Loop and Address Code Optimization for Digital Signal Processors

Jong-Yeol Lee and In-Cheol Park
Dept. of EE, KAIST, 373-1, Guseong-dong, Yuseong-gu, Daejeon, 305-701, Republic of Korea

Abstract

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.

1. Introduction

As the complexity of embedded system grows, programming in assembly language and optimization by hand 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.

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 separated 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 separated address arithmetic enables improvement of both code size and performance by parallelizing address computation 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 [8] 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. We 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 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 conclusion remarks are made in section 5.

2. Previous works

Several authors have tried to optimize the code using high-level transformation [4][5]. The previous works show that the high-level transformation is very effective in enhancing performance. However, no previous works employed high-level transformation considering 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 [2][3]. Although the 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 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. Proposed Approach

The proposed approach starts from the high-level transformation implemented in the front end of a compiler. 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 [6] that is an efficient way of representing the data flow of a routine.

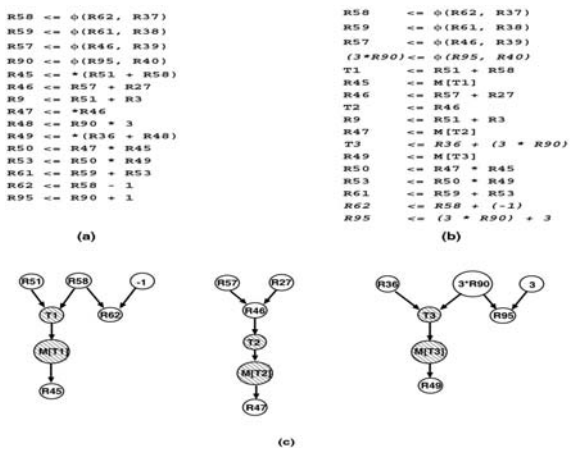

Fig. 1. Example of the VTG. **(a)** SSA form representation of the CDFG of a loop body. **(b)** Code after converting operations in (a). The address nodes and array access nodes are inserted based on the information from the high-level transformation. **(c)** VTG's of the address nodes in (b).

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 using the VTG and then transforming the flow for auto-modification addressing modes to be used. Then, parallel instructions are considered to pack individual instructions together. Fig. 1 and Fig. 2 show an example of a VTG and an example of AGU optimization, respectively.

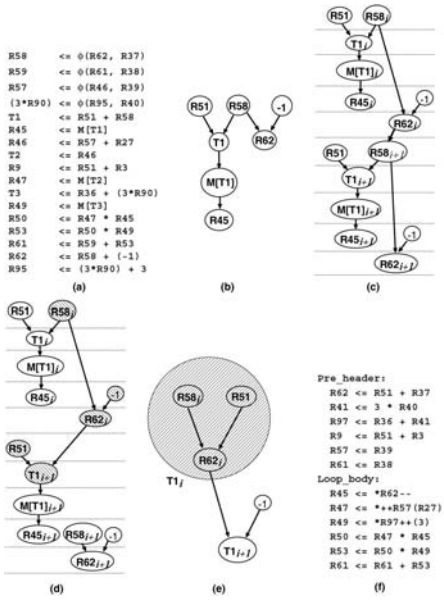

Fig. 2. Example of the AGU optimization. **(a)** SSA form code of a loop body in Fig. 1(b). **(b)** VTG of T1 in Fig. 1(c). **(c)** Flat schedules of successive iterations. **(d)** The shaded nodes are the nodes in the address tree of T1. **(e)** The transformation result of the address tree shows that $T1_{i+1} = T1_i + (-1)$. **(f)** Final result of optimization.

4. Experimental Results

The proposed method was implemented in the GCC back end targeted for TI's TMS320C40 DSP architecture. We selected a set of programs, where arrays are heavily used, from the DSPstone benchmark suite [7] 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*).

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 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 optimizations. 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 1. 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%**	

Table 1 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. The ratio represents the execution cycle of each compiler normalized by that of cl30 for typical parameters. In Table 1, 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. 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.

5. 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.

6. References

1. A. Aho, J. Sethi, and J. Ullman, ``Compilers Principles, Techniques and Tools," Addison-Wesley, 1986.
2. 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.
3. R. Leupers and P. Marwedel, ``Algorithms for Address Assignment in DSP Code Generation," Proc. IEEE ICCAD, pp. 109-112, 1996.
4. 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.
5. A. Basu, R. Leupers, and P. Marwedel, ``Array Index Allocation under Register Constraints in DSP Programs," Proc. VLSI Design, pp. 330-335, 1999.
6. 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, 1991.
7. 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.
8. Texas Instruments. TMS320C4x User's Guide, May 1999.