Journal of Zhejiang University SCIENCE A
ISSN 1673-565X (Print); ISSN 1862-1775 (Online)
www.zju.edu.cn/jzus; www.springerlink.com
E-mail: jzus@zju.edu.cn

Efficient SIMD optimization for media processors

Jian-peng ZHOU, Ce SHI[†‡]

(*Department of Information Science and Electronic Engineering, Zhejiang University, Hangzhou 310027, China*)

[†]E-mail: shice@isee.zju.edu.cn

Received Apr. 18, 2007; revision accepted Nov. 27, 2007

Abstract: Single instruction multiple data (SIMD) instructions are often implemented in modern media processors. Although SIMD instructions are useful in multimedia applications, most compilers do not have good support for SIMD instructions. This paper focuses on SIMD instructions generation for media processors. We present an efficient code optimization approach that is integrated into a retargetable C compiler. SIMD instructions are generated by finding and combining the same operations in programs. Experimental results for the UltraSPARC VIS instruction set show that a speedup factor up to 2.639 is obtained.

Key words: Retargetable compiler, Single instruction multiple data (SIMD) instruction, LCC
doi:10.1631/jzus.A071203 **Document code:** A **CLC number:** TP314

INTRODUCTION

Nowadays more and more attention is paid to multimedia application domain. The trend of using multimedia will increase in the future. In the multimedia application, there are many programs involved executing the same operation on different elements of a large data set (e.g., an array). In the traditional computing model, a single instruction can only deal with a single data element, which is not very efficient for the intensive computation of multimedia applications. The single instruction multiple data (SIMD) model allows the same arithmetic or logical operation to be performed on multiple data elements using one instruction and large registers (called SIMD registers). For example, a 64-bit SIMD register can be logically split into four sub-registers to store four 16-bit data elements and identical computations are performed on these data elements simultaneously. It can lead to a more efficient program obviously. Moreover, it also can obtain better register utilization by packing multiple data elements into a single large SIMD register. In order to improve the performance of multimedia process, SIMD model is widely used on generic-purpose processors (e.g., Sun VIS, Intel MMX/SSE/SSE2 and Motorola AltiVec) and DSP processors (such as TI C6x and Philips Trimedia). These processors with SIMD instructions are known as media processors which are designed for handling audio, video, image and communication tasks.

In general, many multimedia applications are written in assembly code by hand. Although this approach can take full advantage of the processor's SIMD capability, it will lead to poor readability of code, portability problem and high cost of software development. With increasingly fierce competition in the market, this approach no longer meets the requirement of the short development cycle. A better solution is to compile the programs written in high-level programming language into SIMD instructions. It can overcome the shortcomings of using assembly code while still taking full advantage of the processor's SIMD capability. However, the traditional code generation techniques are not well suited for SIMD instructions generation (Aho *et al*., 1987). As a result, most current compilers cannot directly exploit SIMD instructions.

This paper presents an approach of SIMD instructions generation for media processor from ANSI C programs, without use of compiler-known built-in

[‡] Corresponding author

functions. And we discuss the challenges and considerations involved in implementing the approach on the LCC compiler (Fraser and Hanson, 1995) for SPARC processor with VIS instruction set.

The rest of this paper is organized as follows. Some related work is discussed in Section 2. Section 3 describes the structure of the LCC compiler. Detailed description of our approach of SIMD instructions generation is provided in Section 4. The testing platform and experimental results are presented in Section 5, and Section 6 concludes the paper.

RELATED WORK

Most compilers have limited ability to exploit SIMD instructions. Many of them provide semi-automatic SIMD instructions support through compiler-known built-in functions, which are special functions embedded in high-level programming languages. They will be mapped to SIMD instructions by compilers. Programmers can write programs in high level programming language and these programs can utilize SIMD instructions as efficiently as those written in assembly language. However, portability of such programs is poor since different target processors offer different sets of compiler-known built-in functions.

It is possible to develop a high level SIMD language which defines a set of common SIMD operations and provide a portable programming model for the SIMD instructions of a variety of media processors. Some SIMD languages such as SWARC (SIMD within a register) (Fisher and Dietz, 1998) and MMC (Multimedia C) (Bulic and Gustin, 2003) have been developed. The disadvantage of this approach is that it introduces a new programming language, requiring the applications to be rewritten to achieve portable usage of SIMD capabilities of the target processors.

Automatic generation of SIMD instructions has been tried out in both academia and industry. Most of the techniques considered in these studies are based on traditional loop-based vectorization (Allen and Kennedy, 1987). Vectorization has been used to generate vector instructions for vector supercomputers. Because of the similarity between vector instructions and SIMD instructions (Ren *et al*., 2003), it is natural applying vectorization to generate SIMD instructions. The strategy of this approach is to find loops which can be vectorized. If vectorization is possible, compiler-known functions are inserted into the source program through language extensions by the compiler (Krall and Lelait, 2000; Sreraman and Govindarajan, 2000; Naishlos, 2004). In (Bik *et al*., 2002) loops are vectorized to generate SIMD instructions by using traditional compiler optimizations and loop transformations. These transformations can increase the opportunities for exploiting implicit parallelism in a program.

On the other hand, there are some approaches (Larsen and Amarasinghe, 2000; Krall and Lelait, 2000; Leupers, 2000; Hohenauer *et al*., 2006; Pryanishnikov *et al*., 2007) targeting basic blocks rather than loops. In (Leupers, 2000) a SIMD instructions generation approach based on the combination of traditional instruction selection and ILP (integer linear programming) is presented. Highly optimized assembly codes with SIMD instructions are obtained in this approach. However, it takes too much time to solve ILP problems and the time required to solve ILP problems may be unacceptable. Larsen and Amarasinghe (2000) proposed an algorithm for SIMD instruction generation. The approach is performed within a basic block by detecting the structurally equivalent statements whose semantics allow them to be executed in parallel. Such statements are packed together into small groups and the groups are merged until they reach the size of SIMD instructions.

Our approach presented in this paper is a basic block approach since it is simple and effective compared with the loop vectorization approach. Furthermore, our SIMD generation algorithm is machine independent. It is implemented as a special optimization on the compiler. The algorithm combines several structurally equivalent intermediate representation (IR) operations into a single IR operation with the corresponding SIMD operators. IR is accepted by the back end of the compiler and SIMD instructions are generated by the code generator.

LCC OVERVIEW

LCC is a widely used compiler for ANSI C (Fraser and Hanson, 1995). It is a retargetable com-

piler and has been ported to the SPARC, MIPS, X86 and other target processors. Similar to most other compilers, LCC as described in Fig.1 can generally be divided into two parts: front end and back end. The front end which is target-independent performs lexical, syntactic, and semantic analysis, IR generation and some target-independent optimizations. During the IR generation stage, LCC generates trees and the task of the back end is mapping the trees into target-dependent assembly code. The back end is further divided into a target-independent part and a target-dependent part which is generated from machine description (MD) file by using the lburg, a code-generator generator (Fraser *et al*., 1992). Each target processor that LCC supports has its own MD file which models the target processor instruction set architecture (ISA). Since the usage of MD file in compiler reduces the difficulty in retargeting, we can quickly get compiler support for a new target processor by rewriting the MD file.

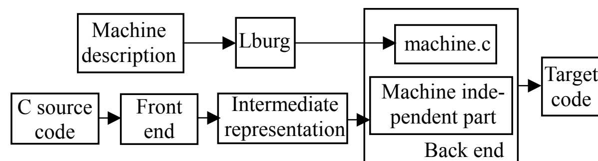

Fig.1 Framework of LCC

The instruction set of the target processor is represented as a set of rules in MD file. Each rule contains a tree pattern which consists of terminal and non-terminal symbols, an assembly code template and a cost part. Non-terminal symbols correspond to statements, variables, registers, constants and so on. Terminal symbols represent operations such as addition, multiplication, loading and storing. The code template part of the rule contains the assembly code which will be inserted into the target code when the rule is used. And the cost part is used by instruction selector to minimize the assembly code size.

The instruction selector is part of the back end and in LCC it is generated automatically from a specification defined in the MD file by lburg. Instruction selection is performed at the tree level. For each tree of the program, the instruction selector uses tree pattern matching and dynamic programming to compute an optimal tree cover with the lowest cost in linear time.

COMPILER OPTIMIZATIONS FOR SIMD INSTRUCTIONS

In this section, our approach of SIMD instructions generation for LCC compiler is presented. As described in the previous section, instruction selection is performed on only one IR tree at a time. But a SIMD instruction generation frequently needs to find operations from different trees. Hence LCC cannot be used directly to generate SIMD instructions.

Our approach is performed on the trees almost directly after the trees have been generated by the compiler's front end. A directed graph called "use graph" whose nodes are data elements is constructed. After the use graph construction, the memory operations of several structurally equivalent trees are grouped together. Each operation accesses data of the same size, and the total size of accessed data is equal to the size of the SIMD register. The memory operations in a group are sorted by their effective addresses. Once the memory operations are grouped, the arithmetic and logical operations of the trees are grouped together. Finally, all grouped operations of the trees are combined into SIMD operations which will generate SIMD instructions in the code generation stage.

Compared to (Larsen and Amarasinghe, 2000)'s approach, our approach is more powerful in terms of utilizing the use graph to manipulate some information of data elements and is also less expensive because analyses are performed on the use graph rather than on the IR structures of the compiler. On the other hand, Larsen and Amarasinghe (2000)'s approach locates statements with adjacent memory references and packs them into groups of two at a time. When more such groups are discovered, all groups are then merged into larger clusters with size consistent with the number of operations that one SIMD instruction can perform. Whereas ours locates statements with adjacent memory references and directly packs them into groups with the same size as that one SIMD instruction can perform. Another major difference is that Larsen and Amarasinghe (2000)'s approach targets three-address representation while our approach is for tree representation. Thus, the code generation for SIMD instructions is quite different.

Loop unrolling

In our approach SIMD instructions generation is focused on the basic block. SIMD instructions are

extremely useful for improving loop performance in multimedia applications. To make the statements in different loop iterations appear in one basic block, loop unrolling technique is needed. Loop unrolling, where loop body is duplicated with a given unrolling factor, can result in large basic blocks and thereby in a high potential for SIMD instructions generation.

The first step of loop unrolling is to compute the unrolling factor. This is done by scanning all the statements in the loop. The unrolling factor is set depending on the data types in the loop body. For example, if the array in the loop contains 16-bit elements and the size of SIMD register is 64 bits, the unrolling factor is set to 4 (=64/16) and three other copies of the loop body are needed.

Use graph

After the loop has been unrolled, the algorithm performs some analyses such as adjacent memory search and data dependence analysis which need some information about the IR nodes, especially the memory addresses. The IR in LCC, i.e. trees, is not suited for these analyses since the information about data arrays such as base address and offset is distributed in various tree nodes. To reduce the complexity of algorithm implementation, a kind of directed graphs defined in (Osman and Williams, 2003) called use graph is constructed.

A node of the use graph represents a variable or an operation in a statement. The variables which are defined in statements are represented as root nodes in the use graph, while the variables which are used in statements are represented as leaf nodes in the use graph. The operator nodes in the use graph represent the arithmetic operations in statements. After the root nodes, operator nodes and leaf nodes are constructed, the edges are then added between relative nodes in the use graph. Figs.2a and 2b show the LCC compiler's IR-tree and the use graph for the statement $a[i]=b+c[i]$, respectively. The tree needs many nodes to represent one variable while the use graph needs only one node, especially for the array variables such as $a[i]$ and $c[i]$ in this example. The use graph is very useful for adjacent memory search and data dependence analysis performed in our approach.

There is data dependence between two nodes if they have the same base address and offset or the same variable. The nodes with data dependence are in the same strongly connected component (SCC). Data dependence analysis is to find the nodes with data dependence and add them into the same SCC. Do this until each node in the use graph appears in one of the SCCs. The trees can be grouped together only if the nodes in the corresponding use graph appear in different SCCs.

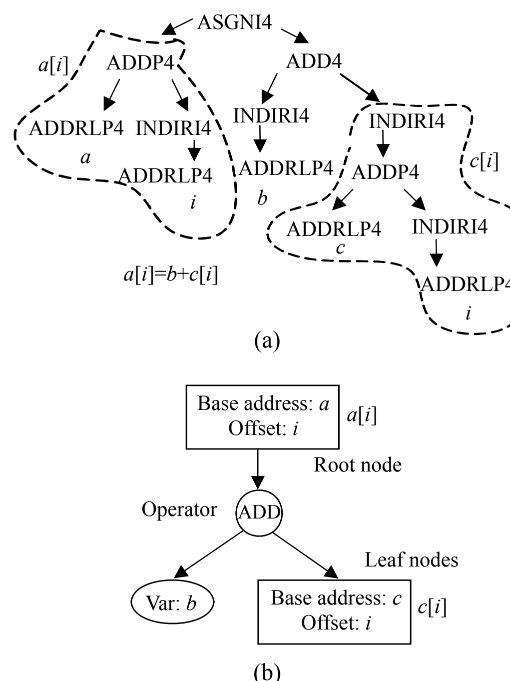

Fig.2 Intermediate representation. (a) LCC's original IR-tree; (b) Use graph

SIMD instructions combination

For the given trees generated by LCC compiler's front end, we perform an algorithm that combines related tree nodes into a new tree node which will be accepted by the instruction selector to generate SIMD instructions. The innermost loops are unrolled a few times which are determined by the unrolling factor and all basic blocks including basic blocks of unrolled loop are inspected. Since memory operations (loads and stores) for SIMD instructions must access consecutive data, adjacent memory searches are performed to find several nodes with memory operations with adjacent memory addresses in the use graph. When the number of selected nodes reaches the number of operations which one SIMD instruction can perform, the trees corresponding to the use graph are grouped into SIMD sets.

A SIMD set is a set of trees whose nodes can potentially be combined into one SIMD node. Once the trees are grouped into a SIMD set, several analyses are performed to check whether the candidates' nodes in the SIMD set can be combined. They can be combined into one SIMD node if

(1) Operations in each candidate's nodes are compatible with each other, i.e., they can be implemented by one instruction in the target ISA;

(2) There is no data dependence between each candidate (check whether the candidates are in different SCCs);

(3) The addresses of memory operations must be consecutive.

After a successful combination of each SIMD set, the trees grouped in SIMD sets are replaced by combined trees with SIMD nodes. The name of LCC's tree node is constructed by a generic operator, a type suffix and a size indicator. For example, given a generic operator "ADD", a type suffix "I", and a size indicator "8", an operator is specified as "ADDI8", which denotes 8-byte integer addition. This method is suited for general operation but is incapable of representing SIMD operation since a SIMD operation contains several individual general operations. Therefore it is necessary to make some modifications on LCC's tree node. Considering the compatibility with the general operation, a tag is attached to the tree node. If the value of the tag is zero, the operator is a general operation; otherwise the operator is a SIMD operation and the value of the tag denotes the number of the sub-registers of SIMD register. For example, for the target processor with 64-bit SIMD register, the operator ADDM8 with tag=4 denotes this operation is a SIMD operation and the individual operation is 16-bit addition, i.e., the operation is a SIMD addition with four individual 16-bit additions operated in the 64-bit SIMD register.

The description of SIMD instructions is added in the MD file. Instruction selector accepts trees with the additional tag in the tree node. The right code will be generated by the code generator according to the tag value.

EXPERIMENTAL RESULTS

This section presents performance improvements with the use of SIMD instructions on the UltraSPARC architecture. The UltraSPARC is LOAD-STORE architecture with VIS instruction set (Tremblay *et al.*, 1996). VIS instruction set is a set of SIMD instructions which are extensions to the standard SPARC V9 instruction set. VIS instructions partition 64-bit floating point registers to hold multiple short integer variables and perform arithmetic and logic computation on the sub-registers, as well as conversion between the different formats such as packing and unpacking instructions.

In our experiments, to show the effectiveness of the proposed algorithms, the speedup factors are evaluated with the use of SIMD instructions for some kernel programs from the DSPStone benchmarks (Zivojnovic *et al.*, 1994). These programs, consisting of vector addition, dot product, fir filtering, matrix operations, n_real_updates and n_complex_updates, show the effect of the proposed algorithm for UltraSPARC processor. All of the programs are executed on UltraSPARC-IIIi 1.5 GHz workstation with Solaris 10 operating system. Execution time and speedup for the test suites are shown in Table 1. VIS-LCC, our compiler which generates VIS instructions, is compared with the LCC compiler which does not use these SIMD instructions. Columns 2 and 3 give the execution time of the benchmarks with and without exploitation of SIMD instructions, respectively. Column 4 shows the speedup for the test suites. Most arithmetic instructions of VIS instruction set work on 16-bit sub-registers, especially the multiplication instructions. Therefore, all experiments have been carried out with 16-bit data types and the unrolling factor is 4. Each test suite is executed multiple times and the execution time is averaged. In Fig.3 we compare the speedups obtained by VIS-LCC with that obtained by GCC 4.1 (Naishlos, 2004) applying vectorization described in Section 2.

Table 1 Speedup of execution time on UltraSPARC processor by using SIMD instructions

Benchmark	Execution time (μs)		Speedup (times)
	VIS-LCC	LCC	
vector addition	101	252	2.495
dot product	572	725	1.267
n_real_updates	164	419	2.555
n_complex_updates	507	1338	2.639
fir	796	725	0.911
matrix1	75 722	106 803	1.410

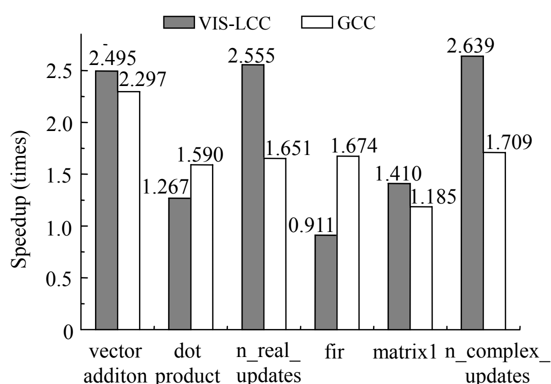

Fig.3 Comparison of speedup between VIS-LCC and GCC (Naishlos, 2004) vectorizer

As illustrated by Table 1, a relatively satisfactory speedup was obtained in most cases. For vector addition, n_real_updates and n_complex_updates, the achieved speedups are between 2.495 and 2.639. Because these programs can almost be completely vectorized, few additional instructions are needed. For the programs of dot product and matrix operations, the speedup is a bit lower since some operations such as reduction, which is used to construct a single value by combining the elements of a vector or array, cannot be used directly to generate SIMD instructions. However, a speedup between 1.267 and 1.410 was still achieved. There is also a counter example, i.e., fir filter, where a slowdown has been measured. A detailed analysis revealed that this is because only a small fraction of this program can be mapped to SIMD instructions and data reorganization is needed in this small fraction program. Thus, it results in many additional instructions such as packing and unpacking instructions.

Fig.3 shows the speedups obtained by our approach and vectorization. An average speedup of 1.880 was achieved by VIS-LCC, while an average speedup of 1.684 was achieved by GCC vectorizer. As illustrated in Fig.3, in most cases our approach obtained better results than vectorization. However, as to dot product and fir, there are some idiom operations such as reduction. Since it is not very good support for these operations in our approach, the speedups are lower than those of vectorization. Therefore, our approach's improvement will focus on idiom operations support in the future.

CONCLUSION

In this paper, a target-independent approach for SIMD instructions generation is presented. This approach has been implemented on LCC. The SIMD-enabled LCC-based compiler can fully utilize the SIMD instructions. The experimental results show that the SIMD instructions are generated efficiently and the performance of SIMD computation is improved. We can easily port the compiler to other media processors with SIMD instruction set since the algorithm is performed before code generation as an optimization of IR. Furthermore, this approach can not only be implemented on LCC but also be applied to any other compiler with the tree IR code.

Future work will focus on recognizing idiom operations such as saturation, min/max, reduction, etc. from multimedia applications. At the same time, we would also like to integrate the energy cost model into our approach to estimate whether it is profitable to use SIMD instructions.

References

Aho, A.V., Sethi, R., Ullman, J.D., 1987. Compilers: Principles, Techniques and Tools. Addison-Wesley Publishing Company.

Allen, R., Kennedy, K., 1987. Automatic translation of Fortran programs to vector form. *ACM Trans. on Programming Languages and Systems*, **9**(4):491-542. [doi:10.1145/29873.29875]

Bik, J.C., Girkar, M., Grey, P.M., Tian, X., 2002. Automatic intra-register vectorization for the Intel® architecture. *Int. J. Parallel Programming*, **30**(2):65-98. [doi:10.1023/A:1014230429447]

Bulic, P., Gustin, V., 2003. An extended ANSI C for processors with a multimedia extension. *Int. J. Parallel Programming*, **31**(2):107-136. [doi:10.1023/A:1022617308483]

Fisher, R.J., Dietz, H.G., 1998. Compiling for SIMD within a Register. Proc. 11th Int. Workshop on Languages and Compilers for Parallel Computing, p.209-304.

Fraser, C.W., Hanson, D.R., 1995. A Retargetable C Compiler: Design and Implementation. Addison-Wesley, Menlo Park, CA.

Fraser, C.W., Hanson, D.R., Proebsting, T.A., 1992. Engineering a simple, efficient code generator generator. *ACM Lett. on Programming Languages and Systems*, **1**(3):213-226. [doi:10.1145/151640.151642]

Hohenauer, M., Schumacher, C., Leupers, R., 2006. Retargetable Code Optimization with SIMD Instructions. Proc. 4th Int. Conf. on Hardware/Software Codesign and System Synthesis, p.148-153. [doi:10.1145/1176254.1176291]

Krall, A., Lelait, S., 2000. Compilation techniques for multimedia processors. *Int. J. Parallel Programming*, **28**(4): 347-361. [doi:10.1023/A:1007507005174]

Larsen, S., Amarasinghe, S., 2000. Exploiting superword level parallelism with multimedia instruction sets. *ACM SIGPLAN Notices*, **35**(5):145-156. [doi:10.1145/358438.349320]

Leupers, R., 2000. Code Selection for Media Processors with SIMD Instructions. Proc. Conf. on Design, Automation and Test in Europe, p.4-8. [doi:10.1145/343647.343679]

Naishlos, D., 2004. Auto-Vectorization in GCC. Free Software Foundation. Http://gcc.gnu.org/projects/treessa/vectorization.html

Osman, S., Williams, R., 2003. Towards Optimal Instruction Vectorization. Http://www.cs.cmu.edu/~sosman/classes/compilers/project/project.ps

Pryanishnikov, I., Krall, A., Horspool, N., 2007. Compiler optimizations for processors with SIMD instructions. *Software Practice and Experience*, **37**(1):93-113. [doi:10.1002/spe.751]

Ren, G., Wu, P., Padua, D., 2003. A Preliminary Study on the Vectorization of Multimedia Applications for Multimedia Extensions. Proc. 16th Int. Workshop on Languages and Compilers for Parallel Computing. Texas A&M University, p.420-435.

Sreraman, N., Govindarajan, R., 2000. A vectorizing compiler for multimedia extensions. *Int. J. Parallel Programming*, **28**(4):363-400. [doi:10.1023/A:1007559022013]

Tremblay, M., O′Connor, J.M., Narayanan, V., He, L., 1996. VIS speeds new media processing. *IEEE Micro*, **16**(4): 10-20. [doi:10.1109/40.526921]

Zivojnovic, V., Velarde, J.M., Schlager, C., Meyr, H., 1994. DSPstone：A DSP-oriented Benchmarking Methodology. Proc. Int. Conf. on Signal Processing Applications and Technology, p.715-720.