# Offset Assignment Showdown: Evaluation of DSP Address Code Optimization Algorithms

Rainer Leupers

Institute for Integrated Signal Processing Systems (ISS)
RWTH Aachen, Germany
leupers@iss.rwth-aachen.de

**Abstract.** Offset assignment is a highly effective DSP address code optimization technique that has been implemented in a number of ANSI C compilers. In this paper we concentrate on a special class of offset assignment problems called "simple offset assignment" (SOA). A number of SOA algorithms have been proposed recently, but experimental results and direct comparisons are still sparse. This makes the practical selection of a suitable SOA algorithm for implementation in a compiler very difficult. This paper aims at closing this gap by providing a comprehensive benchmark suite and empirical evaluation based on real-life application programs. Our results for the first time permit a detailed assessment of all major SOA algorithms. In addition, we propose a new and superior combination of SOA heuristics.

## 1 Introduction

Due to the increased importance of software in embedded system design, code optimization techniques for embedded processors, particularly for *digital signal processors (DSPs)*, have gained high interest in academia and industry. As compared to general-purpose processors, DSPs show a number of special hardware features, many of which impose new challenges on compiler construction:

- Harvard architecture with separate program and data buses
- Dual memory banks for high data access bandwidth
- Hardware multiplier for fast product computation
- DSP-specific instructions like multiply-accumulate, multimedia (SIMD) instructions, and saturating arithmetic
- Limited amount of instruction-level parallelism
- Inhomogeneous register set
- Support for zero-overhead hardware loops
- Real-time capabilities
- Dedicated *address generation units* (AGUs)

This paper considers code optimization techniques aiming at maximum utilization of AGUs.

G. Hedin (Ed.): CC 2003, LNCS 2622, pp. 290–302, 2003.

### 1.1 Address Generation Units and Offset Assignment

*Offset assignment* is a central code optimization technique in many C/C++ compilers for DSPs. It exploits the fact that many standard DSPs (e.g. TI C2x/C5x, Motorola 56xxx, Analog Devices 210x, ST D950) as well as numerous application-specific DSPs comprise an AGU that is capable of performing address (or pointer) arithmetic in parallel to the main data path.

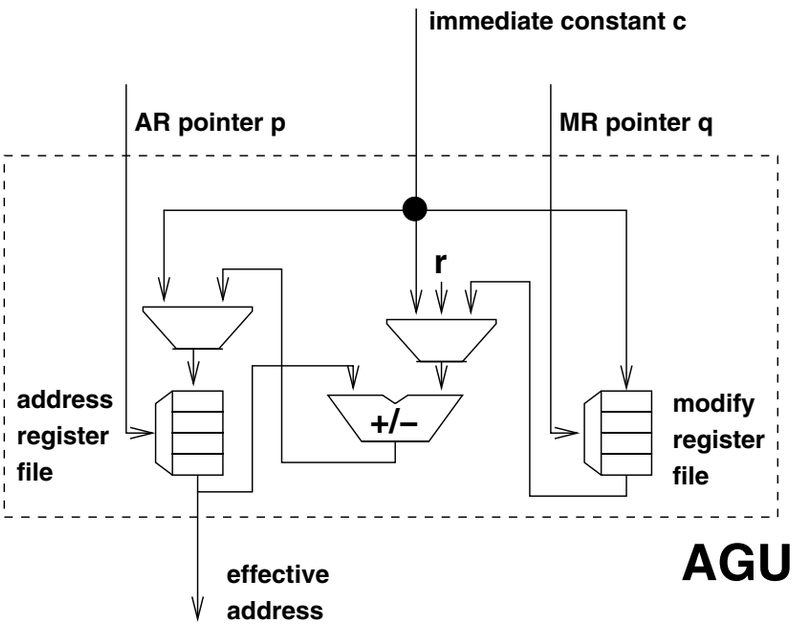


**Fig. 1.** Address generation unit (AGU) architecture in DSPs with address register (AR) and modify register (MR) files.

A typical DSP AGU (see fig. 1) comprises a file of address registers (ARs) that store pointers for indirect memory addressing modes. In order to optimize clock speed and to save silicon area, DSPs, in contrast to CISC and RISC machines, frequently do not support "base-plus-offset" addressing modes. Instead, in order to compute a new address $a' = a \pm c$ from a given address $a$ stored in some AR, that AR has to be explicitly modified by adding or subtracting some constant $c$. The code efficiency of such AR modifications depends on the concrete value of $c$: if the absolute value of $c$ is small enough such that $c$ fits into the *auto-increment range* $R = [-r, r]$, then $c$ can be encoded as an immediate operand into the same instruction that performs a memory access (LOAD or STORE) at address $a$. In that case, the AR modification can be performed within the AGU in parallel to the memory access by means of an *auto-increment* (or *auto-decrement*, dependent on

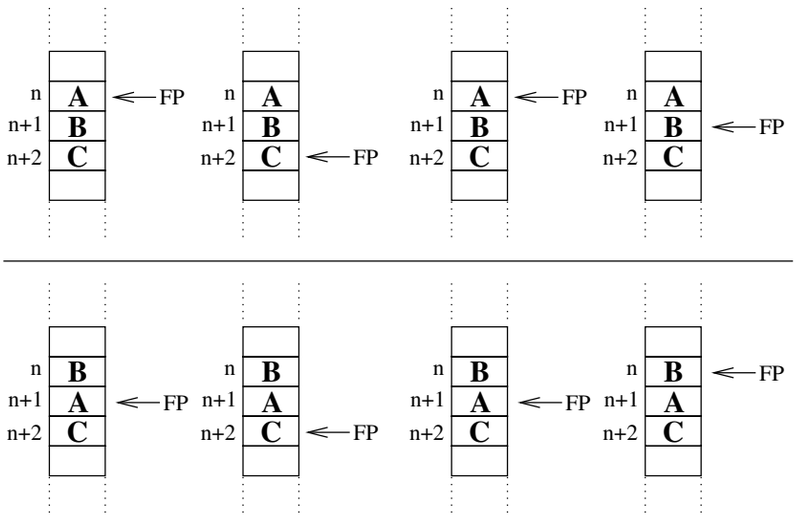


**Fig. 2.** Illustration of offset assignment

the sign of $c$) operation. Otherwise, if $|c| > r$, an extra instruction is required to compute the address $a' = a \pm c$ for the next memory access.

Hence, the auto-increment based address computation results in the highest code performance and density, and any C/C++ compiler for DSPs should aim at maximizing its use when generating code for address computations. One way to do this is *offset assignment*, where the memory layout for program variables is optimized such that the maximum number of address computations for scalar variables can be implemented by auto-increment. This is possible due to the fact that the stack layout for the local scalar variables of a C function can be freely chosen by the compiler.

### 1.2 Offset Assignment Example

Fig. 2 illustrates a sample stack frame layout in a DSP-specific compiler. The compiler typically allocates one of the ARs as a *frame pointer* (FP), which is used to address local variables on the stack. Suppose, we have three such variables, A, B, and C, which are accessed in the sequence $S = (A, C, A, B)$. Furthermore, suppose the auto-increment range $R$ is restricted to $[-1, 1]$. This special case of using a single FP and $R = [-1, 1]$ is called *simple offset assignment* (SOA).

The upper part of fig. 2 illustrates the situation when the variables are assigned to stack locations (or *offsets*, relative to the stack frame boundary) $n$, $n + 1$, $n + 2$, in alphabetic order. Initially, FP points to variable $A$ at address $n$. The next access goes to $C$ located at $n + 2$. Due to the missing "base-plus-offset" addressing mode in DSPs, FP cannot remain constant throughout the entire function execution (as it normally holds for CISC or RISC compiled code), but needs to be implemented as a *floating* or *roving* frame pointer. Thus, in order to access the variables according to sequence $S$, FP needs to be

modified by the values $+2, -2, +1$, in that order. Due to $R = [-1, 1]$, only the last FP modification (+1) can be implemented by auto-increment, while two extra instructions are required to implement the modifications by +2 and -2. However, as shown in the lower part of fig. 2, the situation changes drastically when the variables are assigned to memory addresses in the order $B - A - C$, in which case the access sequence $S$ implies FP modifications by $+1, -1, -1$, all of which fall into the auto-increment range $R$. Hence, the latter variable layout will result in better code, and it is the goal of SOA algorithms to compute such "good" variable layouts.

### 1.3 Motivation

Experimental surveys indicate that it is not unusual for DSP machine code to comprise 20%–30% (sometimes even more than 50%) of instructions used for address computations [1], [2]. In terms of total code size, the effect of performing offset assignment within a C compiler is typically in the order 5%–20% [3], which is quite significant for DSPs with tight ROM size constraints. Due to their high importance for DSP code quality, offset assignment techniques have been implemented in several research (e.g. SPAM [3] or RECORD [4]) and industrial compilers (e.g. TI's C2x/C5x C compiler [5] or CHESS [6]) for DSPs.

Even though SOA is just a special case of offset assignment problems, it represents a real-world problem. This is due to the fact, that many DSPs show a relatively small instruction word length (mostly 16 bits), which allows only for a narrow auto-increment range like $[-1, 1]$. Moreover, generalized offset assignment approaches using multiple frame pointers mostly rely on SOA algorithms as subroutines.

Consequently, a number of different SOA algorithms have been proposed in the literature. In spite of this, from a scientific viewpoint, the situation is not really satisfactory, since so far there has been no comprehensive benchmarking of the different SOA algorithms for real-life problems. Some algorithms have been compared to others, but frequently the comparisons are incomplete and are based on small program fragments or even random problem instances, so that reported results are hardly reproducible. So the question of which SOA algorithm is the "best" (w.r.t. their computation time vs solution quality tradeoff) is still largely open.

Therefore, in this paper we do not just propose yet another SOA algorithm, but our main goal is to consolidate previous work by means of a comprehensive empirical study, in which we evaluate a set of different algorithms for a large suite of realistic SOA problem instances. This allows us to draw conclusions on which algorithms are most useful in practice and may be promising platforms for future offset assignment research. In more detail, the contributions of this paper are:

1. We briefly review the major existing SOA algorithms and available experimental comparisons.
2. We propose an extensible benchmark suite, called *OffsetStone*, for offset assignment algorithms together with the necessary tool support.
3. We use OffsetStone to evaluate a total of 8 SOA algorithms and give detailed experimental results about their performance in terms of computation time and (both absolute and relative) solution quality.

4. We present a new combination of two fast SOA heuristics that turns out to be superior to all previous heuristics.

The remainder of this paper is structured as follows. Section 2 discusses related work and gives a more precise description of the SOA problem. Section 3 outlines the OffsetStone benchmarking methodology and its tools. In section 4, we provide detailed experimental results. Finally, section 5 gives conclusions and mentions future work.

## 2 Related Work

### 2.1 Access Graph Model

Bartley [5] proposed the *access graph* model for the simple offset assignment (SOA) problem, which forms the baseline for most SOA algorithms. Given a variable set $V = \{v_1, \ldots, v_n\}$ and a variable access sequence $S = (s_1, \ldots, s_m)$ of a basic block with $\forall i \in [1, m] : s_i \in V$, the access graph is an undirected, complete, and edge-weighted graph $G = (V, E, k)$ with $E = \{\{v, w\} | v, w \in V\}$. The function $k : E \rightarrow \mathbf{N_0}$ assigns a weight to each edge $e = \{v, w\}$ that denotes the number of *access transitions* between $v$ and $w$ in $S$, i.e., the number of subsequences of $S$ of the form $(v, w)$ or $(w, v)$. Due to the symmetry of auto-increment and auto-decrement, the ordering of $v$ and $w$ is irrelevant here. Likewise, self-edges of the form $\{v, v\}$ can be neglected. The left part of fig. 3 exemplifies the access graph model for $V = \{A, B, C, D\}$ and $S = (D, A, C, B, A, D, A, B, C)$.

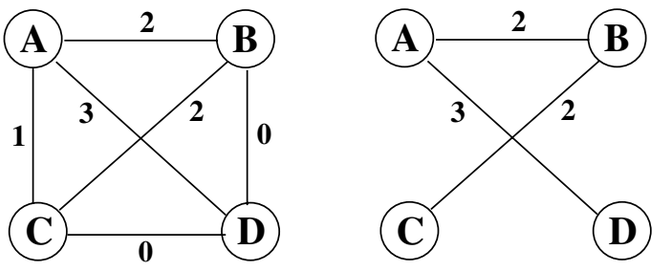


**Fig. 3.** Access graph model and maximum weighted Hamiltonian path

Any access transition $(v, w)$ in $S$ can be implemented by auto-increment, if and only if $v$ and $w$ are assigned neighboring stack locations, i.e. the offset difference of $v$ and $w$ is covered by the auto-increment range $[-1, 1]$. In order to maximize the use of auto-increment addressing, obviously those variable pairs $\{v, w\}$ should be neighbors in the stack frame, whose edge weight $k(\{v, w\})$ in $G$ is high, since this will save many extra instructions for address computation.

### 2.2 Offset Assignment Heuristics

As pointed out by Liao [3], the SOA problem eventually amounts to finding a *maximum weighted Hamiltonian path* $P$ in $G$, i.e. a path touching each node once with the maximum edge weight sum (see right part of fig. 3). The memory layout is derived from $P$ by assigning those node pairs to adjacent memory locations, which are also neighboring in $P$ (i.e. either C-B-A-D or D-A-B-C in the example from fig. 3).

The *cost* of an SOA solution $P$ is defined as the sum of the weights of $G$'s edges *not* covered by $P$. This corresponds to the number of extra address computation instructions to be inserted into the machine code. By means of a simple reduction from the classical Hamiltonian path problem [7] it can be shown that computing $P$ is an NP-complete problem. Hence, heuristics should be used, except for small problem instances.

Bartley [5] proposed a greedy heuristic for finding path $P$. His algorithm iteratively picks an edge $e$ of highest weight $k(e)$ in $G$ and checks whether inclusion of $e$ into a partial path $P$ would still allow for a valid solution. This is iterated until a complete path with $|V| - 1$ edges has been selected.

Liao [3] proposed a more efficient implementation of Bartley's SOA algorithm, by temporarily neglecting edges of zero weight (which are frequent in realistic access graphs) and using an efficient Union/Find data structure for checking for cycles. Besides the implementation issues, Liao's algorithms produces the same results as Bartley's.

In his thesis [1], Liao additionally proposed a branch-and-bound (B&B) algorithm for SOA, which can be used to construct optimal solutions. The B&B algorithm is capable of effectively pruning the huge search space, but it can generally only be applied to small problem instances due to sometimes exhaustive runtime requirements.

Both Bartley's and Liao's heuristics do not include a special handling of edges with equal weight during path construction. However, same-weight edges are very common in access graphs, and the solution quality may critically depend on the order in which edges are investigated during path construction. Therefore, Leupers and Marwedel [4] proposed to extend Liao's algorithm by a *tie-break* heuristic for choosing among same-weight edges. An experimental evaluation for a set of random SOA problem instances indicated that the tie-break heuristic on average gives a slight improvement over Liao's heuristic. This has been confirmed by independent experiments in [8], [9], while other experiments on some of the DSPStone [10] benchmark programs reported in [2] did not indicate such an improvement.

A genetic algorithm (GA) based approach to SOA has been presented in [11]. In contrast to most other methods, it does not use the access graph model, but constructs offset assignments directly by a (relatively time-consuming) simulation of a natural evolution process. Actually, the GA has been mainly intended for a more general class of offset assignment problems, but it can easily be restricted to solve the SOA problem. A direct comparison to fast heuristics for the special case of SOA has not been reported, though.

Atri et al. proposed an *incremental* SOA algorithm [12]. It starts with an initial SOA solution, constructed by some heuristic, and performs an iterative improvement by a local exchange of access graph edges selected for the maximum weighted Hamiltonian path. An experimental comparison to Liao's heuristic [3] for a set of random SOA instances indicated that the initial solution can be improved in 3–8% of the cases considered,

where the average improvement is about 5%. Unfortunately, no comparison to other SOA algorithms was reported.

Besides these approaches, many generalizations of SOA have been considered, including the *general offset assignment* (GOA) problem [3], [4], [17], [11] that handles multiple frame pointers, DSPs with auto-increment operations between the memory accesses [13], auto-increment ranges beyond $\pm 1$ [14], [15], [16], AGUs with modulo addressing modes [17], exploitation of scheduling freedom in the variable access sequence [9], as well as procedure-level offset assignment [18]. Other researchers have dealt with DSP-specific compiler techniques for address register assignment in case of arrays and predefined memory layouts (e.g. [2], [19], [20], [21], [22], [23]), which are not directly related to SOA.

## 3 Evaluation Methodology

Summarizing the discussion of SOA algorithms in section 2, many techniques have not been directly compared to each other so far, while the few comparisons that do exist are mostly based on small data bases or random problem instances. However, random instances generally do not well reflect real-world problems, since the latter tend to show higher locality in the variable access sequences.

### 3.1 OffsetStone Benchmarks

For sake of a more reliable and reproducible evaluation of available SOA algorithms, we have composed OffsetStone, a large suite of SOA problem instances extracted from 31 complex real-world application programs written in ANSI C. These include computation-intensive DSP applications (e.g. MPEG2, MP3, ADPCM, DSPStone, FFT, JPEG, GSM, Viterbi) but also more control-dominated standard applications (e.g. GZIP, FLEX, BISON, CPP). Altogether, the C applications chosen for OffsetStone comprise more than 300,000 lines of C source code. They are certainly representative and much broader than what has been used for SOA benchmarking in previous work.

From a benchmarking viewpoint, an interesting observation is that there are no significant differences in the behavior of the SOA algorithms for different benchmark types (i.e. DSP or general-purpose). Therefore, there was no need to restrict the evaluation to DSP applications only.

For each application program, we extracted SOA problem instances by means of the following steps:

1. The ANSI C sources for the application are translated into a three address code intermediate representation (IR) by means of the LANCE C frontend [24], in order to make the variable access sequences explicit. Additionally, this step inserts temporary variables for intermediate results, that a compiler would normally generate.
2. The IR is optimized by standard techniques used in most compilers, including common subexpression elimination, dead code elimination, constant folding, jump optimization, etc. This step ensures that the IR does not contain superfluous variables and computations, which a compiler would eliminate anyway.

3. From the optimized IR, the detailed variable access sequence is extracted from each basic block.
4. Since any offset assignment is valid throughout an entire C function, one *global* access graph is constructed per function by merging[1] the local access graphs of the basic blocks. In this way, all local access sequences are represented in a single graph. Each global access graph forms one instance of the SOA problem.

With this methodology we obtained a total of more than 3000 realistic SOA problem instances[2]. The extraction is restricted to variables fitting into a single memory word, i.e., variables that directly qualify for offset assignment. We also excluded pointer variables, since these are mostly allocated in address registers and not on the stack frame.

Our approach assumes that all variables extracted will actually be assigned to the stack. This is not necessarily true, since a compiler generally will be able to keep some of the variables in the data path registers. However, as DSPs with AGUs typically show very few data path registers, it is reasonable to assume that the extracted sequences are very close to the actual access sequences in compiled code.

### 3.2 SOA Algorithms Included in OffsetStone

For the extracted benchmarks, we evaluated the following 8 SOA algorithms:

1. **SOA-OFU**: A trivial offset assignment algorithm, where variables are assigned to offsets in the order of their first use in the code. This order would typically be used in non-optimizing compilers without a dedicated SOA phase, and thus serves as a baseline case for our experiments.
2. **SOA-Bartley:** Bartley's SOA heuristic [5] based on the access graph model.
3. **SOA-Liao:** Liao's SOA heuristic [3] based on the access graph model.
4. **SOA-BB**: Liao's branch-and-bound algorithm [1] for optimally solving SOA.
5. **SOA-TB**: SOA-Liao extended by the tie-break heuristic proposed in [4].
6. **SOA-GA**: The genetic algorithm for SOA from [11].
7. **SOA-INC**: The incremental SOA algorithm from [12], using SOA-Liao for constructing initial solutions.
8. **SOA-INC-TB**: A new combination of SOA algorithms, using SOA-INC in combination with SOA-TB for constructing initial solutions. As will be shown later, using SOA-TB instead of SOA-Liao in total results in a higher optimization potential for SOA-INC.

[1] In this formulation, SOA minimizes *code size*. For *performance* optimization, profiling information can be exploited by assigning higher edge weights to frequently executed program paths.

[2] The OffsetStone benchmark access sequences are available from the author upon request, including the corresponding tools for access sequence extraction and the C++ source code for our implementation of the 8 SOA algorithms. This allows other researchers to easily reproduce the results, to add more offset assignment algorithms to the existing infrastructure, and to extend the benchmark suite by extracting access sequences from further application programs.

## 4 Experimental Results

The 8 SOA algorithms have been implemented in C++ in the form of different routines within a single driver program. Naturally, high attention has been paid to uniform software engineering practices, in order to ensure a fair comparison. The algorithms have been applied to all OffsetStone benchmarks, where the costs (according to the metric defined in section 2) and the CPU times (on a 1.3 GHz Linux PC) have been measured. An exception, however, is the SOA-BB algorithm. Due to the sometimes excessive runtime requirements, we restricted its use to problem instances with at most 12 variables (this already corresponds to $12! \approx 479 \cdot 10^6$ possible solutions).

### 4.1 Performance Relative to SOA-OFU

We first focus on a comparison to the "naive" algorithm SOA-OFU. Table 1 gives the average percentage of the solution cost of 6 SOA algorithms (SOA-OFU set to 100%, SOA-BB not included here due to runtime limitations). SOA-Bartley and SOA-Liao are combined into a single column since they always produce identical results.

The line labeled "average" in table 1 shows the average cost values over all OffsetStone benchmarks. As can be seen, all SOA algorithms reduce the cost as compared to SOA-OFU by about 25% on average, with a relatively small difference to each other (the reason for this will become clear in table 3). The best results are produced by SOA-GA, followed by SOA-INC-TB and SOA-TB.

For sake of completeness, we also applied the algorithms to random access sequences, as it has been frequently done in previous work. The line labeled "random" in table 1 shows the average results obtained after applying the SOA algorithms to a set of 3000 random SOA problem instances with varying numbers of variables and access sequence lengths as they typically occur in practice. Even though the order of result quality does not change, the performance difference between the algorithms is smaller, and the result quality as compared to the naive algorithm SOA-OFU is much lower ($< 8\%$) than for real SOA problems. This can be explained by the fact that the edge weights in the access graph are more uniformly distributed for random sequences than for real sequences. This means that there are no big "peaks" in the objective function so that even optimal SOA solutions are not much better than naive (SOA-OFU) solutions. Hence, the optimization potential for SOA algorithms is significantly lower. This confirms our above statement that random problem instances are not the best choice for evaluating SOA algorithms.

### 4.2 Runtimes

Table 2 shows results on the average runtime requirements (CPU milliseconds) per SOA problem instance. SOA-OFU is not included, since it requires essentially no processing time at all. Note that SOA-Bartley in its original form can only be used for small problem sizes, due to its very high runtime requirements. However, we found that it can be easily accelerated by temporarily suppressing the zero-weight edges in the access graph. The first line in table 2, therefore, refers to this improved implementation of SOA-Bartley. Nevertheless, its "twin" algorithm SOA-Liao is still faster on average.

**Table 1.** Relative cost of SOA algorithms compared to SOA-OFU solutions (100%)

| benchmark | Liao | TB | INC | INC-TB | GA |
|---|---|---|---|---|---|
| 8051sim | 83.1 | 79.8 | 80.7 | 79.0 | 79.0 |
| adpcm | 81.1 | 79.3 | 80.1 | 78.6 | 78.5 |
| anagram | 68.9 | 66.9 | 68.2 | 66.2 | 65.6 |
| anthr | 81.1 | 79.9 | 80.9 | 79.9 | 79.9 |
| bdd | 78.6 | 76.9 | 78.4 | 76.9 | 76.9 |
| bison | 78.2 | 77.1 | 78.1 | 77.0 | 77.0 |
| cavity | 85.1 | 82.4 | 84.6 | 82.2 | 82.2 |
| cc65 | 78.4 | 76.3 | 77.2 | 76.3 | 76.2 |
| codecs | 81.5 | 80.3 | 81.4 | 80.3 | 80.3 |
| cpp | 77.4 | 76.3 | 77.3 | 76.3 | 76.3 |
| dct | 77.6 | 77.8 | 77.6 | 77.4 | 77.4 |
| dspstone | 76.4 | 74.4 | 76.0 | 74.3 | 74.3 |
| eqntott | 65.0 | 65.0 | 65.0 | 65.0 | 65.0 |
| f2c | 73.7 | 72.7 | 73.6 | 72.6 | 72.6 |
| fft | 92.0 | 92.0 | 92.0 | 92.0 | 92.0 |
| flex | 71.3 | 69.3 | 71.0 | 69.3 | 69.3 |
| fuzzy | 77.5 | 74.2 | 77.0 | 74.2 | 74.2 |
| gif2asc | 83.1 | 82.0 | 83.0 | 81.7 | 81.7 |
| gsm | 81.5 | 80.9 | 81.3 | 80.9 | 80.8 |
| gzip | 77.1 | 73.2 | 76.3 | 73.2 | 73.2 |
| h263 | 70.3 | 70.0 | 70.0 | 70.0 | 69.6 |
| hmm | 70.5 | 67.4 | 69.8 | 67.3 | 67.3 |
| jpeg | 73.7 | 71.8 | 73.4 | 71.7 | 71.6 |
| klt | 68.2 | 66.1 | 67.6 | 66.1 | 66.0 |
| lpsolve | 78.1 | 77.1 | 77.8 | 77.1 | 77.1 |
| motion | 90.6 | 91.1 | 90.6 | 89.6 | 89.6 |
| mp3 | 72.3 | 71.6 | 72.2 | 71.6 | 71.4 |
| mpeg2 | 77.0 | 76.0 | 76.8 | 75.9 | 75.8 |
| sparse | 75.9 | 75.1 | 75.9 | 75.1 | 75.1 |
| triangle | 65.8 | 64.4 | 65.6 | 64.4 | 64.3 |
| viterbi | 89.3 | 85.0 | 89.1 | 84.9 | 84.9 |
| **average** | 76.71 | 75.23 | 76.40 | 75.16 | 75.10 |
| **random** | 92.74 | 92.24 | 92.62 | 92.17 | 92.13 |

**Table 2.** Average runtime per problem instance

| Algorithm | CPU time (msecs) |
|---|---|
| SOA-Bartley | 0.97 |
| SOA-Liao | 0.67 |
| SOA-TB | 0.68 |
| SOA-INC | 4.60 |
| SOA-INC-TB | 23.00 |
| SOA-GA | 8296.26 |

**Table 3.** Average overhead compared to optimum

| Algorithm | % overhead |
|---|---|
| SOA-BB | 0.00 |
| SOA-OFU | 67.09 |
| SOA-Liao | 4.34 |
| SOA-TB | 0.16 |
| SOA-INC | 2.28 |
| SOA-INC-TB | 0.11 |
| SOA-GA | 0.00 |

The average runtimes are mostly in the order of milliseconds or even less, with SOA-Liao and SOA-TB being the fastest algorithms. There is a big gap to SOA-GA though, which on average needs about 8.3 CPU seconds per problem instance. This leads to a clear separation of SOA algorithms into *fast* and *slow* ones, where the latter category comprises SOA-GA and SOA-BB.

### 4.3 Performance Relative to Optimum

For about 41% of all benchmark problems (i.e. the "small" problems with at most 12 variables), we computed optimal solutions by means of the SOA-BB algorithm. This allowed us to measure the absolute quality of computed SOA solutions. The results are given in table 3, which shows the average percentage of cost overhead compared to the optimal solutions for each algorithm. Naturally, the trivial algorithm SOA-OFU shows the highest overhead. As can be seen, all heuristics get more or less close to the optimum, which explains the small differences found in table 1. SOA-Liao yields an average overhead of 4.34%, while SOA-INC-TB is the best of the fast heuristics, with an overhead of only 0.11%. SOA-GA found the optimum in all cases. For the test cases covered by table 3, SOA-BB needed about 3.5 CPU seconds per SOA instance, while SOA-GA took 0.8 CPU seconds. The CPU times of the fast heuristics are negligible in practice.

## 5 Conclusions

Given that the OffsetStone benchmarks provide a good representation of real-world SOA problems, the experimental data from section 4 permit to draw the following conclusions that were not available from previous work:

- Generally, the performance difference between SOA algorithms for real problems is surprisingly small. Hence, it might appear that the concrete algorithm used in a C compiler for DSPs does not matter much. However, under the tight cost constraints of embedded systems where sometimes every program ROM word matters, the best algorithm with an acceptable runtime should certainly be chosen.
- SOA-Bartley can be easily implemented much more efficiently than in the originally proposed form, but SOA-Liao is still faster while giving the same results.

- SOA-TB achieves better average results for real-life problems than SOA-Liao/SOA-Bartley at virtually no increase in computation time, and it also achieves better solutions than SOA-INC.
- The new combination of SOA algorithms (SOA-INC-TB) proposed in this paper achieves the best results of all fast heuristics tested here. Hence, it can be recommended for fast compilers and can replace the use of SOA-Bartley/Liao, SOA-TB, and SOA-INC. At least for "small" problems it achieves an extremely low average overhead compared to optimal solutions.
- In case priority is given to highest code quality and not to high compilation speed (say, in a final compiler run with highest optimization effort to generate production code with minimal ROM size), the SOA-GA algorithm should be preferred. For "small" SOA problems, SOA-BB can be used to compute optimal solutions, but we observed that SOA-GA finds the optimum in virtually all cases (even though it is not guaranteed to do so) at less than 25% of the computation time requirements of SOA-BB. SOA-BB is frequently fast but sometimes shows extreme peaks in computation time due to its branch-and-bound nature, whereas the runtimes of SOA-GA are predictable.
- The use of random access sequences for evaluation of SOA algorithms, though quite common in previous research, does not accurately reflect the algorithm behavior for real applications. Our experimental results indicate that random sequences do allow for a coarse performance comparison between algorithms, but they definitely do not exhibit their optimization potential for real-life application code.

OffsetStone is the first effort towards fair benchmarking of offset assignment algorithms based on a huge suite of realistic problem instances. It allowed us to provide an in-depth evaluation of most state-of-the-art SOA algorithms. The results provide valuable hints both for compiler developers and researchers working on offset assignment in C compilers for DSPs. As a secondary contribution, we were able to identify a new combination of fast heuristics (SOA-INC-TB) that is superior to previous algorithms.

As a first step, in this paper we have focused only on SOA, the most basic class of offset assignment problems. In the future, the suite of algorithms included in OffsetStone will be extended to also cover generalized offset assignment problem formulations, e.g. offset assignment with variable live range information, exploitation of scheduling mobility of instructions, or general offset assignment with multiple address registers, some of which have been mentioned in section 2.

## References

1. S. Liao: *Code Generation and Optimization for Embedded Digital Signal Processors*, Ph.D. thesis, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1996
2. S. Udayanarayanan, C. Chakrabarti: *Address Code Generation for Digital Signal Processors*, 38th Design Automation Conference (DAC), 2001
3. S. Liao, S. Devadas, K. Keutzer, S. Tjiang, A. Wang: *Storage Assignment to Decrease Code Size*, ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 1995

4. R. Leupers, P. Marwedel: *Algorithms for Address Assignment in DSP Code Generation*, Int. Conference on Computer-Aided Design (ICCAD), 1996
5. D.H. Bartley: *Optimizing Stack Frame Accesses for Processors with Restricted Addressing Modes*, Software – Practice and Experience, vol. 22(2), 1992
6. Target Compiler Technologies: *http://www.retarget.com*
7. M.R. Gary, D.S. Johnson: *Computers and Intractability – A Guide to the Theory of NP-Completeness*, Freemann, 1979
8. B. Wess: *Minimization of Data Address Computation Overhead in DSPs*, 3rd Int. Workshop on Code Generation for Embedded Processors (SCOPES), 1998
9. A. Rao, S. Pande: *Storage Assignment using Expression Tree Transformations to Generate Compact and Efficient DSP Code*, ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 1999
10. V. Zivojnovic, J.M. Velarde, C. Schläger, H. Meyr: *DSPStone – A DSP-oriented Benchmarking Methodology*, Int. Conf. on Signal Processing Applications and Technology (ICSPAT), 1994
11. R. Leupers, F. David: *A Uniform Optimization Technique for Offset Assignment Problems*, 11th Int. System Synthesis Symposium (ISSS), 1998
12. S. Atri, J. Ramanujam, M. Kandemir: *Improving Offset Assignment for Embedded Processors*, Languages and Compilers for High-Performance Computing, S. Midkiff et al. (eds.), Lecture Notes in Computer Science, Springer, 2001
13. N. Sugino, H. Miyazaki, S. Iimuro, A. Nishihara: *Improved Code Optimization Method Utilizing Memory Addressing Operations and its Application to DSP Compilers*, Int. Symp. on Circuits and Systems (ISCAS), 1996
14. B. Wess, M. Gotschlich: *Constructing Memory Layouts for Address Generation Units Supporting Offset 2 Access*, Proc. ICASSP, 1997
15. N. Kogure, N. Sugino, A. Nishihara: *Memory Address Allocation Method for a DSP with* $\pm$ *2 Update Operations in Indirect Addressing*, European Conference on Circuit Theory and Design (ECCTD), 1997
16. A. Sudarsanam, S. Liao, S. Devadas: *Analysis and Evaluation of Address Arithmetic Capabilities in Custom DSP Architectures*, Design Automation Conference (DAC), 1997
17. B. Wess, M. Gotschlich: *Optimal DSP Memory Layout Generation as a Quadratic Assignment Problem*, Int. Symp. on Circuits and Systems (ISCAS), 1997
18. E. Eckstein, A. Krall: *Minimizing Cost of Local Variables Access for DSP Processors*, ACM Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES), 1999
19. C. Liem, P.Paulin, A. Jerraya: *Address Calculation for Retargetable Compilation and Exploration of Instruction-Set Architectures*, 33rd Design Automation Conference (DAC), 1996
20. C. Gebotys: *DSP Address Optimization Using a Minimum Cost Circulation Technique*, Int. Conference on Computer-Aided Design (ICCAD), 1997
21. R. Leupers, A. Basu, P. Marwedel: *Optimized Array Index Computation in DSP Programs*, Asia South Pacific Design Automation Conference (ASP-DAC), 1998
22. W.-K. Cheng, Y.-L. Lin: *Addressing Optimization for Loop Execution Targeting DSP with Auto-Increment/Decrement Architecture*, 11th Int. System Synthesis Symposium (ISSS), 1998
23. G. Ottoni, S. Rigo, G. Araujo, S. Rajagopalan, S. Malik: *Optimal Live Range Merge for Address Register Allocation in Embedded Programs*, 10th International Conference on Compiler Construction (CC), 2001
24. LANCE C Compiler: *http://LS12-www.cs.uni-dortmund.de/lance*