density. DSPs and embedded controllers are designed under the assumption that software that runs would make heavy use of auto-increment and auto-decrement addressing. Such processors are equipped with an address generation unit (AGU) that has auto-increment and auto-decrement capability.

Leupers [6] introduced a generic model of an AGU with k Address Registers (ARs) and m Modify Registers (MRs). Figure 1 shows a typical structure of an AGU. The ARs hold the absolute values of addresses, while MRs hold offset values to update ARs whenever necessary.

In parallel with execution of a program instruction, an AGU can execute any of the following set of primitive operations in a machine cycle:

- (1) Immediate AR store in which an AR is loaded with an immediate value.
- (2) Immediate MR store in which an MR is loaded with an immediate value.
- (3) Auto increment/decrement: in which the AR has a value of 1 added to or subtracted from its value.
- (4) Auto modify: same as the auto increment/decrement operation, but instead of modifying the AR with 1, it is modified by the value stored in the MR.

Of these operations, only operations (1) and (2) need a separate instruction (overhead instructions) to be encoded. The auto-operations (3) and (4) are employing only the AGU's data-path resources and are executed in parallel with program instruction flow—zero overhead. So, the goal of offset assignment optimization is to maximize the usage of AGU auto operations.

Now, let us illustrate through an example how the AGU can be used to optimize the code size, assuming an AGU with one AR and no MRs. Figure 2(a) shows an example of a basic block, and its corresponding access sequence appearing in Figure 2(b). There are two possible assignments of variables for this access sequence, shown in Figure 2(c) and Figure 2(d). In the first one, program variables are placed in memory according to their appearance in the basic block code, whereas the second assignment is optimized so as to maximize the use of AGU's auto-operations. The second assignment has cost of 4 which is an optimized assignment compared to cost of 10 in the first assignment.

The variables' access sequence can be summarized by means of weighted and undirected graph, referred to as an *access graph*.

DEFINITION 2.1 (Access Graph). $AG=(V,E)$ is called an *Access Graph* where

- V set of nodes v_i .
- $E \subseteq V \times V \times \mathbb{N}_+$ is a relation in AG and \mathbb{N}_+ . A triple (v_{k1}, v_{k2}, w_k) exists and refers to an edge $e_k \in E$ where w_k refers to is the edge's weight value, and v_{k1}, v_{k2} are the node pair connected by e_k .
- e_k is a predicate over nodes $v \in V$ such that $e_k(v)$ is true if $v \in (v_{k1}, v_{k2})$. ■

For short, we refer to an edge (v_{k1}, v_{k2}, w) as $v_{k1}v_{k2}$ which is a string of its nodes' names.

The access graph $AG(V,E)$ is derived from an access sequence as follows. Each vertex $v \in V$ in the graph corresponds to a unique variable. An edge $e_k=(v_{k1}, v_{k2}, w_k)$ exists if the

$c = c + d + f$
 $a = h - c$
 $b = b + e$
 $c = g - b$
 $a = a - c$

c d f c h c a b e b g b c a c a

(a) Basic block instructions

(b) The corresponding Access Sequence

Addr.	Data
0	c
1	d
2	f
3	h
4	a
5	b
6	e
7	g

```

LDAR AR0,0
LOAD *(AR0)+
ADD *(AR0)+
ADD *(AR0)
SBAR AR0,2
STOR *(AR0)
ADAR AR0,3
LOAD *(AR0)
SBAR AR0,3
SUB *(AR0)
ADAR AR0,4
STOR *(AR0)+
LOAD *(AR0)+
ADD *(AR0)-
STOR *(AR0)
ADAR AR0,2
LOAD *(AR0)
SBAR AR0,2
SUB *(AR0)
SBAR AR0,5
STOR *(AR0)
ADAR AR0,4
LOAD *(AR0)
SBAR AR0,4
SUB *(AR0)
ADAR AR0,4
STOR *(AR0)

```

Cost = 10

(c) Offset assignment according to variables declaration order

Addr.	Data
0	a
1	c
2	h
3	g
4	b
5	e
6	f
7	d

```

LDAR AR0,1
LOAD *(AR0)
ADAR AR0,6
ADD *(AR0)-
ADD *(AR0)
SBAR AR0,5
STOR *(AR0)+
LOAD *(AR0)-
SUB *(AR0)-
STOR *(AR0)
ADAR AR0,4
LOAD *(AR0)+
ADD *(AR0)
STOR *(AR0)-
LOAD *(AR0)+
SUB *(AR0)
SBAR AR0,3
STOR *(AR0)-
LOAD *(AR0)+
SUB *(AR0)-
STOR *(AR0)

```

Cost = 4

(d) Optimized offset assignment

Figure 2: Address assignment example

variables, corresponding to the nodes v_{k1} and v_{k2} , are adjacent to each other w_k times in the access sequence. As an example, the access graph in Figure 3 is derived from the access sequence in Figure 2(b).

In terms of the access graph, the cost of an assignment is equal to the sum of weights of those edges that connect variables assigned to nonadjacent locations. For example, if we used the assignment in Figure 2(d), the edges that connect nonadjacent variables will be the unbolded edges in Figure 3 and they have a weight sum of 4, which is the cost of that assignment. An optimum assignment is one with minimum cost. For AGUs with only one Address Register, optimization of variables assignment is formulated by Liao [2] as the Simple Offset Assignment, SOA, problem, which is of particular interest to this paper. For AGUs with more than one AR, the corresponding optimization problem is

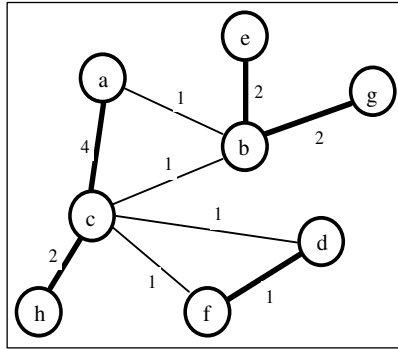


Figure-3: The corresponding access-graph for the access-sequence of variables in Figure-2(b). The bold-edges are the selected MWPC.

referred to as General Offset Assignment, GOA problem,, which is out of the scope of this paper.

Liao [2] showed that finding an optimum variables assignment is equivalent to finding the *maximum weight Path Cover*, MWPC, in an access graph. The MWPC can be defined as follows.

DEFINITION 2.2 (Maximum Weight Path Cover MWPC). Given an access graph $AG(V,E)$, a *Maximum Weighted Path Cover*, MWPC, is a set of *disjoint paths* in AG such that:

- a node v appears exactly once in a disjoint path.
- the set of disjoint paths in AG are covering all the nodes V such that every node v appears exclusively in one path.
- the set of disjoint paths are selected such that the total weight sum of all the edges in the disjoint paths is maximum.

DEFINITION 2.3 Given a MWPC for a graph AG , the *assignment cost* of the MWPC is:
 $(\text{weight-sum of edges in } AG) - (\text{weight-sum of edges in MWPC})$

For example, the bold edges in Figure 3 represent a possible MWPC resulting in the assignment cost of 4.

The algorithm proposed by Liao to find the MWPC is a straightforward greedy algorithm that selects the maximum weighted edge available, and if there is more than one edge having the same weight the algorithm selects one of them arbitrarily. The algorithm proceeds by selecting the next maximum weighted edge until all nodes are covered. Leupers and Marwedel [1] showed that this arbitrary selection leads to less optimized solution, and a proper edge-selection criterion in this case can lead to a reduced assignment cost. Leupers and Marwedel used the notion of *Tie Break Function*, TBF, as selection criterion in presence of several equally-weighted edges.

Our work builds on the notion of TBF but introduces the *Effective Tie Break Function*, ETBF—a more effective edge-selection criterion in case of several equally-weighted edges. In the rest of this paper, we describe our technique for approaching the SOA problem, based on which we detail an algorithm to find the MWPC in an AG.

3. EDGE SELECTION CRITERIA

In this section we discuss criteria for edge selection when applying the greedy algorithm proposed by Liao [2] on an access graph (Definition 2.1) to find the MWPC (Definition 2.2). We study the notion of a Tie Break Function, TBF [1], and introduce a new function; Effective Tie Break Function, ETBF, showing that the new function is more effective than TBF in terms of a reduced cost-assignment (Definition 2.3).

DEFINITION 3.1 (Incident edges). Given an access graph $AG(V,E)$, The $IE(v)$ is the set of *incident edges* on node v such that:

$$\forall e_k \in E, e_k(v) \Rightarrow e_k \in IE(v).$$

DEFINITION 3.2 (Neighboring edges) Given an access graph $AG(V,E)$, The $NE(e_k)$ is the set of *neighborhood edges* to an edge e_k such that:

$$\forall e_k \in E, NE(e_k) = IE(v_{k1}) \cup IE(v_{k2}).$$

where the predicates $e_k(v_{k1})$ and $e_k(v_{k2})$ are true, and $IE(v_{k1})$, $IE(v_{k2})$ are the two sets of incident edges on those nodes, respectively.

For example, in Figure 3, $IE(a)=\{ab, ac\}$ and $IE(d)=\{dc, df\}$ while $NE(ab)=\{ac, be, bc, bg\}$.

3.1 TIE BREAK FUNCTION

The basic idea of a *Tie Breaking Function*, TBF [2], is to estimate the effect of selecting an edge e_k on future selections of edges in neighborhood of e_k , while building the MWPC for an AG. The TBF of an edge e_k , denoted $TBF(e_k)$, is the weight-sum of all neighboring edges of e_k . That is:

$$TBF(e_k) = \sum_{e_i \in NE(e_k)} w_i$$

While selecting the next highest-weight edge for inclusion in the MWPC (Liao [2] algorithm); if there are two or more edges with same weight, the TBF is used as a criterion to select one of those edges. The algorithm in [1] selects the edge with *minimum* TBF value. This decision is based on the intuition that selecting an edge e_k in the MWPC would decrease the *probability* of having edges $e_i \in NE(e_k)$ to be considered in future

selections iterations. Put another way, deferring the selection of e_k , that have a higher TBF value and selecting another one with equal weight but with less TBF value, would leave more opportunities for e_k 's neighbors to get selected in future selection rounds. So, it is better to delay the selection of the edge that has (relatively) higher neighborhood weights, which is quantified by the TBF.

3.2 EFFECTIVE TIE BREAK FUNCTION

We noticed that the TBF overestimates the effect of edge selection. From the definition of MWPC (Definition 2.2), a node may have at most two edges appearing in the MWPC.

LEMMA 1. Consider an access graph $AG(V,E)$. A node $v \in V$ has at most two edges in $IE(v)$ can be included in the MWPC of AG .

PROOF: Because variables are located consecutively in memory, every variable could have a maximum of two neighbors at its predecessor and successor locations. In terms of the MWPC, an edge belonging to the MWPC has its two terminal nodes indicating two consecutive variables adjacent in memory. Therefore, any node can have a maximum of two of its incident edges included in the MWPC. ■

Thus for an edge that connect nodes v_{k1} and v_{k2} , *not all* the edges in $NE(e_k)$ would be affected by selecting the edge e_k . Only four edges—a pair in $IE(v_{k1})$ and another pair in $IE(v_{k2})$ —would be affected (i.e. decreasing their opportunity for future inclusion in the MWPC). Accordingly, the TBF value should consider only the *maximum weighted pairs* in $IE(v_{k1})$ and $IE(v_{k2})$.

DEFINITION 3.3 (Maximum weighted pair). Consider an access graph $AG(V,E)$. For any node $v \in V$ with $|IE(v)| > 2$, The Maximum Weight Pair of a node v , denoted $MWP(v)$, is an edge pair $(e_{m1}, e_{m2}) \subseteq IE(v)$ such that $w_{m1} \geq w_{m2} \geq w_3 \geq \dots \geq w_q$ are the weight of edges $e_{m1}, e_{m2}, e_3, \dots, e_q$, respectively, and $q = |IE(v)|$. ■

For brevity, we will use the notation $|MWP(v)|$ to refer to the summation of these highest weights, i.e. $|MWP(v)| = w_{m1} + w_{m2}$. Note that $|IE(v)|$ refers to *number of elements* in the set $IE(v)$, while $|MWP(v)|$ refers to weight-sum of the pairs in $MWP(v)$. In Figure-3, $MWP(c) = \{ac, ch\}$ and $|MWP(c)| = 4 + 2 = 6$.

It is straightforward to note that $|MWP(v)|$ is the sum of all edges in $IE(v)$ if $|IE(v)| = 2$. In case $IE(v)$ is singleton, $|MWP(v)|$ will be the weight of the single edge in $IE(v)$. Those are the two special cases from Definition 3.3, and we consider them in the algorithm presented in Section 4. A final note is that $MWP(v)$ considers only the edges in $IE(v)$ that

can be included in the MWPC in $AG(V,E)$. So, $|MWP(v)|$ can be zero despite $|IE(v)| > 2$ (according to Definition 2.2 of MWPC)

Now we define the Effective Tie Break Function, ETBF, for an edge e_k and show that it is more accurate than TBF.

DEFINITION 3.4 (Effective Tie Break Function). Given an access graph $AG(V,E)$, the *Effective Tie Break Function* ETBF for an edge e_k is defined as follows:

$$\forall e_k \in E, ETBF(e_k) = |MWP(v_{k1})| + |MWP(v_{k2})|$$

where v_{k1} and v_{k2} are the nodes connected by e_k .

■

With respect to Liao's greedy algorithm in [2] for finding MWPC, ETBF is the best possible criterion for selecting among a set of equally-weighted edges. We formulate this argument in the following theorem.

THEOREM 1 (Effective tie break estimate). Consider an access graph $AG(V,E)$. $\forall e \in E$ assume that the $(e_1, e_2, \dots, e_q, \dots, e_r)$, $r=|E|$, having corresponding weight values $(w_1=w_2= w_3 \dots w_{q-1}= w_q > w_{q+1} \dots w_{r-1} > w_r)$, respectively. Selecting the edge e_k such that $ETBF(e_k)$ is minimum $\forall k \in [1,q]$ is the best possible selection decision by the greedy algorithm in [2].

■

PROOF: Theorem 1 can be proved by counterexample as follows:

- (1) From Lemma 1, deferring to select an edge e_k , $k \in [1,q]$, allows the edge pairs $MWP(v_{k1})$ and $MWP(v_{k2})$ an *opportunity* for inclusion in the MWPC.
- (2) Assume that this *opportunity* is quantified as the *probability of inclusion of a pair* $MWP(v)$ and denoted as a function $prob(MWP(v))$. For brevity, we also assume that $prob(MWP(v))$ becomes zero when selecting an edge in $IE(v)$, and it equals one otherwise¹.
- (3) Before selecting an edge e_k , $k \in [1,q]$, the *prospective cost saving when selecting* e_k can be calculated using the function $Prob()$ as follows:

$$\sum_{\forall e_k, k \in [1,q]} Prob(MWP(v_{ki})) * |MWP(v_{ki})| + C$$

The first term is the weight sum of future $|MWP|$, determined based on their probability selection $prob()$. Note that we considered the terminal node v_{k1} only for each edge e_k so as to prevent duplication in calculations of the $Prob()$ function. The

¹ This is because $Prob()$ is affected only by the selection of one edge e_k , $k \in [1,q]$. Despite $Prob()$ can be affected by other factors, we assume those factors are equally with respect to MWP for all nodes and thus they cancel each other.

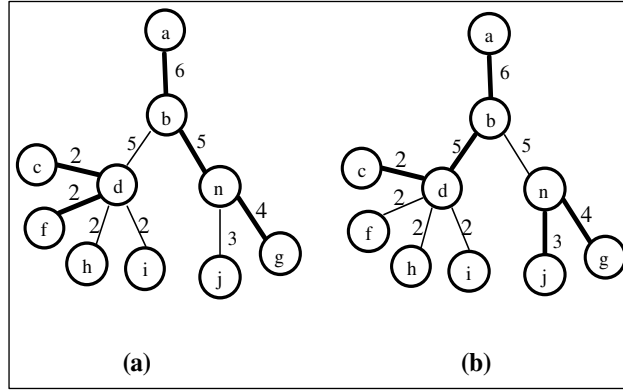


Figure-4: Maximum weight path cover using (a) TBF_{LM} (b) ETBF.

second term C is a constant representing the weight-sum of the edges selected before e_k .

- (4) Assume we have not selected e_k and instead we selected another equally-weighted edge e_i , $i \in [1, q]$, and included it in the MWPC. Then the *prospective cost saving when selecting e_k* at this point is:

$$\sum_{\forall e_k, k \in [1, q], k \neq i} \text{Prob}(\text{MWP}(v_{ki})) * |\text{MWP}(v_{ki})| + C + w_i - (|\text{MWP}(v_{i1})| + |\text{MWP}(v_{i2})|)$$

The terms w_i and $(|\text{MWP}(v_{i1})| + |\text{MWP}(v_{i2})|)$ are the result of selecting the edge e_i in the MWPC. The first term, the weight of e_i , is added to the saved cost value, while the second term is actually the $\text{ETBF}(e_i)$ and it is subtracted from the *prospective cost saving when selecting e_k* . This is because the $\text{prob}(\text{MWP}(v_{ij}))$ and $\text{prob}(\text{MWP}(v_{ij}))$ will become zero when selecting of e_i .

Since the objective of offset assignment is to maximize the term in point (4), the minimum value of $\text{ETBF}(e_i)$ fulfils this objective, which proves the theorem. ■

ETBF function is not an optimal solution for the SOA problem. This is mainly because it follows the greedy algorithm in [2] that has no backtracking. However, ETBF is more accurate than TBF [1] as proved in Theorem 1 and experimentally emphasized by results shown in Section 5.

As illustration, let us see the effect of using the ETBF versus TBF on the access graph in Figure 4. Recall that a maximum of two edges are allowed to be selected for each node (Lemma 1). Figure 4(a) shows a MWPC in bold calculated based on the algorithm in [1], and explained as follows. First, the edge with highest weight, ab , is selected in the

MWPC. The edges bd and bn have the same weight of 5. TBF for each edge is computed and the one with smaller ETBF is selected. Since $TBF(bd)=8$ and $TBF(bn)=7$, then bn is selected in the MWPC. Finally, the edges ng , dc and df are selected in the MWPC. The cost incurred for this MWPC is 12. Figure 4(b) shows the case when the ETBF is used for selection, and the corresponding MWPC is shown in bold. First, the edge with highest weight, ab , is selected in the MWPC. The edges bd and bn have the same weight of 5. The ETBF for each edge is computed and the one with smaller ETBF value is selected. Since $ETBF(bd)=4$ and $ETBF(bn)=7$, then bd is selected in the MWPC. Finally the edges ng , nj and cd are selected in the MWPC. The cost incurred for this MWPC is 11.

Although $TBF(bd) > TBF(bn)$, the selection of the edge bn has prevented the selection of both the edges ng and nj at the same time. However, when ETBF is used, the inclusion of both edges ng and nj in the MWPC was possible, which reduced the cost as explained earlier. We detail in the next section an algorithm that uses the ETBF to solve the SOA problem.

4. THE SIMPLE OFFSET ASSIGNMENT ALGORITHM

In this section, firstly, we give a high level description of the SOA_ETB() algorithm, and then we explain each step of the algorithm along with analysis of expected complexity. The algorithm assumes as input an access graph $AG(V,E)$, built from access-sequence of variables where V is the set of nodes corresponding to these variables and E is the set of edges in AG . Each node v_k has a set of incident edges $IE(v_k)$ (Definition 3.1) and a set of neighboring edges $NE(v_k)$ (Definition 3.2). Finally, we use the notations $e_k \rightarrow n_1$ and $e_k \rightarrow n_2$ to refer to the two nodes connected by e_k . The algorithm's procedure is as follows:

Algorithm: $SOA_ETB(G, Cost)$.

Inputs: $G(V,E)$ is the access graph data structure. V is the set of nodes and E is the set of edges.

Outputs: $Cost$ is the assignment cost.

Procedure:

Step 1: $\forall v_i \in V$, sort the set $IE(v_i)$ in descending order of their weights. Let $IE_{sorted}(v_i)$ be the corresponding sorted-version of the set $IE(v_i)$.

Step 2: Sort the edges set E in descending order of their weights. Let E_{sorted} be the sorted version of E .

Step 3: Initialize $E_{selectable} \subseteq E_{sorted}$ with the set of equally-weighted edges having the highest weight value and are valid to be included in the MWPC (Definition 2.2).

Step 4: $\forall e_k \in E_{selectable}$:

Step 4.1: include $\min(ETBF(e_k \in E_{selectable}))$ in the MWPC.

Step 4.2: update $IE(e_k \rightarrow n_1)$, $IE(e_k \rightarrow n_2)$, E_{sorted} and $E_{selectable}$.

Step 5: loop to Step 4 until $E_{selectable} = \{\emptyset\}$.

Step 6: loop to Step 3 and reinitialize $E_{selectable}$ until no more edges are valid to be included in the MWPC.

■

Complexity analysis. Now we explain the steps of the algorithm and analyze its worst-case complexity. Step 1 sorts the edges in every set $IE(v_i)$, corresponding to each node v_i , to simplify prospective calculations of ETBF for an edge (step 4.1). Step 1 takes $O(|E| \log |E|)$ time.

Step 2 sorts the edges in the set E in descending order of their weights. This step executes in $O(|E| \log |E|)$ steps, assuming quicksort is used. Step 3 initializes $E_{selectable}$ with the edges that have the same and highest weight. Step 3 runs in $O(|E|)$ time.

Step 4 is where the edge selection process takes place. It repeatedly selects an edge from the set of edges $E_{selectable}$ that have the same and highest weight. The selected edge is the one with minimum ETBF value (Step 4.1). In Step 4.2, the selected edge is excluded from the set E_{sorted} , $E_{selectable}$, $IE(e_k \rightarrow n_1)$ and $IE(e_k \rightarrow n_2)$. This assures that $IE(e_k \rightarrow n_1)$ and $IE(e_k \rightarrow n_2)$ are always sorted.

The worst case of Step 4 arises in the situation when all edges have the same weight. In such case, the effort of Step 4 is calculated as follows. Step 4.1 takes $O(|E|/2)$ time, this is because in every new iteration the number of edges decreases by at least one edge. Step 4.2 takes constant time. Since the outer loop (Steps 3-6) repeats $O(|E|)$ iterations, where it reinitialize $E_{selectable}$ with the rest of valid edges, so Step 4 executes $O(|E|^2)$ times.

THEOREM 2. The algorithm SOA_ETB() builds a MWPC for a graph $G(V,E)$ in $O(|E|^2)$ time.

■

It is important to note that this complexity exceeds those of other algorithms such as in [1]. For example, the algorithm implemented in the benchmark [13], which corresponds to the TBF-based technique in [1], has the tie-break function calculated only once before the edge selection loop. The same case applies to other algorithms implemented in the benchmark [13].

Unlike these algorithms, after each selection of an edge for inclusion in the MWPC, (Step 3), our technique re-computes ETBF values of the edges and reinitializes $E_{selectable}$ accordingly. This is important because either or both edges that are involved in the ETBF calculations (or *some* edges in case of TBF [1]) may possibly have become invalid after selection of an edge. So, we re-compute the ETBF values of the unselected edges so as to reflect the exact values of the effective neighborhood weight for each. This explains the experimental results of our technique on random instances, as well as the evaluation on real-world instances. These results are presented in Section 5.

There are, however, special cases arising when calculating ETBF. In both TBF [1] and our ETBF, the basic principle of deferring to select an edge is that it might (in future) impede the selection of high-weighted candidate edges in the next selection step. This is motivated by definition of MWPC, upon which the whole technique is built. For TBF [1], when calculating $TBF(e_k)$, the weight w_k is included in the sum $TBF(e_k)$. This obviously does not affect the selection decision because the same weight value of w_k will be added to all other TBFs calculated for other edges under decision. This is not the case for ETBF. When calculating our $ETBF(e_k)$, the weight value w_k will affect the selection decision only if it is included in the sum $ETBF(e_k)$. This is because, at either node connected by e_k , the *third highest edge* will not be considered if e_k is already included. This can be understood directly from ETBF definition where we consider only the highest two edges.

To explain these special cases, let us denote the k edges incident on a node v and valid for selection as $e_k, e_{k-1}, e_{k-2}, \dots, e_1$ where $w_k > w_{k-1} > w_{k-2} > \dots > w_1$, and let us assume these edges are valid for future selection. Recall that an edge is valid for selection if either (or both) of its nodes has at most one edge already selected. When calculating $ETBF(e_k)$ to compare it to other edges with same weight w_k , there are different cases explained below.

Case1: The node v has more than two edges. We calculate ETBF as follows:

Case1.1: If v has no selected-edges yet, then we consider e_{k+1} and e_{k+2} when calculating $ETBF(e_k)$.

Case1.2: If v has only one selected edge, then we consider only e_{k+1} in the calculation of $ETBF(e_k)$.

Case2: The node v , has only two valid edges. We calculate ETBF as follows:

Case 2.1: If v has no selected-edges yet, then we consider only e_{k+1} when calculating $ETBF(e_k)$.

Case 2.2: If v has only one selected-edge, then we do not consider any edges from v when considering calculating $ETBF(e_k)$.

The rationale behind all of those cases is based on the definition of ETBF in Definition 3.4. For example, in Case 2.2 above, the selection of e_k would impede no other edges on v from future selection, because if e_k is selected, all the other edges that was valid before selection of e_k will become invalid.

5. EXPERIMENTAL RESULTS

We have evaluated our technique on randomly generated problem instances as well as on real-world instances. In this section we show the effectiveness of the algorithm in both evaluations and discuss the results.

5.1 EVALUATION ON RANDOM PROBLEM-INSTANCES

We generated random access graphs with different sizes that range from 10 to 100 nodes and 30 to 590 edges. The ranges of problem-size, in terms of nodes and edges, were selected just to cover as wide a range as possible of the problem instances that could arise in real world applications. Edges' weights were generated and assigned to edges randomly.

For every problem-size, a set of 250 access-graphs instances were generated. The assignment cost of every problem-size was computed as the average of cost values of all the 250 random instances. The cost values represent the cost of the MWPC constructed for each graph. We compared the cost resulting from the MWPCs generated by our algorithm to the results from related algorithms by Liao [2], Leupers and Marwedel [1], and Hong [3]. Table 1 shows sample of these results from selected problem sizes. The first column shows the graph size in terms of the number of nodes and edges of the access graph. Results obtained from algorithms of Liao, Leupers and Marwedel and Hong shown in the second, third and fourth columns, respectively. Finally, the results of our algorithm are shown in the fifth column (denoted as Ours.)

Table 1: *Random problem-instances results shown in terms of the average assignment-cost (Definition 2.3)*

Graph size ($ V , E $)	Liao[2]	Tie Break [1]	Hong [3]	Ours
(10, 20)	459.70	406.97	403.79	382.84
(10, 30)	484.19	442.84	440.00	411.84
(10, 40)	491.10	448.82	445.86	434.14
(20, 40)	449.88	426.94	425.66	401.62
(20, 80)	480.74	424.25	423.28	399.31
(20, 190)	1278.46	1130.16	1119.99	1089.70
(40, 60)	440.07	402.89	402.32	389.91
(40, 200)	1880.85	1761.60	1749.09	1673.87
(40, 500)	8307.47	7346.29	7285.32	7017.18
(60, 100)	467.47	417.78	416.36	397.98
(60, 280)	2280.98	2149.37	2140.34	2062.32
(60, 550)	8967.50	8570.24	8524.81	8025.17
(80, 100)	427.93	392.37	391.27	372.94
(80, 280)	1923.22	1727.24	1715.67	1614.45
(80, 570)	9023.89	8018.63	7965.71	7550.34
(100, 200)	786.89	744.00	742.37	709.71
(100, 300)	2003.35	1919.21	1910.38	1840.91
(100, 590)	9399.89	8666.70	8597.36	8131.10

As an example from Table 1, for the graph of size of 10 nodes and 30 edges, the average cost of offset assignment (as defined in Definition 2.3) is 484.19 for Liao[2], 442.84 for Leupers and Marwedel [1] and 440 for Hong [3], while the average cost of our algorithm is 411.84. Also for a graph of size of 80 nodes and 280 edges, the average cost of offset assignment is 1923.22 for Liao[2], 1727.24 for Leupers and Marwedel [1] and 1715.67 for Hong [3] while the average cost of our algorithm is 1614.45. From the results shown, our algorithm achieves up to 7%.

In general, this improvement in cost-reduction is also quite substantial compared to the improvement made by the TBF [1] over Liao's original algorithm. This is due to the fact that ETBF function provides a more accurate prediction of candidacy of future edges to be included in the MWPC, and, in turn, a more effective guidance to the edge selection

process. In the access-graph generation process, we tended to generate weight values lying within a short range compared to the number of edges, so as to increase the possibility of having groups of edges, with each group having similar-weight edges. The purpose of this was to experiment with the potential of the ETBF in presence of enough opportunity to work.

Another benefit of this improvement is that, unlike other algorithms, we decided to re-compute the ETBF values each time we decide about the next edge to be included in the MWPC. Despite the complexity $O(|E|^2)$ of our algorithm, which is a bit higher than other algorithms with complexity $O(|E| \log |E|)$, the obtained results outweigh this complexity, and the running time of $O(|E|^2)$ is not an issue given the speeds of modern workstations. For example, the problem sizes with edges higher than 250 edges execute in an order of milliseconds in a standard desktop computer (Core2Duo with 2GB of RAM memory). Moreover, it is quite rare to have all edges with identical weights in the same access graph, the case of which we computed the complexity above.

Another side experiment we did on the random problem-instances in OffsetStone [13] showed less figures of improvement, and this is fundamentally due to the different strategy of instances' random generation process. In our process, we tended to generate dense graphs.

In conclusion, this experiment shows the potential of our technique, when applied to random problem-instances, to predict the effect of selecting an edge on the future selection of other edges. However, random problem instances generally do not accurately reflect the real world problems [16]. This necessitated an evaluation on real-world problem instances as we will see in next section.

5.2 EVALUATION OF REAL-WORLD PROBLEM-INSTANCES

Experience shows that real-world problem instances of access sequences tend to have higher locality of accesses [16]. Accordingly, we evaluated our technique of ETBF on the OffsetStone [13,16] testbench. We implemented our algorithm using some of the basic functionality in OffsetStone [13]; however we modified other functions to implement the inner loop in Step 3 of our algorithm. We also suppressed the zero-weighted edges from the edge set E , as we rather focused on computing the cost than the resultant variables placement order.

In this experiment, we used similar strategy of evaluation as in [16] in terms of using all the benchmarks in [13] and measuring the performance in terms of SOA-OFU() which

is a naïve assignment of variables as they appear in the code. We compared only techniques implemented in the testbench [3] and dropped other techniques that are not (such as the technique of Hong [3] used in Table 1). We also dropped the comparison with the branch-and-bound genetic-based techniques. For the *Incremental* algorithm [9], we used its version when combined with TBF as introduced in [16]. Table 2 below shows the results obtained

Table 2: *Evaluation on real-world instances. Results shown as percentages (100%) of the cost generated by the naïve SOA_OFU() algorithm.*

Benchmark	Liao[2]	TBF [1]	INC-TBF	Ours
8051	83.1	79.8	79.0	79.8
adpcm	81.1	79.3	78.6	79.2
anagram	68.9	66.9	66.2	66.2
anthr	81.1	79.9	79.9	79.9
bdd	78.6	76.9	76.9	76.9
bison	78.2	77.1	77.0	77.1
cavity	85.1	82.4	82.2	82.2
cc65	78.4	76.3	76.3	76.2
codecs	81.5	80.3	80.3	80.3
cpp	77.4	76.3	76.3	76.3
dct_unrolled	77.6	77.8	77.4	77.6
dspstone	76.4	74.4	74.3	74.4
eqntott	65.0	65.0	65.0	65.0
f2c	73.7	72.7	72.6	72.7
fft	92.0	92.0	92.0	92.0
flex	71.3	69.3	69.3	69.3
fuzzy	77.5	74.2	74.2	74.2
gif2asc	83.1	82.0	81.7	81.9
gsm	81.5	80.9	80.9	80.8
gzip	77.1	73.2	73.2	73.2
h263	70.3	70.0	70.0	70.0
hmm	70.5	67.4	67.3	67.3
jpeg	73.7	71.8	71.7	71.8
klt	68.2	66.1	66.1	66.0
lpsolve	78.1	77.1	77.1	77.1

Benchmark	Liao[2]	TBF [1]	INC-TBF	Ours
motion	90.6	91.1	89.6	89.9
mp3	72.3	71.6	71.6	71.6
mpeg2	77.0	76.0	75.9	76.0
sparse	75.9	75.1	75.1	75.1
triangle	65.8	64.4	64.4	64.4
viterbi	89.3	85.0	84.9	84.9

From the results we observe that:

- (1) Our technique achieves at least the same improvement as TBF (there is no case in Table 2 where it improves less than TBF). However, the improvement of ETBF over TBF is quite small when compared to the results obtained on random problem-instance in the previous experiment. This is quite consistent with the observation in [16] that the edge weights in the access graph are more uniformly distributed in case of random instances than in the case of real-world instances. Given the behavior of *defaulting* to TBF technique, our ETBF can safely replace it.
- (2) The combined *Incremental* technique [9] seems to have potential to generally improve the *tie-break* techniques (TBF and Ours). This is apparent from the (slight) improvement it makes over Ours. This means that there is more room for improvement if the Incremental technique is combined with our TBF approach. However as noted in the previous observation (and also in [16]) the overall improvement of all SOA algorithms is quite small so that it can be used in situations where a single word in ROM matters.

6. CONCLUSIONS

In this paper, we described and evaluated a heuristic algorithm for solving the SOA problem. Our work is based on Liao’s model [2] of solving the SOA problem, that is to model the variable accesses as an, edge-weighted, undirected graph, called an access graph. The solution to the SOA problem then is to find the maximum-weighted path in the graph covering all of the graph nodes such that each node is visited once.

Our approach to the SOA problem is based on the concept of *tie-break function* TBF introduced by Leupers and Marwedel [2] which is proposed as a selection-criterion for the algorithm in [1] in presence of equally-weighted edges. We proposed a more effective criterion, *effective tie-break function* ETBF, that leads to more reduction of the assignment cost. We argued that ETBF has a more accurate prediction of candidacy of

future edges to be included in the path cover, and, in turn, a more effective guidance to the edge selection process.

Like the TBF technique [2], ETBF is based on the fact that when an edge is selected it reduces the probability of future selection of edges in that edge's neighborhood. Unlike TBF, ETBF does not consider all the neighboring edges, and only considers the *maximum-weighted pair* of edges. This is based on the constraint mentioned above about the path cover; every node is visited only once. This means that a node could have only two of its edges included in the path cover. Moreover, the edge selection step in our algorithm is optimized so as to exclude invalid edges from the calculation of ETBF.

We have evaluated our approach on random problem-instances covering a wide range of problem sizes and we gave analysis and explanations of results. The experiment showed considerable potential of the ETBF technique compared with TBF and other related techniques.

Moreover, we ran another evaluation experiment on real-world problem instances from the OffsetStone testbench [13] and compared it with related techniques using similar evaluation strategy to that in [16]. Like other algorithms, we observed that ours is achieving slightly better results when applied to real-world problem instances, not comparable to the substantive results achieved in case of random instances. This is mainly because the random instances are not reflecting real-world code as those in OffsetStone [13]. However, we also observed that ETBF algorithms *defaults* to *TBF* if not improving over it, and hence it can safely replace it. Another positive point is that the results obtained when combined the TBF approach with the Incremental algorithm (Atri et al. [9]) are better. This suggests that our ETBF approach can similarly be improved when combined with the work in [9].

ACKNOWLEDGEMENT

We are grateful to the anonymous referees and editors for their helpful and detailed reviews on early drafts of this article. This work is supported, in part, by Science Foundation Ireland grant 03/CE2/I303_1 to Lero – the Irish Software Engineering Research Centre (www.lero.ie).

REFERENCES

- [1] R. Leupers and P. Marwedel. Algorithms for Address Assignment in DSP Code Generation. *In Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, p.109-112, November 10-14, 1996, San Jose, California, United States, pp 109-112, 1996.
- [2] S. Liao. Code Generation and Optimization for Embedded Digital Signal Processors. PhD thesis, MIT Department of EECS, January 1996.

- [3] J Hong. Memory Optimization Techniques for Embedded Systems, PhD Thesis, Dept. of Electrical and Computer Engineering, Louisiana State University, July 2002.
- [4] V. Zivojnovic, J. Martinez, C. Schläger and H. Meyr, "DSPstone: A DSP-Oriented Benchmarking Methodology", *In Proceeding of The International Conference on Signal Processing Applications and Technology ICSPAT 1994 - Dallas*, Oct. 1994.
- [5] D. Bartley. Optimizing Stack Frame Accesses for Processors with Restricted Addressing Modes. *Software Practice and Experience*, 22(2):101–110, February 1992.
- [6] R. Leupers. Code Optimization Techniques for Embedded Processors: Methods, Algorithms, and Tools. Kluwer Academic Publisher, 2000.
- [7] H. Falk and P. Marwedel. Source Code Optimization Techniques for Data Flow Dominated Embedded Software Source level optimizations. Springer 2004.
- [8] A. Rao and S. Pande. Storage Assignment Optimizations to Generate Compact and Efficient Code on Embedded Dsps. *ACM SIGPLAN notices*, 1999, Atlanta, GA, USA, pages 128-138, May 1999.
- [9] S. Atri, J. Ramanujam and M. Kandemir. Improving Offset Assignment for Embedded Processors. *In S. Midkiff et al. (Eds.) LCPC 2000, LNCS 2017*, pp. 158–172, Springer- Verlag, 2001.
- [10] J. Ramanujam, J. Hong, M. Kandemir and S. Atri. Address Register-Oriented Optimizations for Embedded Processors. *In Proceeding of the 9th Workshop on Compilers for Parallel Computers*, pp. 281–290, Edinburgh, Scotland, June 2001.
- [11] Y. Choi and T. Kim. Address Assignment Combined with Scheduling in DSP Code Generation. *In Proceeding of the 39th Design Automation Conference*, June 2002.
- [12] H. Ali, H. El-Boghdadi and S. Shaheen. A New Heuristic for SOA Problem Based on Effective Tie Break Function. *In Proceeding of the 11th International Workshop on Software & Compilers for Embedded Systems (SCOPE'08)*, ACM International Conference Proceeding Series Vol. 296.
- [13] Address code optimization benchmark www.address-code-optimization.org (last visited March 2010)
- [14] J. Hong and J. Ramanujam. Memory Offset Assignment for DSPs. *In Proceedings of the 3rd international conference on Embedded Software and Systems*. Daegu, Korea, 2007. Springer-Verlag.
- [15] S. Udayanarayanan and C. Chakrabarti. Address Code Generation for Digital Signal Processors. *In Proceedings of the 38th annual Design Automation Conference*. Las Vegas, Nevada, United States, 2001. ACM.
- [16] R. Leupers, "Offset Assignment Showdown: Evaluation of DSP Address Code Optimization Algorithms", *The 12th International Conference on Compiler Construction, LNCS 2622*, Poland, Apr. 2003. Springer-Verlag.
- [17] H. Salamy and J. Ramanujam. "An Effective Heuristic for Simple Offset Assignment with Variable Coalescing," *LCPC (C. Cascaval et al. Eds.) LNCS 4382*, Springer-Verlag, 2007.