Address Code Generation for DSP Instruction-Set Architectures

J.-Y. LEE
Hynix Semiconductor, Inc.
and
I.-C. PARK
Korea Advanced Institute of Science and Technology

This paper presents a new DSP-oriented code optimization method to enhance performance by exploiting the specific architectural features of digital signal processors. In the proposed method, a source code is translated into the static single assignment form while preserving the high-level information related to the address computation of array accesses. The information is used in generating auto-modification addressing operations provided by most digital signal processors. In addition to the conventional control-data flow graph, a new graph is employed to find auto-modification addressing modes efficiently. Experimental results on benchmark programs show that the proposed method is effective in improving performance and reducing code size.

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

General Terms: Algorithms

Additional Key Words and Phrases: Code size, cycle counts, auto-modification addressing

1. INTRODUCTION

Digital signal processors (DSPs) are increasingly being embedded into many electronic products. Two main trends in the design of embedded systems are clear: the amount of embedded software is growing larger and becoming more complex, and all the components of a system such as DSPs, RAM, ROM, and ASICs are being integrated on a single chip. Traditional approaches to achieve

This work was supported in part by the Korea Science and Engineering Foundation through the MICROS center, and in part by the Ministry of Science and Technology and the Ministry of Commerce, Industry and Energy through the project System IC 2010.
Authors' addresses: J.-Y. Lee, MCU Applications Design Team, Hynix Semiconductor Inc., 1 Hyangjeong-dong, Hungduk-gu Chengju-si, 361-725, Republic of Korea; email: jongyeol.lee@hynix.com; I.-C. Park, Division of Electrical Engineering, Department of EECS, KAIST, 373-1, Guseong-dong, Yuseong-gu, Daejeon, 305-701, Republic of Korea; email: icpark@ics.kaist.ac.kr.

high-quality embedded software have been based on writing the code in assembly language. As the complexity of embedded software grows, however, programming in assembly language and manual optimization become no longer practical except for time-critical portions. These trends mandate the use of high-level languages (HLLs) in order to decrease development cost and time-to-market pressure.

While the conventional code optimization methods [Aho et al. 1986] implemented in most HLL compilers have proved effective for general-purpose processors, direct applications of the conventional code optimization methods cannot efficiently exploit the specific features of DSP architectures, leading to a significant gap between hand-optimized assembly codes and compiler-generated codes. These peculiarities of DSPs remain challenges to compilers.

Compilers for embedded DSPs must take advantage of specialized architectural features that most DSPs provide to make codes meet the constraints of real-time performance. DSPs usually provide separate address generation units (AGUs) that consist of address registers, modify registers, and arithmetic units to calculate addresses in parallel with data computation. The address register can be auto-incremented/auto-decremented by adding or subtracting the value in the modify registers or a constant value. This separate address arithmetic improves both code size and performance by allowing parallel address computations that are otherwise to be performed serially on datapath. In this paper, we focus on code generation for DSPs and present a new method to optimize the address calculation of array accesses by utilizing the auto-increment/auto-decrement capability of AGUs.

The rest of this paper is organized as follows. In Section 2, we describe a short review of related works. The proposed method is presented in Section 3. Experimental results are shown in Section 4 and finally concluding remarks are made in Section 5.

2. PREVIOUS WORKS

Several authors have employed high-level transformations to optimize address calculation. Ottoni et al. [2001] exploits high-level transformation to optimize array accesses for fixed memory layouts. Miranda et al. [1998] focuses on the hardware synthesis of data-transfer-intensive applications. In the work, address expressions are extracted from the control-data flow graph (CDFG) and transformed to reduce the cost overhead associated with the address architecture. The previous works show that the high-level transformation is very effective. However, Ottoni et al. [2001] does not consider the characteristics of AGUs such as modify registers. Transformations used in Miranda et al. [1998] are developed for more efficient architecture exploration, but cannot be directly applied to address code generation.

Previous works such as Gebotys [1997] and Basu et al. [1999] have considered DSP code generation using AGUs. Gebotys uses a minimum circulation technique to solve an address assignment problem. Basu et al. present a heuristic code optimization technique which, given an AGU with a fixed number of address registers, minimizes the number of instructions needed for address

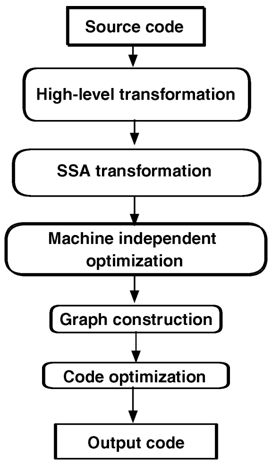


Fig. 1. Overall flow of the proposed method.

computations in loops by converting array accesses to pointer accesses. These works, however, do not use high-level information, hence may fail to generate optimized address code when the source code structure is completely dismantled in the front end of a compiler.

Other works, for example Bartley [1992], Liao et al. [1996], Leupers and Marwedel [1996], and Sudarsanam et al. [1997], have treated the problem of storage assignment. Although the previous works on the storage assignment problem can be applied to determine the relative location of arrays, they cannot be used to determine how to access the elements of an array.

In this work, high-level transformation is employed to obtain the high-level information related to the address computation of array accesses. The information is used to optimize the code by using auto-modification addressing modes. Assuming the relative location of arrays is determined, the accesses to elements within one array are optimized by using auto-modification addressing modes.

3. PROPOSED CODE OPTIMIZATION METHOD

The proposed method assumes that an AGU has post/pre-increment/decrement functionality with a constant offset or a modify register. As depicted in Figure 1, the proposed approach starts from the array access information annotation implemented in the front end of a compiler. The address values of array accesses are identified and the corresponding variables and operations are tagged in the intermediate representation (IR) for the later references in the graph construction step that builds graphs to be used in code optimization. After obtaining the array access information, the code is converted into static single assignment (SSA) form [Cytron et al. 1991; McConnell and Johnson 1992], which is known as an efficient way of representing the data flow of a routine.

Value-trace graphs (VTGs) are constructed after machine-independent optimizations, which are used as a new data-flow representation in addition to the conventional CDFG where non-address computations are intermingled with address computations. In the proposed method, a VTG is constructed based on the high-level information on the address value of an array access. The VTG consists of only the operands involved in the computation of the address value and captures the characteristics of an AGU by including only the operations supported by the AGU. Therefore, it is possible to optimize the array-accessing code by considering only the nodes in the VTG instead of all the operations and operands in the CDFG. The code optimization is performed by finding the flow of the address values of array accesses in the VTG and then transforming the flow into a form in which the auto-modification addressing modes can be used efficiently. In the following sub-sections, each step in Figure 1 is described in more detail.

3.1 Array Access Information Annotation

In this step, the information on array accesses is gathered and stored in the IR. Since it is hard to collect the information after generating the IR code that treats all the memory access in the same way, the information is collected at the source level. In parsing, the array accesses are identified and specially treated. After that, the IR operations and their operands which access the array elements are generated using the information on array accesses. Since the address of an array access is calculated by adding/subtracting the index of the element of an array to/from the start address of the array, the addition or subtraction calculating the array addresses is tagged as an array address calculation and its operands are tagged as either the start address of the array or the index of the array element. This information is used to construct the VTG which plays a major role in the AGU optimization. For example, when two elements of an array are added in a source statement, the address calculations are implemented by two additions, `OP1` and `OP2`, that add an IR variable, `R1`, representing the start address of the array and IR variables, `R2` and `R3`, representing the indices of the elements. In the high-level transformation step, the IR variables, `R1`, `R2` and `R3`, and operations, `OP1` and `OP2`, are tagged with the corresponding array accesses.

3.2 Graph Construction

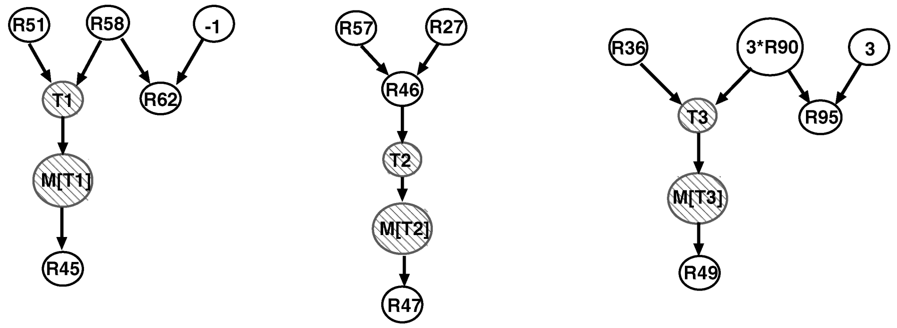
To increase the use of auto-modification addressing modes supported by an AGU, a VTG that captures the characteristics of the AGU is constructed for each address node identified in the array access information annotation step. To construct the VTG, the data flow of a basic block is analyzed and the operations involved in the address computations of array accesses are transformed to plus operations since the auto-modification of an address is performed on the AGU by adding or subtracting the value of a modify register or a constant value. Minus operations are converted into plus operations by changing the sign of operands. In addition, when an address is calculated by adding an offset to a base address in a loop body where a new offset is calculated in a linear fashion by multiplying and adding constants to the offset of the previous iteration, the multiplication can be converted to an addition by inserting appropriate pre-computations. An example is shown in Figure 2(a). A new offset, `R48`, is calculated by `R48 <= R90 * 3` and `R95 <= R90 + 1`, where `R90` is the offset determined by ϕ function. In Figure 2(b), the multiplication is converted into an addition, `R95 <= R99 + 3` by

```
R58 <= φ(R62, R37)
R59 <= φ(R61, R38)
R57 <= φ(R46, R39)
R90 <= φ(R95, R40)
R45 <= *(R51 + R58)
R46 <= R57 + R27
R9  <= R51 + R3
R47 <= *R46
R48 <= R90 * 3
R49 <= *(R36 + R48)
R50 <= R47 * R45
R53 <= R50 * R49
R61 <= R59 + R53
R62 <= R58 - 1
R95 <= R90 + 1
```

(a)

```
R100   <= R40 * 3
R58    <= φ(R62, R37)
R59    <= φ(R61, R38)
R57    <= φ(R46, R39)
R99    <= φ(R95, R100)
T1     <= R51 + R58
R45    <= M[T1]
R46    <= R57 + R27
T2     <= R46
R9     <= R51 + R3
R47    <= M[T2]
T3     <= R36 + R99
R49    <= M[T3]
R50    <= R47 * R45
R53    <= R50 * R49
R61    <= R59 + R53
R62    <= R58 + (-1)
R95    <= R99 + 3
```

(b)



(c)

Fig. 2. An example of VTG generation: **(a)** SSA form representation of the CDFG of a loop body of a benchmark program, *sum*, described in Table I; **(b)** Code after converting operations in (a). The statements shown in italic contain the converted operations. The address nodes and array access nodes are inserted based on the information from the high-level transformation. The address nodes are T1, T2, and T3 and the array access nodes are M[T1], M[T2], and M[T3]; **(c)** VTGs of the address nodes in (b). The shaded nodes are the inserted address nodes and array access nodes. The implied operations are all additions.

inserting a pre-computation, `R100 <= R40 * 3`, that is out of the loop body. This conversion increases the possibility of AGU optimization, as multiplication is not supported in typical AGUs.

A node in a VTG is an operand in a given basic block. To find nodes to be included in the VTG, the CDFG is searched for the nodes that have an effect on an address node and the nodes on which the address node has an effect. In the CDFG, a node, v_a, is said to have an effect on another node, v_b, when there is a dependence relation between them, or v_a is an operand of a ϕ-function whose return value is used in v_b. After that, the nodes affecting the found nodes and the nodes affected by the found nodes are repeatedly included in the VTG until no more nodes are included. Therefore, the VTG contains only the nodes contributing to the computation of an address value.

An edge in a VTG is drawn from a node, v_1, to another node, v_2, when the value of v_1 is used to calculate the value of v_2. As the operations related to address computation are converted to plus operations, the operations implied in the VTG are all addition assignments related to address computation. This leads to easy manipulation of the VTG.

Three VTGs corresponding to Figure 2(b) are shown in Figure 2(c), where the shaded nodes are the address nodes and array access nodes inserted based on the high-level information. It is known from the data-flow analysis on the CDFG that the VTGs have three invariable nodes, R51, R27, and R36, whose values are not changed in the given block. The nodes, R62, R46 and R95 are dead at the end of the block but the values of these nodes are retained as they are used in the next iteration. The VTG of T2 shows that the operands contributing to the computation of T2 are R57, R27, and R46. R46 is computed by adding R57 and R27 since the implied operation is addition as mentioned above. Furthermore, the data-flow analysis shows that R46 is used in the next iteration as a new value of R57.

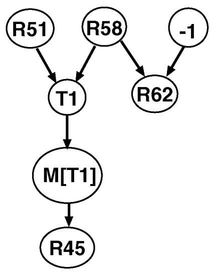
3.3 AGU Optimization

In the proposed AGU optimization, the accesses to an array are optimized by exploiting auto-modification addressing modes using the information on the array accesses collected in the first step and represented in the VTG. To access an element of an array, in the unoptimized IR code, an address register is initialized to point to an appropriate location in the array and the address of the element is calculated by adding a constant or a register to the address register. In the optimized code with auto-modification addressing modes, however, the calculation is replaced by auto-increment or auto-decrement to make the address register point to the element to be accessed next. This optimization is performed in two steps. The flow of the address value represented as an address node is found in the first step by referring to the VTG and then, in the second step, the flow is transformed so that an auto-modification addressing mode can be utilized.

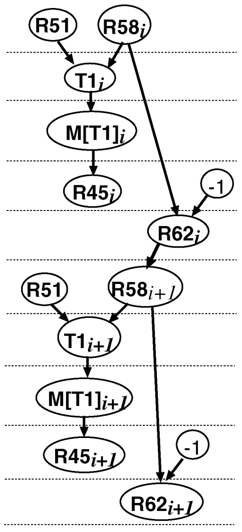
To find the flow of an address value, the VTG consisting of nodes involved in the address computation of an array access is scheduled to generate a *flat schedule* that represents the schedule of operands at a specific iteration. The flat schedule can be obtained by applying the loop unrolling technique used in Basu et al. [1999]. By concatenating the flat schedules of successive iterations, a tree is formed called an address tree, which shows how the next address value of an address node is calculated. The address tree is constructed by selecting the predecessors of the address node in the concatenated schedule and their edges. The number of flat schedules to be concatenated to form an address tree is determined by a loop-carried dependence relation [Wolfe 1996].

```
R58      <= φ(R62, R37)
R59      <= φ(R61, R38)
R57      <= φ(R46, R39)
(3*R90)  <= φ(R95, R40)
T1       <= R51 + R58
R45      <= M[T1]
R46      <= R57 + R27
T2       <= R46
R9       <= R51 + R3
R47      <= M[T2]
T3       <= R36 + (3*R90)
R49      <= M[T3]
R50      <= R47 * R45
R53      <= R50 * R49
R61      <= R59 + R53
R62      <= R58 + (-1)
R95      <= (3*R90) + 3
```

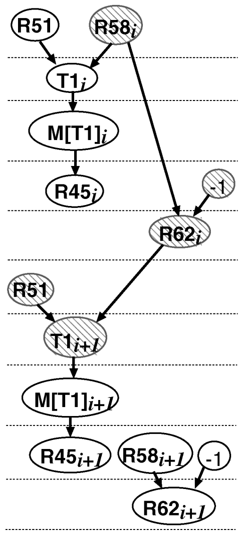
(a)



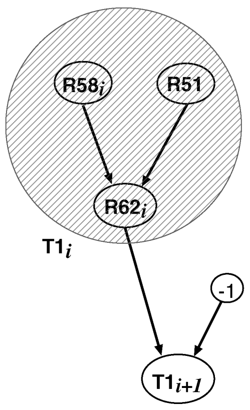
(b)



(c)



(d)



(e)

```
Pre_header:
  R62 <= R51 + R37
  R41 <= 3 * R40
  R97 <= R36 + R41
  R9  <= R51 + R3
  R57 <= R39
  R61 <= R38
Loop_body:
  R45 <= *R62--
  R47 <= *++R57(R27)
  R49 <= *R97++(3)
  R50 <= R47 * R45
  R53 <= R50 * R49
  R61 <= R61 + R53
```

(f)

Fig. 3. An example of AGU optimization: **(a)** SSA code shown in Figure 2(b); **(b)** VTG of T1; **(c)** Concatenated schedule of successive iterations; **(d)** The shaded nodes are associated with the address tree of T1; **(e)** The transformation results in $T1_{i+1} = T1_i + (-1)$; **(f)** Final result of optimization.

The consecutive iterations are concatenated until all the values forming the dependence relation are shown in the concatenated schedule. In most cases, the number of concatenated flat schedules is two as most dependence relations span only two iterations.

An example of the AGU optimization is shown in Figure 3, where Figure 3(a) shows the loop body shown in Figure 2(b) and Figure 3(b) shows the VTG shown in Figure 2(c). The flat schedule of Figure 3(b) is shown in Figure 3(c), where two flat schedules of the *i*th and *(i+1)*th iterations are concatenated. The nodes belonging to the address tree of T1 are shaded in Figure 3(d).

An address tree is modified so that the flow of an address value in the address tree can be implemented by using one auto-modification addressing operation. The relation between address values involved in auto-modification on an AGU can be represented as follows.

$$\mathrm{T_i} = \mathrm{T_{i-1}} + \mathrm{OFFSET} \tag{1}$$

where $\mathrm{T_i}$ and $\mathrm{T_{i-1}}$ are the current and previous address values and OFFSET is either a constant or a register. When OFFSET is a register, the register value is not changed after memory access. Equation (1) implies the address tree must be modified such that the constant or register node used as OFFSET is placed as a direct predecessor of the address node, since the direct predecessor and the address node are added to produce the next address value. The addition can be executed on an AGU by replacing the addition with an auto-modification operation. Since an address tree is built from the corresponding VTG in which all operations are addition, the nodes in the address tree can be reordered without considering the type of operation if the change affects only the address calculation. For example, an intermediate node whose value is used only for address calculation can be placed anywhere in the address tree. In Figure 3(e), $\mathrm{R62}_i$ is such an intermediate node.

To find the appropriate ordering for the auto-modification operation, the order of nodes in the address tree is changed and the successive additions with constants are merged to form a subtree that represents the calculation of the current address. The subtree is placed as a direct predecessor of the address node, because an auto-modification operation can be exploited when the next address is calculated by adding or subtracting a constant offset or a modify register to or from the current address as shown in equation (1). After each change of the order, the result is examined to know whether the result can be implemented by an auto-modification operation without changing the schedule of data computation that is determined in the preceding optimization steps. When a constant offset is tried, its size is checked to determine if it can be represented on the AGU. For example, the offset size is limited to 5 bits in TMS320C4X. On the other hand, when a modify register is attempted, the modify register must be an invariable register such as R51, R27, and R36 in Figure 2, since auto-modification operations modify only the address. If there are multiple candidates for an offset, one that is closest to the address node in the original address tree is selected. This process repeats until an auto-modification addressing operation is found or all the cases are examined. The detailed procedure is summarized in Figure 4.

The proposed AGU optimization is effective when address calculation is separated from data calculation in the source code, because inter-mixed calculations reduce the number of possible node orderings. If the changes of the address node orders in inter-mixed calculations affect the data calculation schedules, they are not included. Most DSP kernels have separate address calculation.

Figure 3(e) shows how the address tree can be implemented by using an auto-decrement addressing mode. By placing the constant node as shown in

```
AGU_Optimization(VTG List)
1.  foreach VTG in VTG List begin
2.    generate a flat schedule
3.    construct an address tree
4.    auto_modification_transform(address tree)
5.  endforeach


auto_modification_transform(address tree)
1.  form a subtree representing the calculation of
    current address value by changing the order of
    nodes and merging successive additions with
    constants
2.  if (a subtree is not found) then
3.    return
4.  endif
5.  place the subtree as a direct predecessor of
    the address node
6.  do begin
7.     find a constant node which can be placed as a
       direct predecessor of the address node representing
       the next address value
8.     if (a constant node is found) then
9.        modify code using the auto-modification by
          using the constant as OFFSET
10.       return
11.    endif
12. while (all the cases with constants are tried)
13. do begin
14.    find a invariable register node which can be placed as a
       direct predecessor of the address node representing
       the next address value
15.    if (a invariable register node is found) then
16.       modify code using the auto-modification by
          using the invariable register as OFFSET
17.       return
18.    endif
19. while (all the cases with invariable registers are tried)
```

Fig. 4. AGU optimization procedure.

Figure 3(e), the next address, $\mathsf{T1}_{i+1}$, can be computed by adding the value of $\mathsf{R62}_i$ in the current iteration and (-1). Furthermore, $\mathsf{R62}_i$ is used as an address value, $\mathsf{T1}_i$, in the current iteration. It can be determined in Figure 3 that an address is calculated by adding R51 and R58 and the address is decremented by one after accessing a memory location. Therefore, an auto-decrement addressing mode can be used.

4. EXPERIMENTAL RESULTS

The proposed method was implemented in the GCC back end targeted for TI's TMS320C40 DSP architecture [Texas Instruments 1999]. We selected a set of

Table I. Benchmark Programs

Benchmark	Description
fir	Finite impulse response (FIR) filter
iir	Infinite impulse response (IIR) filter
conv.	Convolution using pointers
comp.	Notch filter
mat1x3	Matrix multiplication
index	Array accessing with explicit indexing
fir2d	Two-dimensional FIR filter
dot_pdt.	Dot product
lms	Least mean square (LMS) filter
dft	Discrete Fourier Transform (DFT)
n_real.	Array accessing with pointer
sum	Summation of array elements

Table II. Comparison of Static Code Sizes Measured Before and After AGU Optimization. The Average Code Size Reduction is 19%

Benchmark	Before	After	Reduction (%)
fir	15	12	20
iir	27	22	19
conv.	11	7	36
comp.	16	12	25
mat1x3	15	11	27
index	11	6	45
fir2d	45	40	11
dot_pdt.	9	8	11
lms	23	21	9
dft	110	100	9
n_real.	9	9	0
sum	13	11	15

programs, where arrays are heavily used, from the DSPstone benchmark suite [Zivojnovici et al. 1994] and some other applications. The programs used in our experiments are summarized in Table I. We compiled the selected programs and observed the static code size and execution cycle of each program. The code sizes and cycle counts before and after AGU optimization are shown in Table II and III, respectively. As can be seen in Table II, the AGU optimization decreases the code size by 19% on the average. Table III shows the execution cycles of the kernel programs, which are represented as functions of two parameter values, filter length (N) and input size (M). The parameters are specified in *param.* column in Table III. The AGU optimization reduce the execution cycle by 28%.

Table IV shows total CPU time and the percentage of two important phases. On the average, the phase of constructing VTGs consumes 25% compilation time because it needs to scan the whole IR code. However, the overall optimization time is reduced, as the code is optimized with considering small-sized VTGs and address trees instead of large-sized CDFGs. The values in the fourth column include the time consumed to find flat schedules and transform address trees as well as the optimization time.

Table III. Comparison of Execution Cycles Measured Before and After AGU Optimization. N is Filter Length and M is Input Size. The Average Cycle Count Reduction is 28%

Benchmark	Before	After	Reduction (%)	param.
fir	$11 + 4N$	$10 + 2N$	44	$N = 16$
iir	$7 + 20N$	$7 + 15N$	24	$N = 16$
conv.	$6 + 5M$	$5 + 2M$	57	$M = 15$
comp.	$13N$	$12N$	8	$N = 400$
mat1x3	$\approx 4M^2$	$\approx 3M^2$	25	$M = 100$
index	$8 + 3M$	$4 + 2M$	34	$M = 100$
fir2d	$\approx 16N^3$	$\approx 9N^3$	44	$N = 256$
dot_pdt.	$6 + 3M$	$6 + 2M$	33	$M = 256$
lms	$14 + 9N$	$14 + 7N$	20	$N = 16$
dft	$\approx 26N^2$	$\approx 24N^2$	13	$N = 256$
n_real.	$5 + 4M$	$5 + 4M$	0	$M = 16$
sum	$6 + 8M$	$6 + 5M$	37	$M = 100$

Table IV. Total CPU Time and Percentage of the Proposed Optimization Phases

Bench-mark	Total CPU time (sec)	Percentage of VTG construction time (%)	Percentage of AGU optimization time (%)
fir	0.18	34	6
iir	0.28	25	14
conv.	0.17	33	14
comp.	0.16	31	6
mat1x3	0.12	25	8
index	0.11	18	9
fir2d	0.54	37	6
dot_pdt.	0.13	15	8
lms	0.31	33	6
dft	1.42	19	1
n_real.	0.20	20	10
sum	0.19	26	16

5. CONCLUSION

In this paper, we have presented a new approach of DSP code generation to optimize auto-modification addressing modes by tracing the address values of array accesses. Given a source code, the high-level information specialized for address calculation is first collected to consider DSP specific features, then the SSA form is generated to construct new graphs, VTGs, that are developed to make it easy to find patterns of address modification. Based on the graph, the code is optimized for auto-modification addressing modes. We implemented the proposed approach in the back end of GCC and compiled DSP kernels in DSPstone benchmark suite. The results before and after the AGU optimization were compared. The code size and execution cycle were reduced by 19% and 28%, respectively after applying the proposed AGU optimization.

REFERENCES

AHO, A., SETHI, J., AND ULLMAN, J. 1986. *Compilers Principles, Techniques and Tools*. Addison-Wesley, Reading, Mass.

BARTLEY, D. 1992. Optimizing stack frame accessed for processors with restricted addressing modes. *Software-Practice and Experience 22*, 3 (Feb.), 101–110.

BASU, A., LEUPERS, R., AND MARWEDEL, P. 1999. Array index allocation under register constraints in dsp programs. In *Proceedings of the Twelfth International Conference on VLSI Design*. 330–335.

CYTRON, R., FERRANTE, J., ROSEN, B., WEGMAN, M., AND ZADECK, K. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst. 13*, 4 (Oct.), 451–490.

GEBOTYS, C. 1997. DSP address optimization using a minimum cost circulation technique. In *Proceedings of the Twelfth International Conference on Computer-Aided Design*. 100–103.

LEUPERS, R. AND MARWEDEL, P. 1996. Algorithms for address assignment in DSP code generation. In *Proceedings of the IEEE International Conference on Computer-Aided Design*. 109–112.

LIAO, S., DEVADAS, S., KEUTZER, K., TJIANG, S., AND WANG, A. 1996. Storage assignment to decrease code size. *ACM Trans. Program. Lang. Syst. 18*, 3 (May), 186–195.

MCCONNELL, C. AND JOHNSON, R. E. 1992. Using static single assignment form in a code optimizer. *ACM Lett. Program. Lang. Syst. 1*, 2 (June), 152–160.

MIRANDA, M. A., CATTHOR, F. V. M., JASSEN, M., AND MAN, H. J. D. 1998. High-level address optimization and synthesis techinques for data-transfer-intensive applications. *IEEE Trans. VLSI Syst. 6*, 4 (Dec.), 667–686.

OTTONI, G., RIGO, S., ARAUJO, G., RAJAGOPALAN, S., AND MALIK, S. 2001. Optimal live range merge for address register allocation in embedded programs. In *Proceedings of the 10th International Conference on Compiler Construction*. Springer, Genoa, Italy, 274–288.

SUDARSANAM, A., LIAO, S., AND DEVADAS, S. 1997. Analysis and evaluation of address arithmetic capabilities in custom DSP architectures. In *Proceedings of the IEEE Design Automation Conference*. 287–292.

TEXAS INSTRUMENTS. 1999. *TMS320C4x User's Guide*. Texas Instruments.

WOLFE, M. 1996. *High Performance Compilers for Parallel Computing*. Addison-Wesley, Redwodd City, CA.

ZIVOJNOVICI, V., VELARDE, J. M., AND SCHLAGER, C. 1994. Dspstone : A DSP-oriented benchmarking methodology. In *Proceedings of the International Conference on Signal Processing and Technology*.

Received January 2002; revised March 2003; accepted March 2003