LOOP OPTIMIZATION WITH TRADEOFF BETWEEN CYCLE COUNT AND CODE SIZE FOR DSP APPLICATIONS

Bogong Su [1] sub@wpunj.edu
Jian Wang [2] jiwang@nortelnetworks.com
Rafi Rabipour[2] rabipour@nortelnetworks.com
Erh-Wen Hu [1] hue@wpunj.edu
Joseph Manzano [1] josbry21@cs.com

ABSTRACT

Software pipelining is an effective technique to reduce cycle count by exploiting instruction level parallelism in loops. It has been implemented in most VLIW DSP compilers. However, software pipelining expands the code size due to the introduction of prelude and postlude. To address this problem, many VLIW DSP compilers include certain code size reduction features. During compilation, a user is given limited options of exercising these code reduction features. As a result, the tradeoff options between cycle count and code size are also limited. Yet today's software development often requires an optimum balance between code size and cycle count, which in turn requires a much wider tradeoff space. This paper presents a new heuristic code-size-constraint loop optimization approach to extend the tradeoff space. Preliminary experimental results indicate that the new approach can significantly widen the tradeoff space, thus providing DSP users with more flexibility to meet their various design criteria.

1. INTRODUCTION

The evolution path of DSP devices over the last two decades has shaped the approaches, strategies and the performance criteria needed to guide the development of DSP software. Whereas cycle count was the predominant software performance metric for early DSP devices, memory utilization has become a significant factor in later designs. The DSP devices of the current generation deliver a quantum leap in processing power over the previous generation to serve high application densities on a single device. Advances in Silicon technology account for part of this increased capability, leading to faster clock rates and sophisticated core architectures that support complex operations and powerful instructions. However, the most significant progress in DSP design which has made systems-on-chip feasible can be attributed to the architecture trend, adopted almost universally, towards devices with multiple processing cores. Multi-core devices take advantage of the fact that many high complexity applications require a finite number of signal processing operation types performed on many independent data streams simultaneously. Exemplifying such applications, telecommunication systems serve a large number of communication channels with similar types of processing. The commonality of the signal processing functions allows multi-core architectures to achieve economy by sharing a large program memory space among all cores. This, in turn, has necessitated cache structures to allow each core to operate independently and at maximum speed.

The combination of fast clock rates, finite data space, and shared program memory have shifted the DSP software development performance criteria away from simple cycle efficiency, toward a search for optimum balance between cycle count, program and data memory footprints. In this quest the critical reliance on cache has further complicated the relationship between cycle count and program memory size. Furthermore, Systems-on-Chip require the delivery of a range of services on the same DSP device or core, such services varying widely in their resource utilization profiles. For example, data services typically require large memory spaces but are not compute-intensive, whereas voice processing may have moderate memory footprints but require complex computations. These considerations provide a strong motivation to seek formal methods for traversing the memory size/cycle count tradeoff space; the objective is to develop the set of techniques necessary to converge to the optimal resource utilization balance for each application. Formal methods facilitate the automation of the code generation process, with the desired tradeoff fed as an input parameter. This in itself is an important factor given the increased reliance on high-level-language compilers as opposed to the assembly-language programming.

In this paper, we will present a new code-size-constraint loop optimization approach to treat the cycle count and code size tradeoff problem for DSP applications. Our approach, based on the well-known software pipelining technique, introduces the code size constraint into the loop schedule problem. The next section will discuss the impact of software pipelining on code size and summarize the previously published results. In section 3, we will formally present the code-size-constraint loop optimization problem.

[1] Dept. of Computer Science, The William Paterson University of New Jersey, Wayne, NJ 07470, USA.
[2] Wireless Speech and Data Processing, Nortel Networks, 2351 Blvd. Alfred-Nobel, St-Laurent, QC, Canada, H4S 2A9.

The new code-size-constraint software pipelining approach will be presented in section 4. Section 5 contains the preliminary experimental results, and we conclude this paper in section 6.

2. THE IMPACT OF SOFTWARE PIPELINING ON CODE SIZE

Software pipelining is an effective technique to exploit instruction-level parallelism in loops [1, 4, 10]. It can significantly reduce runtime and is used extensively in VLIW DSP [2, 3, 6, 8, 11]. However, software pipelining expands code size due to the introduction of prelude and postlude. The size of prelude and postlude grows in proportion to the number of overlapped iterations, which can be large in VLIW DSP processors with many function units.

Recently researchers [2, 6, 11] have tackled the code size reduction problem for software pipelined loops which do not rely on special-purpose hardware. [2] proposes a prelude and postlude collapsing technique for Texas Instruments' TMS320C62 DSP and reports an average of 30% loop code size reduction. [11] uses a code size reduction technique based on a re-timing concept to collapse prelude and postlude, and achieves similar results as [2]. [6] combines scheduling heuristics, postlude collapsing schemas and speculative modulo scheduling, and again realizes a code size reduction of 30% on average with larger benchmark programs.

Some DSP manufacturers such as Texas Instruments have started addressing the code size expansion problem due to software pipelining. Their DSP compilers have incorporated certain code size optimization techniques to allow users to balance the tradeoff between speed and code size. For example, TI's TMS320C62 compiler has ms0, ms1, ms2 and ms3 options [3,9], where ms0 and ms1 use prelude and postlude collapsing technique, and ms2 and ms3 basically turn off software pipelining.

Figure 1 shows the result of the dot-product code generated by TMS320C62 compiler with various code size reduction options. One finds in the figure that, using the collapsing technique, ms0 and ms1 options can reduce the code size by 13%. If code size is the top priority, ms2 or ms3 options must be used to turn off software pipelining, which can reduce the code size by 65%. However, the code size saving comes at the cost of a significant increase in runtime which is nearly 8 times longer. Now the question is: is it possible to provide more choices between these two extremes? By using our new size-constraint pipelining technique, we obtain a wide tradeoff region for the dot-product program. The points Tii=3, Tii=4 and Tii=6 in Figure 1 show our solution where we first apply the code-size-constraint software pipelining approach with different values of initial interval Tii, and then use TMS320C62 assembler to obtain the assembly code. The tradeoff region for code size ranges from 43% to 67% of the original code size, and the corresponding tradeoff region for cycle count ranges from 254% to 138% of its original value.

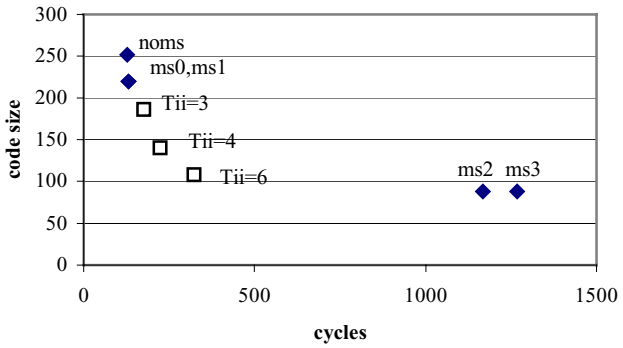

Figure 1 Code size vs. runtime of Dot Product code

3. CODE-SIZE-CONSTRAINT LOOP OPTIMIZATION PROBLEM

In this section, we will formalize the code-size-constraint loop optimization problem. The following definitions will be used throughout this paper.

Definition 3.1 The data dependence between the operations in a loop program can be represented by a doubly weighted data dependence graph, G=(O,E,d,t), which is called the Loop Data Dependence Graph (LDDG), where O is the set of the operations in the loop, E is the set of dependence edges, d is the dependence distance and t is the delay. Both d and t are nonnegative integers and (d, t) is associated with each edge. For example, edge e = (op1,op2) means that op2 can only be issued for execution t(e) cycles after the start of the operation op1 of the d(e)th previous iteration. A data dependence is called a ***loop-independent dependence*** if its dependence distance is 0. A data dependence is called a ***loop-carried dependence*** if its dependence distance is greater than 0.

Definition 3.2 For a given loop and its *LDDG=(O,E,d,t),* a ***loop schedule*** *ls* is a mapping from *OXN* to *N* where *O* is the set of the operations of the loop and *N* is the nonnegative integer set. *ls(op,i)* denotes the cycle number at which the instance of operation op of the ith iteration is issued for execution. *ls* is ***a valid loop schedule*** if and only if the following three conditions are satisfied

1. **hardware constraints**: in each cycle, there is no hardware resource conflict.

2. **data dependence constraints**: for any edge *e = (op1,op2)* and for any *j > 0, ls(op1,j) + d(e) <= ls(op2,j+d(e));*

3. **cyclicity constraints**: *ls* must be expressible in the form of a loop, that is, there is an integer ***II***, for any operation *op* in the loop and for any integer *j > 1, ls(op,j) = ls(op,j-1) + II*(j-1),* where *ls(op,1)* denotes the cycle number at which the instance of *op* of the first iteration is issued for execution and ***II*** is called the ***initiation interval***. The ***cycle count*** of a loop schedule is measured by the initiation interval.

Definition 3.3 For a given loop, the performance of a valid loop schedule is measured by the ***initiation interval*** and the ***code size*** of the scheduled loop. ***Code size*** is defined as the number of bytes which the scheduled loop takes in the program memory. The objective of loop optimization is to find a valid loop schedule with minimum initiation interval and minimum code size.

For loop optimization, reducing initiation interval may increase the code size.

Definition 3.4 For a given loop, ***the minimum initiation interval (min_t)*** and ***the maximum code size (max_m)*** are defined as the initiation interval and the code size, respectively, when we find a valid loop schedule with only one objective of minimizing initiation interval; ***the maximum initiation interval (max_t)*** and ***the minimum code size (min_m)*** are defined as the initiation interval and the code size, respectively, when we find a valid loop schedule with only one objective of minimizing code size.

The objective of existing loop optimization approaches is to find a valid loop schedule with minimum initiation interval only.

The code-size-constraint loop optimization problem is defined as follows.

Definition 3.5 Given a loop and the code size, ***budget_m***, the code-size-constraint loop optimization is to find a **valid loop schedule** with minimum initiation interval such that the code size of the scheduled loop is less than or equal to ***budget_m***.

4. CODE-SIZE-CONSTRAINT SOFTWARE PIPELINING APPROACH

Our code-size-constraint software pipelining approach consists of the following heuristic algorithms.

- The algorithm that finds the minimum code size, *min_m*;
- The algorithm that finds the minimum initiation interval, *min_t*;
- The algorithm that finds a valid loop schedule under the code size constraint;
- The algorithm that finds the relationship between the cycle count and the code size.

Definition 4.1 ***Software pipelining*** is to find a valid loop schedule with minimum initiation interval. A software-pipelined loop consists of three parts: the prelude, the pipelined loop body, and the postlude. Given a loop and its *LDDG = (O,E,d,t)* and assuming the loop schedule *ls* and its initiation interval *II* are found by software pipelining, then the ***pipelining depth*** of the software pipelined loop, *pd*, is defined as *max (ls(op2,j)+t(op2,j)-ls(op1)) / II* for any nonnegative integer *j* and any two operations *op1* and *op2* of *O*.

Pipelining depth will be used to calculate the code size of a software pipelined loop.

Definition 4.2 The code size of a software pipelined loop is the sum of the code size of the prelude, the pipelined loop body, and the postlude. The code size of the pipelined loop body is equal to the code size of the body of the original loop, which is denoted as *CS0*.

Theorem 4.1 The code size of a software-pipelined loop is equal to *CS0*pd,* where *CS0* is the code size of the original loop body and *pd* is the pipelining depth.

Proof: From Definition 4.1, the pipelining depth is actually the number of iterations which are overlapped in the pipelined loop body, which is also equivalent to the number of unrolled loop bodies. From Definition 4.2, the code size of the software-pipelined loop is the sum of the code size of the prelude, the pipelined loop body and the postlude, which is *CS0*pd*, where *CS0* is the code size of the original loop body and *pd* is the pipelining depth.

From the definition of *pd* and Theorem 4.1, it is clear that we can change the code size of software-pipelined loop by choosing different value of initiation interval II. Now we are ready to present our four algorithms.

Algorithm 1: Finding *min_m* and *max_t*

1. apply a local instruction scheduling approach (list scheduling) on the original loop body;
2. *min_m* = the code size of the original loop body, *CS0*;
3. *max_t* = the execution time of the scheduled loop body.

Algorithm 2: Finding *min_t* and *max_m*

1. apply an existing software pipelining approach (Modulo scheduling or URPR) on the original loop;
2. *min_t* = the initiation interval of the software pipelined loop;
3. *max_m* = the code size of the software pipelined loop.

Algorithm 3: Code-size-constraint software pipelining

1. find *min_m,max_t,min_t* and *max_m*;
2. given the code size, *budget_m*, if *budget_m>=max_m* , then output the software pipelined loop by using Algorithm 2; return (success);
   if *budget_m<min_m*,then return (fail);

3. let the code size of the original loop body be *CS0*, calculate *pd* = integer part of *budget_m/CS0*;
4. apply list scheduling on the original loop body;
5. apply the URPR software pipelining algorithm [7]. The number of pipelined loop bodies is limited to *pd*;
6. output the software pipelined loop, return (success).

Algorithm 4: Finding the relationship between the cycle count and the code size

1. apply Algorithm 1 and 2 to find *min_t, max_t, min_m* and *max_m*;
2. select different *budget_m* points between *min_m* and *max_m*;
3. apply Algorithm 3, obtain the initiation interval for each *budget_m* point;
4. find the relationship between the cycle count and the code size of software pipelined loop

5. EXPERIMENTS

The four algorithms described above were applied to six DSP kernel programs, which are vec_mpy, mac, fir, fir_norld, iir and codebook from [5]. The results are presented in Figure 2. In Figure 2, the point that corresponds to code-size = 100% and cycle count = 100% represents the fully software-pipelined code for those kernel programs generated by TIC62 compiler without using the code size reduction option. All other points in Figure 2 are normalized to this point. The points on the high-end of cycle count are sequential codes generated by TIC62 compiler without software pipelining. The points in the oval shaped area are generated by our code-size-constraint software pipelining algorithms with various values of Tii. We are able to span the tradeoff space of 40% ~ 90% of the original code size with 200% ~20% longer cycle count.

6. CONCLUSION

In this paper, we have presented our code-size-constraint software pipelining approach. Our preliminary experimental results show that the approach can offer a wide tradeoff space for code size vs. cycle count. The wider tradeoff region between time and space provides greater flexibility for the DSP programmer to meet various design criteria. Although the code-size-constraint technique is not currently implemented in DSP compilers, the algorithm can be applied to the sequential code to obtain the desired tradeoff point between time and space. The sequential code is generated by the compiler with the software pipelining feature turned off. In the case where source code is unavailable, it is possible to use software de-pipelining technique [8] to obtain a semantically equivalent loop assembly code, and then apply the Code-size-constraint software pipelining algorithms with various Tii values to obtain the proper tradeoff between code size and cycle count for DSP applications. Currently we are continuing our experiments with larger and more complicated DSP programs.

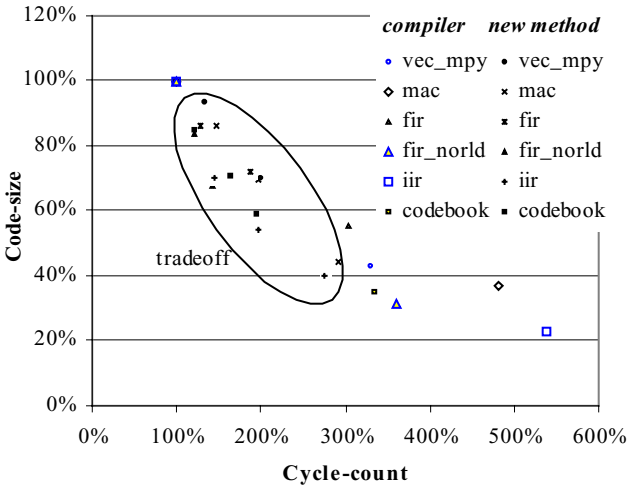

Figure 2 Code size vs Cycle count tradeoff Space

ACKNOWLEDGEMENT

Su and Hu would like to thank ART award from William Paterson University.

REFERENCES

[1] Fisher J. and Rau R., "Instruction-Level Parallel Processing", *Science* vol.253, 1991.

[2] Granston E. etc., Controlling Code Size of Software-Pipelined Loops On the TMS320C6000VLIW DSP Architecture, *Proc. of MICRO-34*, 2001

[3] Kumbhare R. Optimizing DSP Applications on TMS320C6x, *Proc. of ISPC'03*, 2003

[4] Lam M., Software Pipelining: An effective Scheduling Technique for VLIW Machines, *Proc. of SIGPLAN 88 Confernece on Programming Language Design and Implementation*, 1988

[5] Levy M., C Compilers for DSPs Flex Their Muscles, *EDN Magazine*, June 5, 1997

[6] Llosa J. and Freudenberger S., Reduced Code Size Modulo Scheduling in the Absence of Hardware Support, *Proc. of MICRO-35*, 2002.

[7] Su B., Ding S., and Xia J., "URPR - An Extension of URCR for Software Pipelining", *Proc. of the 19th Microprogramming Workshop(MICRO-19)*, Oct. 1986,

[8] Su B., Wang J., Hu E., and Manzano J., De-Pipeline a Software-Pipelined Loop, *Proc. of ICASSP03*, 2003

[9] *TMS320C62x/C67x Programmer's Guide*

[10] Wang J., Eisenbeis C. ,Su B. and Jourdan M., "Decomposed Software Pipelining: A New Perspective and A New Approach". *International Journal on Parallel Processing*, Vol.22, No.3, 1994

[11] Zhuge Q. etc, Optimizal Code Size Reduction for Software-Pipelined Loops on DSP Applications, *Proc. of the 2002 International Conference on Parallel Processing (ICPP 2002)*, Aug. 2002