A VLIW-Based Post Compilation Framework for Multimedia Embedded DSPs with Hardware Specific Optimizations

Meng-Hsuan Cheng, Kenn Slagter, Tai-Wen Lung, and Yeh-Ching Chung

System Software Laboratory
Department of Computer Science
National Tsing Hua University
Hsinchu, Taiwan 30013, R.O.C
{luse,kennslagter,lungkaiser}@sslab.cs.nthu.edu.tw,
ychung@cs.nthu.edu.tw

Abstract. In high performance and low power multimedia embedded system design, VLIW-based embedded DSPs compilers that exploit ILP have become popular and play an important role today. For this reason, we need optimizing embedded DSP compilers that can both generate capable and efficient code in terms of performance, power, size, and productivity. In this paper, we show a post-compilation framework that can further optimize programs that have already been compiled and optimized by another compiler, by using runtime information and exploiting hardware specific features of DSPs. Finally, we show in our simulation results, that even programs compiled at the best optimization level, can obtain significant improvement through the use of this framework.

Keywords: VLIW Compiler optimization, DSP Compiler optimization, Post optimization.

1 Introduction

In multimedia embedded system design, it is desirable for the system to be high in performance and low in cost. To achieve both these goals, VLIW-based DSPs use some hardware specific features, such as zero-overhead loops, vector and pixel sub-words operations, heterogeneous processing units, and compiler-supported branch prediction. These hardware specific features are used to increase computing efficiency instead of using dynamic scheduling logic that would increase hardware complexity and cost. In traditional VLIW-based compiler design, the most important optimization technique is done by using ILP (Instruction-Level Parallelism). For the inquisitive an example of this technique in use can be seen in the IMPACT VLIW compiler framework [4]. However, since there is a tendency for the number of instructions in a basic block of a multimedia program to be small, ILP in a multimedia program that has specialized algorithms and program structure tends to be rather limited. Due to this problem, it is important that a VLIW-based DSP compiler can capitalize on hardware specific features, when it is optimizing an application program. Since hardware specific features tend to be application-friendly not compiler-friendly, most VLIW-based DSP compilers cannot take the advantage of those specialized features effectively. Moreover, in VLIW-based DSPs, the connectivity between computation units and storage units is restricted in order to minimize hardware and interconnection cost. The partial connection of registers and functional units is also an obstacle for VLIW-based DSP compilers to select and schedule instructions effectively. Without an efficient VLIW-based DSP compiler, designers of high performance and low cost multimedia embedded systems are forced to use fully handwritten assembly codes in order to get better performance and code density. However, handwriting assembly code is not an acceptable solution as it tends to result in long development time and it lacks portability.

In general, off the shelf VLIW-based DSP compilers cannot use hardware specific features effectively due to design trade-offs. In this paper, we propose a VLIW-based post compilation framework for multimedia embedded DSPs with hardware specific optimizations. The main purpose of our framework is to enhance the performance of the executable code generated by other VLIW-based DSP compilers. Traditionally, post compilation framework instrumentations are used to provide methods that allow low-level language code such as machine code or assembly code to be optimized. During runtime, a multimedia application program can have its code separated into regions and have each region classified as having either cold region code or hot region code according to the execution time of that region. With this runtime information, compilers can focus their more aggressive optimizations on the hot region codes [17]. By focusing on hot region codes and less on cold region codes compilers can achieve better overall performance. Therefore, most post compilation frameworks tend to focus on specific optimizations such as instruction rescheduling, register reallocation, speculative execution [21], post-pass power optimization and post-pass loop optimization, to enhance the machine code generated by other compilers based on this type of runtime information.

Our proposed framework focuses on instruction rescheduling and post-pass loop optimizations. It consists of six parts, a frontend parser, a run-time information collector, a profiling database synthesizer, a hardware database synthesizer, a hardware specific optimizer, and a code generator. The frontend parser is used to parse the codes generated by other VLIW-based DSP compilers and transform them into the intermediate representation (IR) of our framework. The run-time information collector is used to collect the necessary run-time information. The profiling database synthesizer and the hardware database synthesizer generate useful runtime information about a program as well as hardware related information to help the hardware specific optimizer to optimize a program. The hardware specific optimizer contains two optimization techniques, hardware specific instruction optimizations and hardware specific loop optimizations. The hardware specific instruction optimizations include specific instruction rematch optimizations, instruction rescheduling and recourse reallocation optimization. These optimizations can increase the computation performance, exploit more ILP, and use a compiler-supported profiling-based branch predictor to improve branch performance. For hardware specific loop optimizations, we combine the zero-overhead loop, vector operations, pixel operations, and simple software pipelining techniques to improve the loop performance that dominates multimedia embedded programs. Finally, the code generator is used to generate the enhanced machine code.

To evaluate the performance of the proposed post compilation framework, we implemented a post compilation framework for the ADI Blackfin DSP [1] and used the Blackfin GCC 3.4 [8] and VDSP++ 4.5 [23] as the frontend compilers. Two benchmarks, the DSP Stone [18] benchmark and the JM 9.8 H.264 reference code [10], were used as test samples and were then simulated by the Blackfin cycle-accurate simulator in order to collect runtime information. Experimental results showed that, for the DSP Stone benchmark, our framework on average was able to get 17.5% and 9% performance gain with the codes generated by the Blackfin GCC 3.4 and VDSP++ 4.5 respectively, when using optimization level 3. For the JM9.8 H.264 reference code, which is an optimized DSP library that has been hand-tuned, our framework was able to get a 5.8% performance gain.

The organization of the rest of the paper is as follows. In Section 2, we describe our VLIW-based post compilation framework in more detail. In Section 3, we give more details about the target experimental platform and the results. Finally, in Section 4 we present our conclusions.

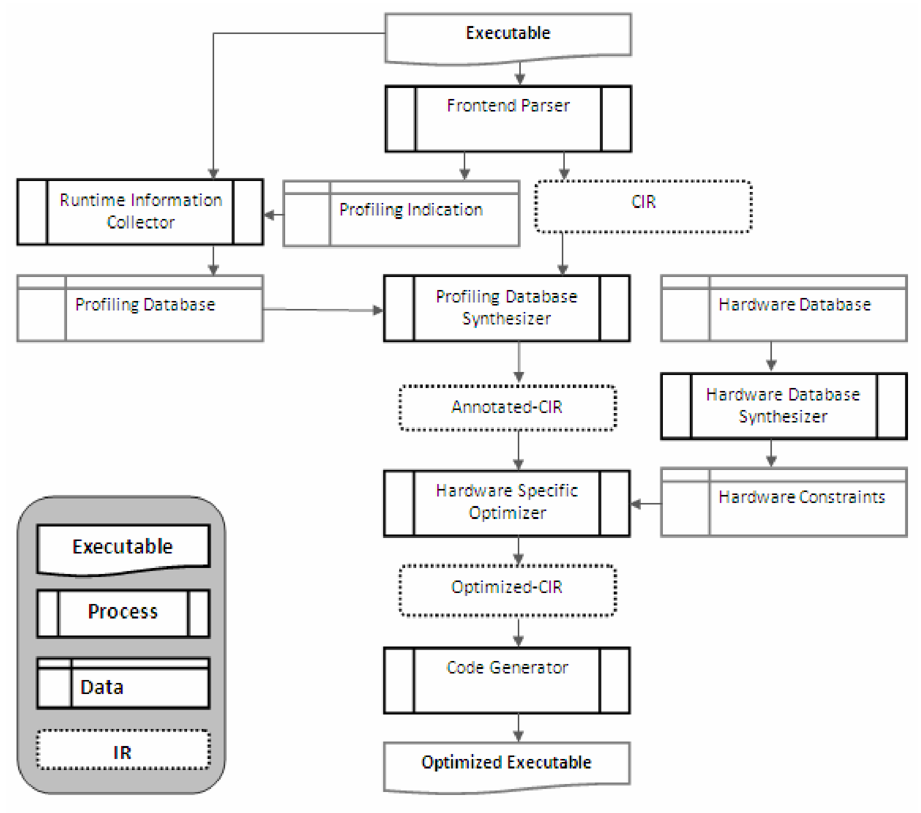

Fig. 1. Compilation flow of the post-compilation framework

2 The Post-Compilation Framework

The architecture of the proposed post compilation framework is shown in Figure 1. As is shown in the figure, the framework consists of six components, a frontend parser, a runtime information collector, a profiling database synthesizer, a hardware database synthesizer, a hardware specific optimizer, and a code generator. The frontend parser is used to parse the machine code generated by a frontend compiler into the common intermediate representation (CIR) and generates profiling indicators that provides hints for the runtime information collector. The runtime information collector is used to record the runtime information, needed for the hardware specific optimizer, into a database and is based on the profiling indicators and the execution of the machine code. The profiling database synthesizer is used to annotate the CIR according to the profiling database. The hardware database synthesizer is used to generate useful hardware information according to the hardware database. The hardware specific optimizer is used to optimize the annotated-CIR with specific hardware features based on any useful hardware information generated by the hardware database synthesizer. The final optimized executable is then generated by the code generator. In the following subsection, we will describe the optimization stage in more detail.

2.1 Hardware Specific Optimizations

The hardware specific optimizer contains two components, a hardware specific instruction optimizer and the hardware specific loop optimizer. The hardware specific instruction optimizer handles optimizations with multiply-accumulate calculations (MAC), ILP scheduling, and with branch prediction. The hardware specific loop optimizer also handles optimization of the zero-overhead loop buffer, the vector-unit and pixel-unit, and in the software pipeline

2.1.1 The Hardware Specific Instruction Optimizer

Since multimedia programs use multiply-accumulate operations frequently, especially in some matrix computing, the multiply-accumulate unit is an important feature of DSPs that can improve the performance of programs that perform multiply-accumulate calculations (MACs). The MAC instruction of most DSPs use specific registers (such as accumulate register). Since defects in a compiler intermediate representation may result in a compiler generating MAC instructions ineffectively, we used IBURG [6] [7] style algorithms, in our framework, to generate MAC tree pattern-matching code in case the core compiler (frontend compiler) lost some MAC instruction matching.

Due to the multiple-issues involved in VLIW-based DSPs, it is often difficult to do scheduling optimizations that can exploit ILP. Since ILP exploited scheduling is a complicated problem that cannot be solved in polynomial time, most DSP compilers cannot generate multiple issue instructions effectively. In our work, we used the artificial resource assignment (ARA) algorithm in [14] [15] combined that with our generated runtime information to solve this problem. This augmented algorithm allowed us to reschedule the result of the frontend compiler. Since register utilization can be increased by using our runtime information to reallocate registers, the artificial resource assignment algorithm is used to help schedule instructions efficiently.

Branch prediction is an important feature in modern deep pipeline design because a stall in the pipeline can affect a systems performance a lot. In order to minimize hardware and power consumption, most DSPs use compiler supported branch prediction (also known as static branch prediction), instead of dynamic branch prediction a mechanism that is implemented in the hardware. For example, the ADI Blackfin DSP supports a special instruction called the branch prediction appendix (bp). The branch prediction appendix helps a compiler improve its branch performance by predicting if a branch is to be taken or non-taken. In general, compilers will predict a backward-going branch as taken and a forward-going branch as non-taken. In the proposed framework, we use a profile based branch predictor [11] to support the branch prediction appendix. In early studies [11], the profile based branch predictor was able to improve prediction rates from 3% to 24% in SPEC92.

2.1.2 The Hardware Specific Loop Optimizer

Zero-Overhead Loop Buffer Optimization

Most DSPs have a zero-overhead loop buffer, which is a hardware technique that can minimize loop overhead without the penalty of increasing code size. The zero-overhead loop buffer is a specific instruction buffer used for low cost counter-based loops. Without decrementing a counter, evaluating a loop condition, then calculating and branching to a new target address, the zero-overhead loop instruction can save 2 to 6 instructions in one loop depending on the structure of the loop. In addition, the zero-overhead loop instruction can also reduce power consumption and increase performance in a low instruction cache utilization environment since it is not accessing the instruction memory. Moreover, when external memory is used in an embedded system, the synchronization of an external memory bus tends to be costly and the data transfer of the external memory bus can consume a lot of energy. With a zero-overhead loop optimization, an embedded system can execute a loop efficiently and use less power.

However, the zero-overhead loop instruction has some restrictions, as it cannot be used with branch instructions or call/return instructions. In the proposed framework, the hardware specific loop optimizer finds loops and tries to optimize them with the zero-overhead loop instruction. Any loop that does not contain a call/return or branch instruction in it can be optimized with a zero-overhead loop instruction easily. However, to optimize loops that contain call/return or branch instructions, the framework has to use other methodologies such as function inlining, loop unswitching[3], and speculative execution in order to take advantage of the zero-overhead loop instruction.

Vector-Unit and Pixel-Unit Optimization

In order to increase the parallelism of a program, some small precision operations can be computed in parallel. This is called data-level parallelism (DLP) [12] and it is independent of the ILP. In modern DSP design, a SIMD operation (also known as a vector operation) is the methodology used to exploit data-level parallelism. SIMD operations are very suitable for several media-processing applications, such as audio, video and image processing. The pixel unit is designed for multimedia applications to take advantage of these instructions to align bytes, perform dual 16 bit and quad 8 bit addition (or subtraction) operations, and for pixel averaging operations. Moreover, a program can have good code-density if it takes advanatge of vector and pixel operations.

In our proposed framework, with run-time value range information, the hardware specific optimizer can identify a variable that is declared as a word (32-bit) but is instead used as a sub-word (8-bit or 16-bit) during runtime. Vector and pixel optimizations can then be performed on these variables by using the algorithm in [24].

Software Pipelining Optimization

In the proposed framework, the hardware specific loop optimizer integrates all the optimizations described above into one simple software pipeline that can construct several loop iterations and combine them into one new loop iteration. Since the new loop can compute several loop iterations and pack them into a single loop iteration, we can issue multiple instructions in parallel and allows us to optimize the software pipeline.

3 Performance Evaluation

To evaluate the performance of the proposed post compilation framework, we implemented a post compilation framework for the ADI Blackfin DSP [1] and used the Blackfin GCC 3.4 [8] and VDSP++ 4.5 [23] as the frontend compilers. Two benchmarks, the DSP Stone [18] benchmark and the JM 9.8 H.264 reference code [10], were used as test samples and were then simulated by the Blackfin cycle-accurate simulator in order to collect runtime information. Tables 1 and 2 show the execution results of the proposed framework for the DSP Stone benchmark, our framework on average was able to get 17.5% and 9% performance gain with the codes generated by the Blackfin GCC 3.4 and VDSP++ 4.5 respectively, when using optimization level 3.

In Table 3, a comparison of the execution cycles on the JM9.8 H.264 reference code is obtained by decoding three frames using a baseline profile with and without the hand-tuned DSP library. From Table 3, we observe that, with the post-optimizing framework, we were able to get an additional 15.16 % performance improvement.

Table. 1. Execution cycles in DSP Stone on Blackfin GCC 3.4

DSP Stone File Name	Execution	Cycles		Performance	Improve
Optimization Level	**None**	**O3**	**Post**	**41.35%**	**17.56%**
complex_multiply	1060	930	719	32.17%	22.69%
convolution	1905	1391	1131	40.63%	18.69%
dot_product	989	942	771	22.04%	18.15%
fir	2391	1556	1305	45.42%	16.13%
fir2dim	9582	5439	4755	50.38%	12.58%
iir_biquad_N_sections	2507	1675	1326	47.11%	20.84%
iir_biquad_one_section	1022	977	834	18.40%	14.64%
lms	3045	1899	1503	50.64%	20.85%
matrix (matrix1)	49926	25377	20283	59.37%	20.07%
matrix(matrix2)	47328	25518	20029	57.68%	21.51%
matrix1x3	1266	1097	974	23.06%	11.21%
n_complex_updates	5689	3856	3445	39.44%	10.66%
n_real_updates	3079	1884	1460	52.58%	22.51%
real_update	935	890	736	21.28%	17.30%
startup	5383	2545	2149	60.08%	15.56%

Table. 2. Execution cycles in DSP Stone on Blackfin GCC 3.4

DSP Stone File Name	Execution	Cycles		Performance	Improve
Optimization Level	**None**	**Opt**	**Post**	**69.39%**	**8.80%**
complex_multiply	456	366	326	28.51%	10.93%
convolution	2257	416	383	83.03%	7.93%
dot_product	474	323	311	34.39%	3.72%
fir	25160	2712	2523	89.97%	6.97%
fir2dim	3460	622	585	83.09%	5.95%
iir_biquad_N_sections	3636	564	514	85.86%	8.87%
iir_biquad_one_section	424	342	312	26.42%	8.77%
lms	4575	1035	705	84.59%	31.88%
matrix (matrix1)	155253	14345	12194	92.15%	14.99%
matrix (matrix2)	147221	15053	12344	91.62%	18.00%
matrix1x3	1147	324	302	73.67%	6.79%
n_complex_updates	11573	1521	1521	86.86%	0.00%
n_real_updates	4633	741	741	84.01%	0.00%
real_update	371	320	320	13.75%	0.00%
startup	14567	2677	2485	82.94%	7.17%

Table 3. Execution cycles on JM 9.8 H.264 decoder using a baseline profile

DSP Stone File Name	Execution	Cycles			Performance	Improve
Optimization Level	**O3 Only**	**O3+DSPLib**	**Post - O3**	**Post - DSPLib**		
JM 9.8 H264 Decoder	450011213	184517090	381791064	173817381	15.16%	5.80%

When the code did not use the hand-tuned DSP library an improvement of 5.8% was obtained when we did use the hand-tuned DSP library. These results indicate that the framework can work well with real multimedia programs even when they use hand-tuned optimizations.

4 Conclusion

In this paper, we proposed a VLIW-based post compilation framework for multimedia embedded DSPs with hardware specific optimizations. The proposed framework was able to better optimize a program by using runtime information and exploiting specific hardware features of an embedded DSP. The hardware specific optimizer in our VLIW-Based post compilation framework was divided into two parts. One was the hardware specific instruction optimizer and the other was the hardware specific loop optimizer. These post-optimizations were shown to have a significant impact on a programs performance. Finally, the simulation results showed that it is possible for our proposed framework to work well with real multimedia programs even if they have had hand-tuned optimizations.

References

1. The Analog Devices, Inc. Website (1995), `http://www.analog.com/en/`
2. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers Principles, Techniques, and Tools, 2nd edn. Addison-Wesley, Reading (2006)

3. Bacon, D.F., Graham, S.L., Sharp, O.J.: Compiler Transformations for High-Performance Computing. ACM Computing Surveys (December 1994)
4. Chang, P.P., Mahlke, S.A., Chen, W.Y., Warter, N.J., Hwu, W.W.: IMPACT: An architectural framework for multiple-instruction-issue processors. In: Proc. 18th. Int. Symp. Computer Architecutre (1996)
5. Falk, H.: Control Flow Optimization by Loop Nest Splitting at the Source Code Level, Research Report No 773 (October 2002)
6. Fisher, J.A., Faraboschi, P., Young, C.: Embedded Computing: a VLIW approach to architecture, compilers and tools. Morgan Kaufmann, San Francisco (2005)
7. Fraser, C.W., Hanson, D.R., Proebsting, T.A.: Engineering a simple, efficient code-generator generator. ACM Letters on Programming Languages and Systems, 213–226
8. The GCC - the gnu compiler collection (1987), `http://gcc.gnu.org/`
9. Gyllenhaal, J.C., Hwu, W.M., Rau, B.R.: Hmdes version 2.0 specification, Univ., Illinois, Urbana, IL, Tech. Rep. IMPACT (1996)
10. The H.264/AVC JM Reference Software, The Image Processing HHI (2006), `http://iphome.hhi.de/suehring/tml/`
11. Hennessy, J.L., Patterson, D.A.: Computer Architecture: A quantitative approach, 4th edn. Morgan Kaufmann, San Francisco (2006)
12. Kozyrakis, C.E., Patterson, D.A.: Scalable Vector Processors for Embedded Systems. IEEE Computer Society Press, Los Alamitos (2003)
13. Marwedel, Goosens, G. (eds.): Code Generation for Embedded Processors. Kluwer, Norwell (1995)
14. Rajagopalan, S., Rajan, S.P., Malik, S., Rigo, S., Araujo, G., Takayama, K.: A Retargetable VLIW Compiler Framework for DSPs With Instruction-Level Parallelism. IEEE Transactions on CAD of IC and System 20(11) (November 2001)
15. Rajagopalan, S., Vachharajani, M., Malik, S.: Handling irregular ILP within conventional VLIW schedulers using artificial resource constraints. In: Proc. Int. Conf. Compilers, Architecture, and Sysnthesis for Embedded Systems, November 2000, pp. 157–164 (2000)
16. Padua, D.A., Wolfe, M.J.: Advanced Compiler Optimizations for Supercomputers. Communication of the ACM (December 1986)
17. Zhang, K., Zhang, T., Pande, S.: Binary Translation to Improve Energy Efficiency through Post-pass Register Re-allocation. In: Proceedings of the 4th ACM international conference on Embedded software (2004)
18. Zivojnovic, V., Velarde, J.M., Schläger, C., Meyer, H.: DSP-stone: A DSP-oriented benchmarking methodology. In: Proc. Int. Conf. Signal Processing Applications and Technology, October 1994, pp. 715–720 (1994)
19. Saghir, M.A.R., Chow, P., Lee, C.G.: Application-driven design of DSP architectures and compilers, Acoustics, Speech, and Signal Processing. In: ICASSP-94 (1994)
20. Kumar, R., Gupta, A., Pankaj, B.S., Ghosh, M., Chakrabarti, P.P.: Post-Compilation Optimization for Multiple Gains with Pattern Matching. ACM SIGPLAN Notices (2005)
21. Liao, S.S., Wang, P.H., Wang, H., Hoflehner, G., Lavery, D., Shen, J.P.: Post-Pass Binary Adaptation for Software-Based Speculative Precomputation. In: ACM PLDI'02 (June 2002)
22. Angiolini, F., Menichelli, F., Ferrero, A., Benini, L., Oliveri, M.: A Post-Compiler Approach to Scratchpad Mapping of Code. In: International Conference on Compilers, Architectures and Synthesis of Embedded Systems CASES 2004 (September 2004)
23. The Analog Devices, Visual DSP++, Website (1995), `http://www.analog.com/en/`
24. Suzuki, M., Fujinami, N., Fukuoka, T., Watanabe, T., Nakata, I.: SIMD Optimization in COINS Compiler Infrastructure. In: Proceedings of the Innovative Architecture for Future Generation High-Performance Processors and Systems (2005)