IEICE TRANS. FUNDAMENTALS, VOL.E85–A, NO.6 JUNE 2002

LETTER

Loop and Address Code Optimization for Digital Signal Processors

Jong-Yeol LEE[†], ***Regular Member and*** **In-Cheol PARK**[†a)], ***Nonmember***

SUMMARY 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 loops and the address computation of array accesses. The information is used in generating hardware loop instructions and parallel instructions provided by most digital signal processors. In addition to the conventional control-data flow graph, a new graph is employed to make it easy to find auto-modification addressing modes efficiently. Experimental results on benchmark programs show that the proposed method is effective in improving performance.

key words: *digital signal processor (DSP), compiler, code optimization*

1. Introduction

Digital signal processors (DSP's) 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 DSP's, RAM, ROM, and ASIC's, are being incorporated into a single chip. Traditional approaches to achieve high-quality embedded software have based on writing the code in assembly language. As the complexity of embedded system grows, however, programming in assembly language and manual optimization becomes no longer practical, except for the time-critical portions. These trends mandate the use of high-level languages (HLL's) in order to decrease development cost and time-to-market pressure.

While the conventional code optimization methods [1] 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 DSP's remain challenges to compilers.

Manuscript received September 17, 2001.
Manuscript revised December 18, 2001.
Final manuscript received February 28, 2002.

[†]The authors are with the Department of EECS, KAIST. The address is CHiPS building 2nd floor, KAIST, 373-1, Guseong-dong, Yuseong-gu, Daejeon, 305-701, Republic of Korea.
a) E-mail: icpark@ics.kaist.ac.kr

Compilers for embedded DSP's must take advantage of specialized architectural features that most DSP's provide to make efficient codes that meet the constraints of real-time performance. In addition, DSP's usually provide separate address generation units (AGU's) 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 constant values. This separate address arithmetic improves both code size and performance by allowing parallel address computations that are otherwise to be performed serially on the datapaths. A common feature of digital signal processing algorithms is that a small number of instructions called a *kernel* is repeatedly executed. For this, DSP's include hardware loop instructions to handle this sort of repeated executions. A hardware loop instruction allows a single instruction or a block of instructions to be repeated a number of times without the overhead that would normally come from the decrement-test-branch sequence at the end of a loop. The hardware loop loses no time in incrementing or decrementing a counter, checking to see if the loop is finished, and branching back to the top of the loop. In most DSP's, parallel instructions are provided to exploit the parallelism in architectures. For example, TMS320C4x [13] provides an instruction group that permits parallel-operations to make a high degree of parallelism.

In this paper, we focus on the code generation for DSP's that have such specific architectural features, and present a new method to optimize the address calculation of array accesses by utilizing the auto-increment/auto-decrement capabilities of AGU's. In the proposed method, hardware loop instructions and parallel instructions are also considered to reduce execution time.

The rest of this paper is organized as follows. In Sect. 2, we describe a short review of related works. An overview of the proposed method is presented in Sect. 3. Details are described in Sects. 4, 5, and 6. Experimental results are shown in Sect. 7 and finally conclusion remarks are made in Sect. 8.

2. Previous Works

Several authors have tried to optimize the code using high-level transformation. In [7], a C source-to-source transformation is described, which transforms an array reference to a pointer reference and modifies the resulting pointer for the next reference of the array. Basu et al. [8] presented a heuristic code optimization technique which, given an AGU with a fixed number of address registers, minimizes the number of instructions needed for address computations in loops by converting array accesses to pointer accesses. Su et al. [9] applied a source-level loop optimization technique which transforms a source code using a software-pipelining technique. The previous works show that the high-level transformation is very effective in enhancing performance. However, no previous works have considered the high-level transformation of loops that can be optimized by using hardware loop instructions.

Other previous works for DSP code generation focused on the problem of storage assignment. The storage assignment problem is to determine the layout of scalar variables, which are declared local, in a stack frame so that the code needed for address calculation is minimized by applying auto-increment/auto-decrement modes. Bartley [3] firstly addressed a simple version of the offset assignment problem. The storage assignment problem with multiple address registers has been addressed by Liao et al. [4] and Leupers and Marwedel [5], but they considered only unit increment. Sudarsanam et al. [6] generalized the offset assignment problem by allowing multiple address registers and auto-increment/auto-decrement features that varies from $-l$ to $+l$. Although the previous works on storage assignment problem can be applied to determine the relative location of arrays, they cannot be used for determining the layout of the elements in an array. In the works, the elements are assumed to be linearly arranged according to their indices.

In this work, the high-level transformation is employed to obtain the high-level information related to loops and the address computation of array accesses. The information is used in optimizing code with hardware loop instructions and auto-modification addressing modes. The high-level transformation also performs function inlining to achieve a trade-off between code size and execution cycle. Assuming the relative location of arrays is determined, we in this paper optimize the accesses to elements within one array by using auto-modification addressing modes.

3. Overview of the Proposed Approach

The proposed method assumes that hardware loop instructions support single- and multi-instruction hardware loops with constant bounds and steps and single-instruction hardware loops can be nested within a multi-instruction loop. The AGU is assumed to have post/pre-increment/decrement functionality with a constant offset or a modify register.

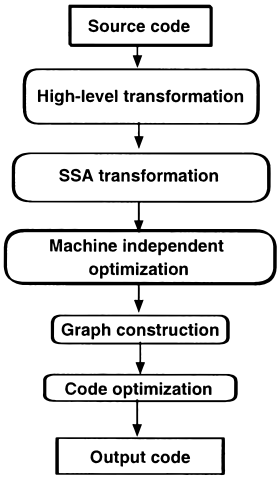

Fig. 1 Overall flow of the proposed method.

As depicted in Fig. 1, the proposed approach starts from the high-level transformation implemented in the front end of a compiler. The special features of a DSP are considered in order to transform a source code in a way that the machine-dependent optimizations become more effective on the transformed code.

In the high-level transformation, the loops that can be implemented by hardware loop instructions are found by examining the pattern of loops. The loops found are transformed so that hardware loop instructions can be mapped to the loops in the code optimization step. Function inlining is also performed to achieve a trade-off between code size and execution cycle. The address values of array accesses are identified in the high-level transformation and the corresponding variables and operations in the intermediate representation (IR) are tagged for references in the graph construction step that builds graphs to be used in the code optimization.

After the high-level transformation, the code is converted into the static single assignment (SSA) form [10], [11] that is an efficient way of representing the data flow of a routine. A code is said to be in SSA form if the definition of a memory location or a register appears statically only once in the routine. The SSA representation is invaluable for eliminating false dependencies and finding the flow of an address value. Once a code is transformed into SSA form, we can build both *use-def* and *def-use* chains [1] that are extensively used in many classical optimizations, including dead-code elimination, constant propagation, and global value renumbering. Machine-independent optimizations such as constant propagation, common subexpression elimination, and jump optimization [1] are performed after the

SSA transformation.

A value-trace graph (VTG) is constructed after machine-independent optimizations. The VTG is used as a new data-flow representation in addition to the conventional control-data flow graph (CDFG) where non-address computations are intermingled with address computations. In the proposed method, the 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 code accessing elements of arrays with auto-modification addressing modes by considering only the nodes in the VTG instead of considering all the operations and operands in a CDFG. The code optimization utilizing AGU's 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. Then, parallel instructions are considered to pack individual instructions together. In the following sections, each step in Fig. 1 is described in more detail.

4. High-Level Transformation

The high-level transformation deals with loop transformation and function inlining. First, loops are transformed into canonical forms whenever possible so that the hardware loop instructions can be exploited. As shown in Fig. 2, by applying the loop transformation at high-level, more loops can be optimized since the loop transformation prevents the front end of a compiler from transforming a loop into a general form to which hardware loop instructions cannot be mapped directly. Hardware loop instructions can be used to generate effective codes for the loops whose *bound* and *step* are constant at compile time. At the source level, the loop transformation finds natural loops [1] and converts the loops into canonical forms by applying various loop transformations presented in [2] such as loop interchanging. For example, when a loop with a non-constant loop bound is nested in another loop with a constant loop bound, hardware loop instructions cannot be utilized if the hardware loop instructions support only flattened loops. However, when the loops are interchanged, the inner loop can be implemented with the hardware loop instructions.

The second high-level transformation is to achieve a trade-off between code size and execution cycle using function inlining. Generally function inlining increases the object code size and reduces the execution cycle since function calls are slower but take less space than the inlined code. In addition, the function inlining may enlarge basic blocks by removing function calls that make basic blocks end. The chances of optimization can be increased with the enlarged basic blocks since a basic block is a unit to which most optimizations are applied. In this transformation, all the functions whose sizes are less than a threshold value are inlined. The size of a function is estimated by counting the number of operations in the IR.

Finally, the information on the addresses of array accesses are 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 array accesses are identified, the IR operations and their operands which access the array elements are generated with the information on array accesses. Since the addresses of array accesses are 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 array access addresses is tagged as an array address calculation and the operands of the array address calculation are tagged as either the start address of an array or the index of an array element. This information is used

```
loop_transform_at_high_level(source code)
1.  find natural loops
2.  foreach loop L in source code begin
3.      find the constant bound and step of L
        by calling make_bound_step_constant(L)
4.      if (the constant bound and step are found) then
5.          transform L into a canonical form
6.          register L in the list of canonical loops
7.      endif
8.  endforeach

make_bound_step_constant (L)
1.  if (the bound and step of L are constant) then
2.      return TRUE
3.  else
4.      transform L by using loop transformations in [2] to make
        the bound and step of L constant
5.      if (the bound and step of L are constant) then
6.          return TRUE
7.      endif
8.  endif
9.  return FALSE

map_loop_instructions(list of canonical loops)
1.  foreach loop L in the list of canonical loops begin
2.      apply pattern matching to L to find the bound and step
3.      insert instructions that set loop repeat count and step
4.      insert a hardware loop instruction
5.      remove unencessary instructions
6.  endforeach
```

Fig. 2 Hardware loop instruction mapping procedure.

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.

5. Graph Construction

To optimize the code by using auto-modification addressing modes supported by an AGU, a VTG that captures the characteristics of the AGU is constructed for each address node that is identified in the high-level transformation. To construct the VTG, the data flow of the CDFG 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 by adding or subtracting the value in the modify registers or constant values on an AGU. Minus operations are converted into plus operations by changing the sign of operands. In addition, when an address is changed in a linear fashion by multiplying and adding constants to an address value, the multiplication can be converted to an addition by inserting appropriate pre-computations. An example of the conversion is shown in Fig. 3(b), where R90 * 3 and R90 + 1 are converted to (3 * R90) + 3.

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 a 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 finding nodes as above, 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 a VTG are assignments and additions and hence a node in the VTG has one or two incoming edges. The incoming edges of a node are ordered according to the kind of the source node of each edge to reduce the code optimization time by reducing the search space. Since a VTG contains only the operands involved in the computation of an address value and the implied operation in the VTG is addition, the AGU optimization can be performed by manipulating the VTG.

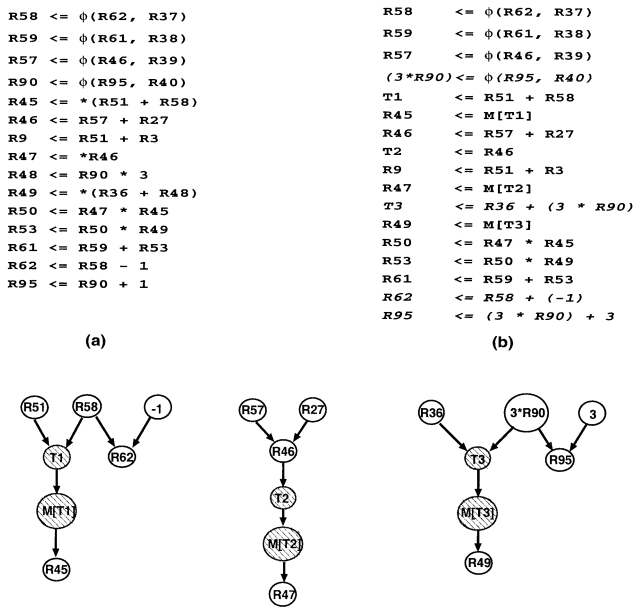

Fig. 3 Example of the VTG. **(a)** SSA form representation of the CDFG of a loop body. **(b)** Code after converting operations in (a). The statements shown in italic contain the converted operations. The address nodes and array access nodes are inserted based on the information from the high-level transformation. The address nodes are T1, T2, and T3 and the array access nodes are M[T1], M[T2], and M[T3]. **(c)** VTG's of the address nodes in (b). The shaded nodes are inserted address nodes and array access nodes. The implied operations are additions.

Three VTG's corresponding to Fig. 3(b) are shown in Fig. 3(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 VTG's have three invariable nodes, R51, R27, and R36 whose values are not changed in the given block and the nodes, R62, R46 and R95 are dead at the end of the block but the values of these nodes should be retained as they are used in the next iteration. The VTG of T2 shows that the operands contributing to the computation of T2 is R57, R27, and R46 and R46 is computed by adding R57 and R27 since an addition is implied. Furthermore, the data-flow analysis shows that R46 is used in the next iteration as a new value of R57.

6. Code Optimization

In the proposed method, three code optimizations are applied; loop optimization, AGU optimization, and parallel optimization. In the loop optimization, hardware loop instructions are mapped to loops using the result of the high-level transformation. The procedure *map_loop_instructions* in Fig. 2 shows how hardware loop instructions are mapped to the loops which are transformed into canonical forms in the high-level

transformation. The bound and step of a loop are found by applying *pattern matching* to the IR. In this case, the pattern matching is very efficient since the pattern matching routine considers only the small number of pre-determined patterns generated as a result of the high-level transformation. After finding the loop bound and step, a hardware loop instruction and instructions that set a *loop repeat count* are inserted into the pre-header of the loop. The instructions that calculate the value of a loop index variable and check the branch conditions are eliminated and then code modifications are performed if necessary.

In the proposed AGU optimization, the accesses to arrays are optimized by exploiting auto-modification addressing modes in the information on the array accesses collected in the high-level transformation 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 constants or registers to the address register. In the optimized code with auto-modification addressing modes, however, the calculation with the address register 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, where the flow of the address value represented as an address node is found using the VTG in the first step 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. By concatenating the flat schedules of successive iterations, a tree that shows how the next address value of an address node is calculated using the values in the current iteration is formed, which is called an *address tree* of the address node. The address tree is constructed by selecting the predecessors of the address node in the concatenated schedule and the edges between the selected nodes. The number of the flat schedules to be concatenated to form an address tree is determined by loop-carried dependence relations since the flat schedules of the consecutive iterations are concatenated until two values forming a dependence relation are shown in the concatenated schedule. In most cases, however, the number of the flat schedules to be concatenated is two as most dependence relations span only two iterations.

An example of the AGU optimization is shown in Fig. 4, where Fig. 4(a) shows the loop body shown in Fig. 3(b) and Fig. 4(b) shows a VTG shown in Fig. 3. The flat schedule of Fig. 4(b) is shown in Fig. 4(c), where two flat schedules from ith and $(i+1)$th iterations are concatenated. The nodes of the address tree

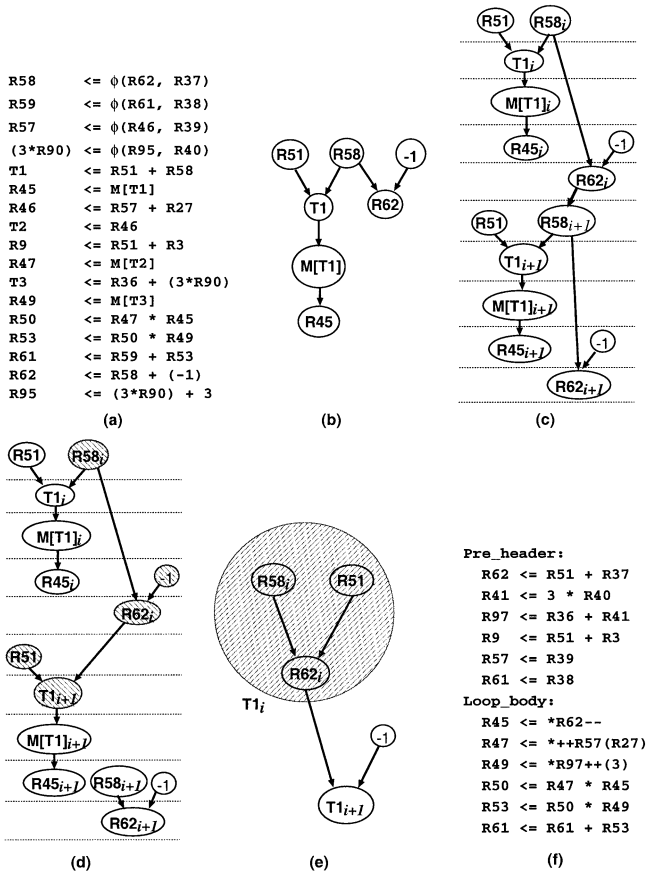

Fig. 4 Example of the AGU optimization. **(a)** SSA form code of a loop body in Fig. 3(b). **(b)** VTG of T1 in Fig. 3(c). **(c)** Flat schedules of successive iterations. **(d)** The shaded nodes are the nodes in the address tree of T1. **(e)** The transformation result of the address tree shows that $\mathsf{T1}_{i+1} = \mathsf{T1}_i + (-1)$. **(f)** Final result of optimization.

of T1 are shown as shaded nodes in Fig. 4(d).

An address tree is modified so that the flow of an address value in the address tree can be implemented by one auto-modification addressing operation. For the flow in an address tree to be implemented by an auto-modification addressing operation, constant or invariable register nodes must be placed as close as possible to the address node by changing the order of the nodes in the tree. This is because the direct predecessors of an address node are added to produce the address values and the addition can be executed on an AGU by replacing the addition with an auto-modification operation. Since a VTG is constructed for each address node and an address tree is a part of a VTG, additions are implied between the nodes. Therefore, the order of the nodes in an address tree can be changed without considering the operation when the change affects only the address calculation. For example, an intermediate node whose value is used not in data calculation but only in address calculation can be placed anywhere in an address tree. $\mathsf{R62}_i$ in Fig. 4(e) is such an intermediate node. The total number of changes is reduced

because of the ordered edges in the VTG and restrictions on node changing.

Since an address tree represents an address calculation, an auto-modification operation is applied to the address tree. To find the appropriate ordering of nodes for the auto-modification operation, the order of nodes in the address tree is changed and the successive additions with constants are merged so that some nodes form a subtree that represents the calculation of the current address and the subtree is placed as close as possible to the address node that represents the next address because an auto-modification operation can be exploited when the next address is calculated by adding or subtracting a constant offset or a register used as a modify register to or from the current address. After each change of the order, the result is examined to know whether the result can be implemented by an auto-modification operation with a constant offset or a modify register. When a constant offset is tried, the size of the offset is checked if it can be represented on the AGU. For example, the size of the offset is 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 Fig. 3 since auto-modification operations modifies only the address and the modify register must remain unchanged. When there are multiple candidates for an offset or a modify register, the one that is closest to the address node in the original address tree is selected. This process stops when an auto-modification addressing operation is found or all the cases are searched.

The proposed AGU optimization is more effective with the source code where address calculation is separated with data calculation since inter-mixed data and address calculations reduce the number of possible node orderings. However, in most DSP kernels, the data and address calculations are separated and the proposed AGU optimization proves to be efficient.

Figure 4(e) shows how the address tree can be implemented by an auto-decrement addressing mode. By placing the constant node as shown in Fig. 4(e), the next address after the memory access, $\mathsf{T1}_{i+1}$, can be calculated 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. Therefore, from Fig. 4, it can be determined that an address is calculated by adding R51 and R58 and after accessing a memory location, the address is decremented by one and hence an auto-decrement addressing mode can be used.

In the proposed method, the parallel optimization, which packs more-than-one individual instructions into a parallel instruction, is performed after the AGU optimization, since in DSP's the addressing mode of the operands of some parallel instructions are restricted to auto-modification addressing modes. In the parallel optimization, the instructions in the CDFG are scheduled in two steps. First, a *modulo scheduling* technique, which is a kind of *software-pipelining* technique [15] is applied to loops. Then, an *as-soon-as-possible* (ASAP) scheduling technique is applied to basic blocks. The modulo scheduling technique is to schedule the instructions of a number of different iterations together. In the modulo scheduling, the *list scheduling* technique is used to select an instruction to be scheduled next in the ready instructions by giving priority to the instructions with considering the restrictions on parallel instructions. Since in most DSP's, only the restricted instructions can be packed as parallel instructions, the instructions that can be packed with the previously scheduled instructions are given high priority. When there are more-than-one instructions that can be packed with the previously scheduled instructions, the instructions are prioritized by counting the number of legal locations where the instructions can be placed so that the instructions which are more difficult to place are scheduled first. The instructions with the same number of legal locations are selected randomly. After finding parallel instructions from different iterations of a loop, an ASAP scheduling is applied to basic blocks to reveal the independent instructions that can be scheduled in the same time slot, and then scans the independent instructions to find adequate instructions for packing.

7. Experimental Results

The proposed method was implemented in the GCC back end targeted for TI's TMS320C40 DSP architecture [13]. We selected a set of programs, where arrays are heavily used, from the DSPstone benchmark suite [12] and some other applications. We compiled the selected programs and observed the static code size and execution cycle of the kernel of each program. The results of our compiler are compared to those of TI's C4x compiler (*cl30*) and GCC ported for TMS320C40 (*c4x-gcc*). GCC is selected since it is equal or sometimes superior to many other commercial compilers in terms of performance. To show the effect of each optimization, the proposed optimizations are applied in an incremental way. First, only the hardware loop optimization is applied and then the hardware loop optimization and AGU optimization are applied. Finally, all the optimizations including the parallel optimization are applied. To evaluate the performance of the proposed compiler, the code size and execution cycle are normalized by the code size and execution cycle of the codes generated from cl30. The results are summarized in Tables 1 and 2.

Table 1 shows the ratio of static code size. The AGU optimization decreases the average code size by 24% compared to the code resulting from the application of the hardware loop optimization. The code size is further decreased by applying the parallel optimization to the results of the hardware loop and AGU

Table 1 Comparison of static code size.

Benchmark	cl30		c4x-gcc		proposed (no opt.)		proposed (H/W loop)		proposed (H/W loop +AGU)		**proposed (H/W loop +AGU +Parallel)**	
	size	ratio(%)	size	ratio(%)	size	ratio(%)	size	ratio(%)	size	ratio(%)	size	ratio(%)
fir	12	100%	13	108%	13	108%	15	125%	12	100%	11	**92%**
iir	16	100%	17	106%	28	175%	27	169%	22	138%	17	**106%**
convolution	7	100%	5	71%	13	186%	11	157%	7	100%	7	**100%**
compaction	16	100%	12	75%	15	94%	16	100%	12	75%	11	**69%**
mat1x3	12	100%	12	100%	15	125%	15	125%	11	92%	11	**92%**
index	8	100%	5	63%	11	138%	11	138%	6	75%	7	**88%**
fir2dim	38	100%	41	108%	49	129%	45	118%	40	105%	40	**105%**
dot_product	8	100%	9	113%	8	100%	9	113%	8	100%	8	**100%**
lms	27	100%	21	78%	27	100%	23	85%	21	78%	20	**74%**
n_realupdate	8	100%	9	113%	7	88%	9	113%	9	113%	8	**100%**
dft	83	100%	110	133%	117	141%	110	133%	100	120%	96	**116%**
sum	11	100%	13	118%	14	127%	13	118%	11	100%	11	**100%**
(ave.)		100%		97%		126%		124%		100%		**95%**

Table 2 Comparison of the execution cycles of DSP kernels.

Bench-mark	cl30		c4x-gcc		proposed (no opt.)		proposed (H/W loop)		proposed (H/W loop + AGU)		**proposed (H/W loop + AGU + Parallel)**		param.
	exec. cycle	(%)	exec. cycle	(%)	exec. cycle	(%)	exec. cycle	(%)	exec. cycle	(%)	exec. cycle	(%)	
fir	$10+2N$	100	$11+2N$	102	$7+9N$	360	$11+4N$	179	$10+2N$	100	$10+N$	**62**	$N=16$
iir	$3+13N$	100	$5+12N$	93	$5+23N$	177	$7+20N$	155	$7+15N$	117	$5+12N$	**93**	$N=16$
conv.	$5+2N$	100	$3+2N$	95	$6+10N$	422	$6+5N$	219	$5+2N$	95	$6+N$	**59**	$N=15$
comp.	$16N$	100	$12N$	75	$15N$	94	$13N$	100	$12N$	75	$11N$	**69**	$N=400$
mat1x3	$\approx 4N^2$	100	$\approx 4N^2$	100	$\approx 8N^2$	197	$\approx 4N^2$	99	$\approx 3N^2$	75	$\approx 3N^2$	**75**	$N=100$
index	$5+2N$	100	$3+2N$	99	$6+8N$	393	$8+3N$	150	$4+2N$	100	$7+N$	**52**	$N=100$
fir2d	$\approx 12N^3$	100	$\approx 12N^3$	100	$\approx 26N^3$	300	$\approx 16N^3$	133	$\approx 9N^3$	75	$\approx 9N^3$	**75**	$N=256$
dot_pdt.	$5+3N$	100	$6+3N$	100	$3+8N$	265	$6+3N$	100	$6+2N$	67	$7+N$	**34**	$N=256$
lms	$18+9N$	100	$14+7N$	78	$10+18N$	184	$14+9N$	98	$14+7N$	78	$14+6N$	**68**	$N=16$
n_real.	$5+3N$	100	$3+6N$	102	$1+9N$	274	$5+4N$	130	$5+4N$	130	$5+3N$	**100**	$N=16$
dft	$\approx 28N^2$	100	$\approx 21N^2$	75	$\approx 30N^2$	107	$\approx 26N^2$	93	$\approx 24N^2$	86	$\approx 20N^2$	**71**	$N=256$
sum	$5+6N$	100	$6+7N$	117	$5+12$	199	$6+8N$	133	$6+5N$	84	$6+5N$	**84**	$N=100$
ave.		100%		95%		248%		132%		90%		**70%**	

optimizations. However, for some programs, the parallel optimization increases the code size. This results from the software pipelining that needs additional instructions in the prologue and epilogue of a loop. It is not always possible to schedule these additional instructions in the delay slots of branch or hardware loop instructions. On the average, the code size generated from our compiler and c4x-gcc are about 95% and 97% of the codes generated from cl30.

Table 2 shows the execution cycles of the kernel programs as functions of parameter values. The parameters are the length of a filter or the size of inputs. In Table 2, we can see that all the proposed optimizations consistently reduce the execution cycle. The AGU optimization and parallel optimization reduce the execution cycle by 42% and 20%, respectively. Table 2 also shows that in most cases the size of a loop in the code generated from the proposed compiler is smaller than that generated from the other compilers. For example, in case of "*comp.*," the size of the loop in cl30 is 16 and is decreased to 12 in c4x-gcc and to 11 in the proposed compiler. On the average, the code generated from our compiler reduces the execution cycle to 70% and 72% of those of the codes generated from cl30 and c4x-gcc, respectively.

The reduction comes from the use of hardware loop instructions, auto-modification addressing modes, and parallel instructions. After all the proposed methods are applied, the average execution cycle is reduced to 36% of that of the codes with no optimization. The hardware loop instructions allow a block of code to be repeated without any penalty for looping such as the cycle overhead of branch instructions. The average execution cycles are reduced to 60% of the unoptimized case when only hardware loop instructions are employed. With the use of auto-modification addressing modes, we can eliminate the instructions that modify address values and increase the possibility of

packing instructions since the operands of some parallel instructions must have auto-modification addressing modes. The additional reduction in the average execution cycles with the AGU optimization is 17%. The parallel instructions make it possible to pack instructions, which are from different iterations of a loop body, in one instruction word when the software pipelining technique is applied. By using the parallel instructions after applying the AGU optimization, an 8% additional reduction in the average execution cycle is achieved.

8. Conclusion

In this paper, we have presented a new DSP code optimization approach that optimizes auto-modification addressing modes by tracing the address values of array accesses. Given a source code, the high-level transformations specialized for hardware loops, functional inlining, and address calculation are first applied to consider DSP specific features, and then the SSA form is generated to construct new graphs, VTG's, that make it easy to find patterns of address modification. Based on the graphs, three code optimizations are used to consider hardware loop instructions, AGU instructions, and parallel instructions. We implemented the proposed approach in the back end of GCC and compiled DSP kernels in DSPstone benchmark suite to compare the results with TI's TMS320C4x compiler and GCC ported for TMS320C4x. The code size and execution cycle are reduced by 5% and 30% with respect to TI's compiler and by 2% and 26% with respect to GCC ported for TMS320C4x.

Acknowledgement

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.

References

[1] A. Aho, J. Sethi, and J. Ullman, Compilers Principles, Techniques and Tools, Addison-Wesley, 1986.
[2] M. Wolfe, High Performance Compilers for Parallel Computing, Addison-Wesley, 1996.
[3] D.H. Bartley, "Optimizing stack frame accessed for processors with restricted addressing modes," Software-Practice and Experience, vol.22, no.3, pp.101–110, Feb. 1992.
[4] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, and A. Wang, "Storage assignment to decrease code size," ACM Trans. Program. Lang. and Sys., vol.18, no.3, pp.186–195, May 1996.
[5] R. Leupers and P. Marwedel, "Algorithms for address assignment in DSP code generation," Proc. IEEE ICCAD, pp.109–112, 1996.
[6] A. Sudarsanam, S. Liao, and S. Devadas, "Analysis and evaluation of address arithmetic capabilities in custom DSP architectures," Proc. ACM/IEEE Design Automation Conference, pp.287–292, 1997.
[7] C. Liem, P. Paulin, and A. Jerraya, "Address calculation for retargetable compilation and exploration of instruction-set architectures," Proc. ACM/IEEE Design Automation Conference, pp.597–600, 1996.
[8] A. Basu, R. Leupers, and P. Marwedel, "Array index allocation under register constraints in DSP programs," Proc. VLSI Design, pp.330–335, 1999.
[9] B. Su, J. Wang, and A. Esguerra, "Source-level loop optimization for DSP code generation," Proc. IEEE Acoust., Speech & Signal Process., pp.2155–2158, 1999.
[10] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," ACM Trans. Program. Lang. Sys., vol.13, no.4, pp.451–490, Oct. 1991.
[11] C. McConnell and R.E. Johnson, "Using static single assignment form in a code optimizer," ACM Letters on Programming Languages and Systems, vol.1, no.2, pp.152–160, June 1992.
[12] V. Zivojnovici, J. Martinez Velarde, and C. Schlager, "DSPstone: A DSP-oriented benchmarking methodology," Proc. Int. Conf. on Signal Processing and Technology, Dallas, Oct. 1994.
[13] Texas Instruments. TMS320C4x User's Guide, May 1999.
[14] R. Stallman, Using and Porting GNU CC, Free Software Foundation, Cambridge, MA, 1993.
[15] V. Allan, R. Jones, R. Lee, and S. Allan, "Software pipelining," ACM Computing Surveys, vol.27, no.3, pp.367–432, Sept. 1995.