Memory Offset Assignment for DSPs

Jinpyo Hong[1] and J. Ramanujam[2]

[1] School of Internet-Media Engineering
Korea University of Technology and Education, Cheonan, Korea
jphong1@kut.ac.kr
[2] Dept. of Electrical and Computer Engineering
Louisiana State University, Baton Rouge, LA, USA
jxr@ece.lsu.edu

Abstract. Compact code generation is very important for an embedded system that has to be implemented on a chip with a severely limited amount of size. Even though on-chip data memory optimization technique has been given more attention, on-chip instruction memory optimization should not be neglected. We propose in this paper some algorithms for a memory offset assignment for embedded DSP processors in order to minimize the number of instructions for address register operations. Extensive experimental results demonstrate the efficacy of our solution.

1 Introduction

Embedded DSP processors contain an *address generation unit* (AGU) that enables the processor to compute the address of an operand of the next instruction while executing the current instruction. An AGU has auto-increment and auto-decrement capability, which can be done in the same clock of execution of a current instruction. It is very important to take advantage of AGUs in order to generate high-quality compact code. In this paper, we propose heuristics for the *single offset assignment with modify registers* (SOA-MR) problem and the *general offset assignment* (GOA) problem in order to exploit AGUs effectively. Experimental results show that our proposed methods can reduce address operation cost and in turn lead to compact code. The storage assignment problem was first studied by Bartley [6] and Liao [8,9,10]. Liao showed that the offset assignment problem even for a single address register is NP-complete and proposed a heuristic that uses the *access graph,* which can be constructed from a given access sequence. The access graph has one vertex per variable and edges between two vertices in the access graph indicate that the variables corresponding to the vertices are accessed consecutively; the weight of an edge is the number of times such consecutive access occurs. Liao's solution picks edges in the access graph in decreasing order of weight as long as they do not violate the assignment requirement. Liao also generalizes the storage assignment problem to include any number of address registers. Leupers and Marwedel [11] proposed a tie-breaking function to handle the same weighted edges, and a variable partitioning strategy to minimize GOA costs. They also show that the storage assignment cost can be reduced by utilizing modify registers. In [1,2,3,14], the interaction between instruction selection and scheduling is considered in order to improve code size. Rao and Pande [13] apply algebraic transformations to find a better

Y.-H. Lee et al. (Eds.): ICESS 2007, LNCS 4523, pp. 80–87, 2007.

access sequence. They define the least cost access sequence problem (LCAS), and propose heuristics to solve the LCAS problem. Other work on transformations for offset assignment includes those of Atri et al. [4,5] and Ramanujam et al. [12]. Recently, Choi and Kim [7] presented a technique that generalizes the work of Rao and Pande [13].

The remainder of this paper is organized as follows. In Section 2 and 3, we propose our heuristics for SOA with modify registers, and GOA problems. We also explain the basic concepts of our approach. In Section 4, we present experimental results. Finally, Section 5 provides a summary.

2 Our Approach to the SOA-MR Problem

2.1 The Single Offset Assignment (SOA) Problem

Given a variable set $V = \{v_0, v_1, \cdots, v_{n-1}\}$, the single offset assignment (SOA) problem is to find the offset of each variable $v_i, 0 \leq i \leq n-1$ so as to minimize the number of instructions needed only for memory address operations. In order to do that, it is very critical to maximize auto-increment/auto-decrement operations of an address register that can eliminate the explicit use of memory address instructions.

Liao [8] proposed a heuristic that finds a path cover of an access graph $G(V, E)$ by choosing edges in decreasing order of the number of transitions in an access sequence while avoiding cycles, but he does not say how to handle edges that have the same weight. Leupers and Marwedel [11] introduced a tie-breaking function to handle such edges. Their result is better than Liao's as expected.

2.2 Our Algorithm for SOA with an MR

Definition 1. *An edge $e = (v_i, v_j)$ is called an* uncovered edge *when variables that correspond to vertices v_i and v_j are not assigned adjacently in a memory.*

After applying the existing SOA heuristic to an access graph $G(V, E)$, we may have several paths. If there is a Hamiltonian path and SOA luckily finds it, then memory assignment is done, but we cannot expect that situation all the time. We prefer to call those paths partitions because each path is disjoint with others.

Definition 2. *An uncovered edge $e = (v_i, v_j)$ is called an intra-uncovered edge when variables v_i and v_j belong to the same partition. Otherwise, it is called an inter-uncovered edge. These are also referred to as intra-edge and an inter-edge respectively.*

Definition 3. *Each intra-edge and inter-edge contributes to an address operation cost. We call these the intra-cost and the inter-cost respectively.*

Uncovered edges account for cost if they are not subsumed by an MR register. Our goal is to maximize the number of uncovered edges that are subsumed by an MR register. The cost can be expressed by the following cost equation.

$$cost = \sum_{e_i \in intra\_edge} intra\_cost(e_i) + \sum_{e_j \in inter\_edge} inter\_cost(e_j).$$

It is clear that a set of intra-edges and a set of inter-edges are disjoint because from Definition 2, an uncovered edge e cannot be an intra-edge and an inter-edge at the same time. First, we want to maximize the number of intra-edges that are subsumed by an MR register. After that, we will try to maximize the number of inter-edges that will be subsumed by an MR register. We think this approach is reasonable because when the memory assignment is fixed by a SOA heuristic, there is no flexibility of intra-edges in such a sense that we cannot rearrange them. So, we want to recover as many intra-edges as possible with an MR register first. Then, with the observation that we can change the distances of inter-edges by rearranging partitions, we will try to recover inter-edges with an MR register.

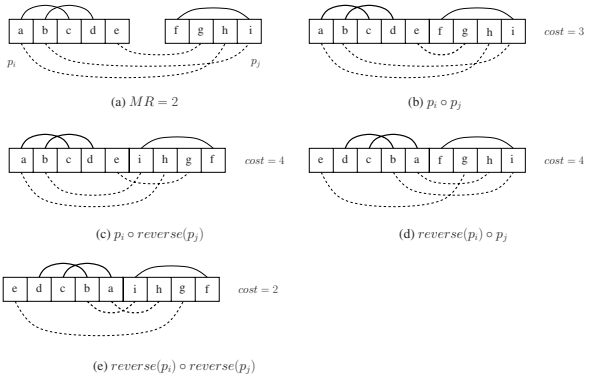

Fig. 1. Merging combinations

There are four possible merging combinations of two partitions. Figure 1 shows those four merging combinations. Intra-edges are represented by a solid line, and inter-edges by a dotted line. In Figure 1-(a), there are 6 uncovered edges among which there are 3 intra-edges and 3 inter-edges. So, the AR cost is 6. First, we try to find the most frequently appearing distance of intra-edges. In this example, distance 2 is the one because $distance(a, c)$ and $distance(b, d)$ are 2 and $distance(f, i)$ is 3. By assigning 2 to an MR register, we can recover two out of three intra-edges, which reduces the cost by 2. When an uncovered edge is recovered by an MR register, the corresponding line is depicted by a thick line. Next, we want to recover as many inter-edges as possible by making the distance of inter-edges 2 by applying proper merging combination. In Figure 1-(b), the two partitions are concatenated. One inter-edge,$e = (e, g)$ will be recovered, because $distance(e, g)$ in a merged partition is 2. So, the cost is 3. In Figure 1-(c), the first partition is concatenated with the reversed second one. No inter-edge will be recovered. The cost is 4. In Figure 1-(d), the reversed first partition is concatenated with the second one. No inter-edge will be recover, either. The cost is 4. In Figure 1-(e), the two partitions are reversed and concatenated. It is actually equal to exchanging the two partitions. Two inter-edges will be recovered. In this case, we recover four out of six uncovered edges by applying our method. Figure 2 shows our MR optimization algorithm.

Procedure SOA_mr
begin
$G_{partition}(V_{par}, E_{par}) \leftarrow$ Apply SOA to $G(V, E)$;
$\Phi_{m\_sorted} \leftarrow$ sort m values of edges (v_1, v_2) by frequency in descending order;
$M \leftarrow$ the first m of Φ_{m_sorted};
$optimizedSOA \leftarrow \phi$;

for each partition pair of p_i and p_j **do**
Find the number, $m_{(p_i,p_j)}$ of edges, $e = (v_1, v_2)$, $e \in E$, $v_1 \in p_i$, $v_2 \in p_j$ such that their distance (m value) $= M$ from four possible merging combinations, and assign a rule number that can generate $m = M$ most frequently to (p_i, p_j);
enddo

$\Psi_{sorted\_par\_pair} \leftarrow$ Sort partition pairs (p_i, p_j) by $m_{(p_i,p_j)}$ in descending order;

while ($\Psi_{sorted\_par\_pair} \neq \phi$) **do**
$(p_i, p_j) \leftarrow$ choose the first pair from $\Psi_{sorted\_par\_pair}$;
$\Psi_{sorted\_par\_pair} \leftarrow \Psi_{sorted\_par\_pair} - \{(p_i, p_j)\}$;
if ($p_i \notin optimizedSOA$ and $p_j \notin optimizedSOA$)
$optimizedSOA \leftarrow (optimizedSOA \circ merge\_by\_rule(p_i, p_j))$;
$V_{par} \leftarrow V_{par} - \{p_i, p_j\}$;
endif
enddo

while ($V_{par} \neq \phi$) **do**
Choose p from V_{par};
$V_{par} \leftarrow V_{par} - \{p\}$;
$optimizedSOA \leftarrow (optimizedSOA \circ p)$;
enddo

return $optimizedSOA$;
end

Fig. 2. Heuristic for SOA with MR

3 General Offset Assignment (GOA)

The general offset assignment problem is, given a variable set $V = \{v_0, v_1, \cdots, v_{n-1}\}$ and an AGU that has k ARs, $k > 1$, to find a partition set $\mathcal{P} = \{p_0, p_1, \cdots, p_{l-1}\}$, where $p_i \cap p_j = \phi, i \neq j, 0 \leq i, j \leq l-1$, subject to minimize GOA cost $\sum_{i=0}^{l-1} SOA\_cost(p_i)$ $+l$, where l is the number of partitions, $l \leq k$. The second term l is the initialization cost of l ARs. Our GOA heuristic consists of two phases. In the first phase, we sort variables in descending order of their appearance frequencies in an access sequence, i.e., the number of accesses to a particular variable. Then, we construct a partition set $\mathcal{P}$ by selecting the two most frequently appearing variables, which will reduce the length of the remaining access sequence most, and making them a partition, $p_i, 0 \leq i \leq l-1$. After the first phase, the way we construct a partition set $\mathcal{P}$, we will have $l, l \leq k$,

partitions that consist of only 2 variables each. Those partitions have zero SOA cost, and we have the shortest access sequence that consists of $(|V| - 2l)$ variables. In the second phase, we pick a variable v from the remaining variables in the descending order of frequency, and choose a partition p_i such that $SOA\_cost(p_i \cup \{v\})$ is increased minimally, which means that merging a variable v into that partition increases the GOA cost minimally. This process will be repeated $(|V| - 2l)$ times, till every variable is assigned to some partition.

4 Experimental Results

We generated access sequences randomly and apply our heuristics, Leupers' and Liao's. We repeated the simulation 1000 times on several problem sizes. Our experiments show that introducing an MR can improve the AGU performance and that an optimization heuristic for an MR register is needed to maximize a performance gain. Our experiments show that the results of 2-AR AGU are alway better than 1AR_1MR's and even ARmr_op's. It is because even if we apply a MR optimization heuristic, which is naturally to be more conservative than GOA heuristic of 2-AR in such a sense that only after several path partitions are generated by SOA heuristic on entire variables, a MR optimization heuristic would try to recover uncovered edges whose occurrences heavily depend on SOA heuristic. A GOA heuristic can exploit a better chance by partitioning variables into two sets and applying SOA heuristic on each partitioned set. However, GOA's gain over ARmr_op does not come for free. The cost of the partitioning of variables might not be negligible as it was shown in section 3. However, from the perspective of performance of an embedded system, our experiment shows that it is better to pay that cost to get performance gain of AGU. The gain of 2-AR GOA over ARmr_op is noticeable enough to justify our opinion. When an AGU has several pairs of a AR and an MR, in which AR[i] is coupled with MR[i], our path partition optimization heuristic can be used for each partitioned variable set. Then, the result of each pair of the AGU will be improved as we observed in Figure 3. Figures 3 shows bar graphs based on the results of randomly generated access sequences. When an access graph is dense, two heuristics perform similarly as shown in Figure 3-(a). In this case, introducing a mr

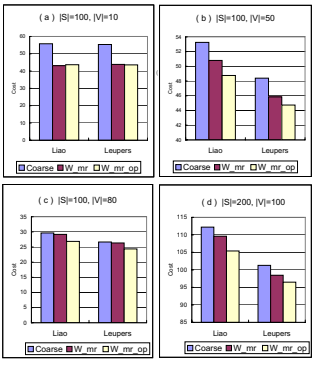

Fig. 3. Results for SOA and SOA_mr

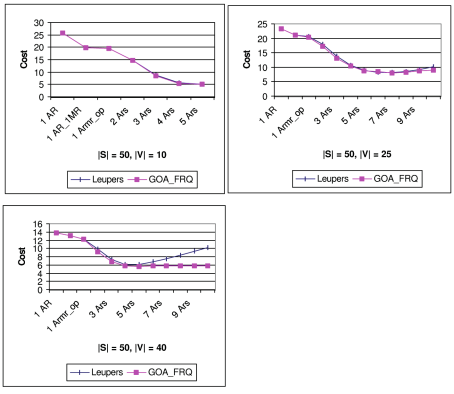

Fig. 4. Results for GOA_FRQ

optimization technique does not improve performance much. Figure 3-(b), 3-(d) show that when the number of variables is 50% of th length of an access sequence, introducing optimization technique can reduce the costs. Figure 3-(c) shows that when the access graph becomes sparse, the amount of improvement becomes smaller than when the graph is dense, but it is still reduce the costs noticeably. Except the case when an access graph is very dense like in Figure 3-(a), applying our mr optimization technique is beneficial in all heuristics including Liao's and Leupers'. Figure 4 shows that our GOA_FRQ algorithm outperforms Leupers' in many cases. Especially in Figure 4, we can witness that beyond certain threshold, our algorithm keeps its performance stable. However, Leupers' algorithm tries to use as many ARs as possible, which makes performance of his algorithm deteriorated as the number of ARs grows. Line graphs in Figure 4 shows that our mr optimization technique is beneficial, and that 2 ARs configuration always outperforms ar_mr_op as we mentioned earlier.

We experiment DSP benchmarks like BIQUAD_ONE, COMP (Complex multiplication), and ELLIP (Elliptical wave filter) and also numerical analysis algorithms like GAULEG (Gauss-Legendre weights and abscissas), GAUHER (Gauss-Laguerre weights and abscissas) and GAUJAC (Gauss-Jacobi weights and abscissas) from [15]. We also use several programs such as CHENDCT, CHENIDCT, LEEDCT and LEEIDCT from JPEG-MPEG package. Figure 5 shows the improvements of results of 1AR_1MR, ARmr_op, and 2 ARs to 1 AR. Improvement is computed as $(\frac{1AR-x}{1AR} \times 100)$, where x is one of the above three AGUs. Except COMP which is too simple to show a meaningful result, introducing extra resource (MR) in AGU is always beneficial. The average improvement of rest 5 algorithms of including MR is 18.5%. With the same amount of resources (1 AR and 1 MR), we achieve more gains by applying our MR optimization technique. The average improvement of our MR optimization is 25.4%. The average improvement of 2 ARs is 44.2%. MR takes a supplemental role to recover edges that were not included in path covers. With understanding such a role of MR, superiority of the result of 2ARs over MR and MR_OP is understandable. However, we believe that improvement of our MR optimization technique shows that more

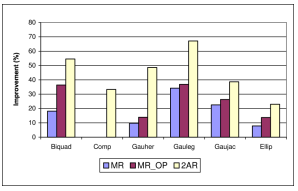

Fig. 5. Improvements of 1AR_1MR, MR_OP and 2ARs to 1 AR

aggressive method for MR optimization should be enforced and that MR be given more attention in a sense that setting value 1 to MR has an exactly same effect as AR's auto-increment/-decrement, which means MR has more flexibility than `AR++` and `AR--`. Our MR optimization technique can be used to exploit $m \geq 1$ pairs of (AR,MR) in AGU.

5 Summary

We have found that several fragmented paths are generated as the SOA algorithm trys to find a path cover. We have proposed a new optimization technique of handling these fragmented paths. As the SOA algorithm generates several fragmented paths, we show that our optimization technique of these path partitions is crucial to achieve an extra gain, which is clearly captured by our experimental results. We also have proposed usage of frequencies of variables in a GOA problem. Our experimental results show that this straightforward method is better than the previous research works.

Acknowledgments. This work is supported in part by the US National Science Foundation through awards 0073800, 0103933, 0121706, 0508245, 0509442 and 0541409.

References

1. G. Araujo. Code Generation Algorithms for Digital Signal Processors. PhD thesis, Princeton Department of EE, June 1997.
2. G. Araujo, S. Malik, and M. Lee. Using Register-Transfer Paths in Code Generation for Heterogeneous Memory-Register Architectures. In *Proceedings of 33rd ACM/IEEE Design Automation Conference*, pages 591-596, June 1996.
3. G. Araujo, A. Sudarsanam, and S. Malik. Instruction Set Design and Optimization for Address Computation in DSP Architectures. In *Proceedings of the 9th International Symposium on System Synthesis*, pages 31-37, November 1997.
4. S. Atri, J. Ramanujam, and M. Kandemir. Improving offset assignment on embedded processors using transformations. In *Proc. High Performance Computing–HiPC 2000,* pp. 367–374, December 2000.
5. Sunil Atri, J. Ramanujam, and M. Kandemir. Improving variable placement for embedded processors. In *Languages and Compilers for Parallel Computing,* (S. Midkiff et al. Eds.), Lecture Notes in Computer Science, vol. 2017, pp. 158–172, Springer-Verlag, 2001.
6. D. Bartley. Optimization Stack Frame Accesses for Processors with Restricted Addressing Modes. *Software Practice and Experience,* 22(2):101-110, February 1992.

7. Y. Choi and T. Kim. Address assignment combined with scheduling in DSP code generation. in *Proc. 39th Design Automation Conference,* June 2002.
8. S. Liao. Code Generation and Optimization for Embedded Digital Signal Processors. PhD thesis, MIT Department of EECS, January 1996.
9. S. Liao et al. Storage Assignment to Decrease Code Size. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation,* pages 186–196, 1995. (This is a preliminary version of [10].)
10. S. Liao, S. Devadas, K. Keutzer, S. Tjiang, and A. Wang. Storage assignment to decrease code size. *ACM Transactions on Programming Languages and Systems,* 18(3):235–253, May 1996.
11. R. Leupers and P. Marwedel. Algorithms for Address Assignment in DSP Code Generation. In *Proceedings of International Conference on Computer-Aided Design,* pages 109-112, 1996.
12. J. Ramanujam, J. Hong, M. Kandemir, and S. Atri. Address register-oriented optimizations for embedded processors. In *Proc. 9th Workshop on Compilers for Parallel Computers (CPC 2001),* pp. 281–290, Edinburgh, Scotland, June 2001.
13. A. Rao and S. Pande. Storage Assignment Optimizations to Generate Compact and Efficient Code on Embedded Dsps. *SIGPLAN '99, Atlanta, GA, USA*, pages 128-138, May 1999.
14. A. Sudarsanam and S. Malik. Memory Bank and Register Allocation in Software Synthesis for ASIPs. In *Proceedings of International Conference on Computer Aided Design*, pages 388-392, 1995.
15. W. H. Press, S. A. Teukolsky, W. T. Vetterling and B. P. Flannery (Editors), *Numerical Recipes in C: The Art of Science Computing,* Cambridge University Press, pages 152-155, 1993.