A SWP Compiler Design Approach for Multimedia Processing

Dan Wu, Zhiying Wang, Kui Dai, Li Shen
Computer School, National University of Defense Technology
Changsha, Hunan, 410073 China
alavender1979@yahoo.com.cn

***Abstract*—Media and signal processing has forced profound changes not only in current microprocessors, but also in parallel compiling techniques. Upcoming multimedia processors provide special instruction sets for fast execution of signal processing algorithms on different media data types. These instructions are generated with sub-word parallelism (SWP). Much recent work has been done in the context of compilers for SWP, but the exploitation of SWP is still weakly supported by current compilers. To reduce the gap, we present an approach to design a compiling framework to realize automatic SWP exploitation for multimedia processors. This compiling framework is programmed to analyze high-level multimedia programs, identify sub-word parallelism instruction and perform vectorization automatically. And with the aid of scheduling provided by architecture, this compiling framework can realize mapping multimedia applications to the SWP units in such a way that all sub-word units can be active in parallel.**

I. INTRODUCTION

Characteristics of multimedia applications bring a special micro-processing in processor [1]. More memories included and number of calculation units and/or their functionality increases, raises complexity in processor manufacturing. An effective method to exploit available functionality for media applications is sub-word parallelism (SWP). SWP is exhibited by instructions which act on a set of lower precision data packed into a word, resulting in the parallel processing of the data collection. To take advantage, special instruction sets have to be provided for the appropriate microprocessors. Currently, available compilers cannot use such SWP instructions automatically.

In this paper, we present an approach to design a compiling framework to realize automatic SWP exploitation for multimedia processors. In this compiling framework, front-end compiler is built by autovectorizing gcc and back-end is built upon architecture-specific schedule optimization [2]. The automatic SWP identification and utilization in compiler framework with low-level architecture-specific factors differs our work from others.

Sponsored by National Natural Science Foundation of China under Grant NO.60173040.

The rest of the paper is organized as follows. Some common issues, like the basic problems and related work affecting efficient SWP compilation are covered in Section 2. In section 3, we illustrate our approach to exploit SWP for a multimedia processor with subword TTA datapath. In section 4 test results are shown. Finally, conclusions are given.

II. COMMON ISSUES IN SWP COMPILATION

1) ***SWP*** A specific instance of DLP (data level parallelism), which allows the parallel calculation of 2, 4, or 8 similar operations in one calculation unit (SWP unit), if the width of the sub-words used in each operation is the same in each operation and if this sub-word width is one half, one fourth, or one eighth of the calculation unit word width. Figure 1(a) shows an example of 32-bit SWP processing. The main advantage of SWP is its flexibility in supporting many different word lengths (for the subwords) on the same datapath and fully utilizing hardware resources.

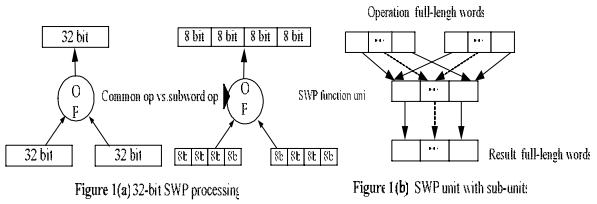

Figure 1. (a) 32-bit SWP processing (b) SWP unit with sub-units

2) ***SWP function unit*** Where sub-word parallelism instructions are calculated.

3) ***Packing/unpacking*** As shown in Figure 1(b), the operands (sub-words, data) for the operations in the SWP function unit have to be provided from two full-length operation words. After calculation in the SWP function unit in parallel, the results are collected again in a full-length word. The full-length result word has to be reorganized after each parallel computation. This data reorganization and the data reading and writing from and to memory are called

unpacking and packing. Packing/unpacking is the main cost for SWP exploitation.

4) ***SWP instruction utilization*** For an application, at least part of it has to be optimized at the assembler level in order to exploit SWP instructions. Alternative compiling approaches include use of macros in high-level code, which are replaced by assembly instructions, and calls to library routines hand-coded by experienced assembly programmers. All these approaches suffer from such shortcomings that lack of portability and high cost of software development. Two vectorisation techniques [3] [4], classical loop-based vectorisation and vectorisation by unrolling, have been integrated in SWP compilation for multimedia applications in multimedia extension architectures (e.g. Intel with MMX). However, both draw out the parallelism depending upon program transforming (e.g. using FORTRAN), with the knowledge of parallel compilation techniques targeted vector processors, and can hardly use SWP instructions automatically. Leupers use integer linear optimization model in compiler as an approach [5]. Those equations of the code will be selected that can be executed in parallel with half word width.

5) ***Architecture limitations*** Multimedia extension architectures, those GPPs (General Purpose Processors) like Intel or AMD processors, usually have only one SWP function unit. While, uniform media processors, those embedded media processors like DSPs, contain more than one SWP function unit in contrast. Packing/unpacking is the main cost requiring additional time and abolish speed-up of SWP function unit in multimedia extension architectures. In uniform media processors, another unit can spared to execute these processing.

From above, we believe only with low-level architecture-specific factors, can vectorization be suitable for SWP exploitation. And uniform media architecture, especially VLIW-based media architecture, is advantageous to exploiting SWP, with compiler's ability of exploiting parallelism and dynamically schedule, besides its flexible datapath.

III. DESIGN APPROACH

In this section, we propose a compiler framework on sub-word TTA (Transport Triggered Architecture [6]) datapath. TTA is an enhanced VLIW architecture. Like VLIW, it offers concurrent execution through a simple and flexible execution model. And the premise behind this architecture is that the compiler is capable of effectively analyzing an application statically in order to extract the available parallelism and target the available hardware resources. But TTA depends even more on compiler than VLIW by trusting tasks of translating operations into data-transports to the compiler. The sub-word TTA datapath is shown in Figure 2.

By adding autovectorization, front-end of the compiler framework is able to analyze high-level multimedia programs, identify sub-word parallelism instruction and perform vectorization automatically.

With the aid of scheduling provided by TTA, the back-end realize mapping multimedia applications to the SWP units in such a way that all sub-word units can be active in parallel. A SAD (sum of differences) algorithm representing motion estimation decoder is examined and implemented in this framework.

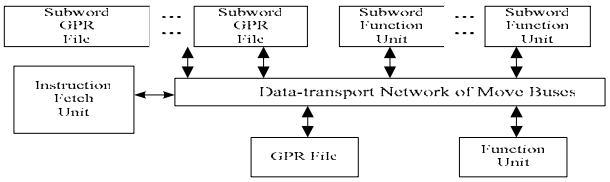

Figure 2 Structure of subword TTA datapath

Figure 2. Structure of subword TTA datapath

The design approach is illustrated in Table I.

TABLE I. ALGORITHIM OF DESIGN APPROACH

Algorithm: Design approach

FOR··· DO
FOR··· DO
Matching
Code transformation
Bitwise dataflow analysis
Control flow analysis
Full length word calculation
Application analysis
Full length word calculation
Matching
End for
End for

The design approach starts with the inner part, application analysis, to find out for loops, which is useful in Matching. Then, full length word calculation identifies the operation sections. Autovectorization is executed including control flow analysis, dataflow analysis, loop transformation and matching. This vectorization is different from others in its DDG (Data Dependence Graph) generated from bitwise dataflow analysis. Code transformation in this framework is combination of some simple loop transformation, like loop fission and loop unrolling.

A. Motivating example analysis

Figure 3 illustrates the extracted codes of motion estimation kernel of MPEG-4 decoder. From this segment, two distinct characteristics can be drawn. The first is inherent parallelism. As seen in Figure 1, the instructions in the for loop have an output dependence (write after write). However, the compiler can easily transform this code and yield

independent instructions. Hence, the data path should have the capability of processing the data in a parallel fashion to allow concurrent execution on independent functional units. The other, execution is performed on subword data. Both pix1 and pix2 arrays are unsigned 8-bit integers. Therefore, the processor must be capable of subword data transfer, storage and processing. This enables efficient use of memory bandwidth and functional units; the latter also decreases the amount of power dissipated by the processor. The simplified loop is shown in Figure 4.

```
int pix_abs16x16_c(UINT8 *pix1, UINT8 *pix2, int line_size, int h)
{
int s, i;
s = 0;
for(i=0;i<h;i++) {
s += abs(pix1[0] - pix2[0]);
s += abs(pix1[1] - pix2[1]);
[…] code deleted for brevity
s += abs(pix1[14] - pix2[14]);
s += abs(pix1[15] - pix2[15]);
pix1 += line_size;
pix2 += line_size;
}
return s;
}
```

Figure 3. Extracted code of motion estimation kernel of the MPEG-4 decoder algorithm

```
for(i=0;i<h;i++) {
S1: C[i]=A[i]-B[i]
S2: D[i+1]=D[i]+C[i]
}
```

Figure 4. A simplified loop from motion estimation kernel of the MPEG-4 decoder

B. SWP instructions identification

Media applications perform many operations on subword operands. Thus, a rich and general subword instruction set is required. Subword instructions process subword data as a SIMD way. How to select and exploit this kind of instruction is a great task burdened on compiler.

There are two major difficulties in exploiting subword instructions. First, parallel loading or storing of values located in sub-registers from/to memory requires establishing that the memory address difference is correct. A more difficult task is to correctly pack potentially parallel instructions together during code generation, so as to form subword instructions. A common solution is to generate subword instructions on-the-fly only during the instruction scheduling and register allocation phases, although possible, would be very difficult, because a large number of constraints need to be obeyed. If, for instance, multiple values share a single 32-bit register, then their live ranges are tightly coupled. As a consequence, standard register allocation techniques such as graph coloring, cannot be applied.

C. Bitwise data flow analysis

We use lattice as a formal ordering of the internal data structure in data flow analysis. Traditionally data flow analysis is viewed as working on propagating the bitwidth of each variable or on maintaining a bit vector for each variable. But the former does not yield accurate results on arithmetic operations. When applying the lattice's transfer function, incrementing an 8-bit number always produces a 9-bit resultant, even though it may likely only need 8-bits. In addition, only the most significant bits of a variable are candidates for bit elimination. And the latter does not support precise arithmetic analysis, for it must still conservative assume that every addition results in a carry.

We select lattice of propagating data-ranges as the structure in data flow analysis. In this structure, a data range is a single connected subrange of the integers from a lower bound to an upper bound. Thus a data-range keeps track of a variable's lower and upper bounds. Because only a single range is used to represent all possible values for a variable, this representation does not permit the elimination of low-order bits. However, it does allow us to operate on arithmetic expressions precisely.

The reason why we choose data-range propagation lies on three facts. The first is this representation maps bitwidth analysis to the more general value range propagation problem, which is known to be useful in value prediction, branch prediction, constant propagation, procedure cloning, and program verification. The second is this representation can achieve exact precision and many applications with arithmetic can benefit from this. The last but the most important, in our compiling algorithm, we use life range as a criterion to process superwords. In this structure, we map each superword's life range to the value range and get the precise result of each life range. Thus, this structure can be viewed as a propagating life-range

The propagation of data range lattice is shown in Figure 5.

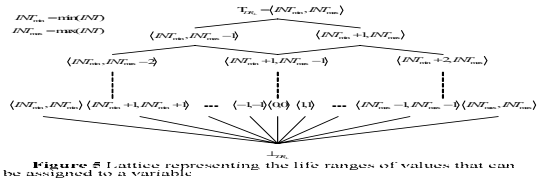

Figure 5. Lattice representing the life ranges of values that can be assigned to a variable

This lattice is lifted from a bottom element. Definitions and computations of the value in this lattice are as follows:

$\perp_{DR_\perp}$: The value of superword's life range that have not been initialized

$\mathrm{T}_{DR_\perp}$: The value that can not be statically determined. It is the upper bound of superwords-set's life range.

$\bigcup$: Life range union. The union over the single connected subrange of the integers where $\langle a_l, a_h \rangle \bigcap \langle b_l, b_h \rangle = \langle \min(a_l, b_l), \max(a_h, b_h) \rangle$

$\bigcap$: Life range intersection. The set of all integers in both subranges where

$$\langle a_l, a_h \rangle \bigcap \langle b_l, b_h \rangle = \langle \max(a_l, b_l), \min(a_h, b_h) \rangle$$

After analysis, we find true dependence and output dependence in this code segment. The data-dependencegraph of this loop is illustrated in Figure6 (left).

D. *Autovectorization under construction*

After loop-distribution, the strong connected component is generated, as shown in Figure 6 (right).

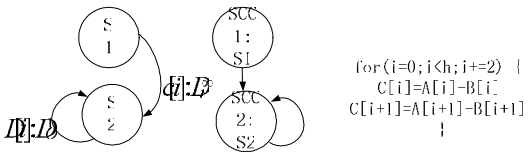

Figure 6. Data-dependence graph and strong connected components

Figure 7. S1 after loop-unrolling

A strong connected component of a dependence graph is a maximal set of vertices in which there is a directed path between every pair of vertices in the set. A non-singleton SCC of a data dependence graph represents a maximum set of statements that are involved in a dependence cycle. We can fine out that statement S1 forms a singleton SCC (without a self-arc). Hence, S1 can be executed in subword way. Statement S2 is self-dependent during the loop. To eliminate this dependence, we can do loop-unrolling on S2. Figure 7 shows the unrolled loop. The loop is first unrolled the correct number of times depending on the type of the operands. In this example, if the register is 32-bit, the loop is unrolled 2 times for short int operands. Then the loop body is inspected and acyclic instruction scheduling is performed in order to have all instances of the same loop instruction grouped together. Hence, S2 can also be executed in subword way.

The *for* loop (used for matching) assumes well defined boundaries. Thus, the specification of SWP description can be started.

IV. EXPERIMENTAL DESIGN

The front-end of the compiling framework is built upon GCC 3.4.0, by planting its compiler gcc, assembler gas and linker gld to the subword architecture, with BSD libraries libc and libm. By adding autovectorization in gcc, the vectorizable serial programs are generated ready for parallization

The back-end is built by utilizing compiling optimizations provided by TTA, including software bypassing, dead result move elimination, operand sharing, operand sharing, socket sharing and scheduling freedom. It needs to read sequential codes from front-end and the architecture description from user. Then, subword parallel codes are realized

Prototype of the compiling framework is shown in Figure 8. After identifying data parallel sections, subword instructions are selected out. Benefited from the subword TTA datapath, they can be mapped to machine-dependent instructions easily. The generated codes afterwards can operate on corresponding registers and sub-registers, so that existing instruction scheduling and register allocation techniques can still be used.

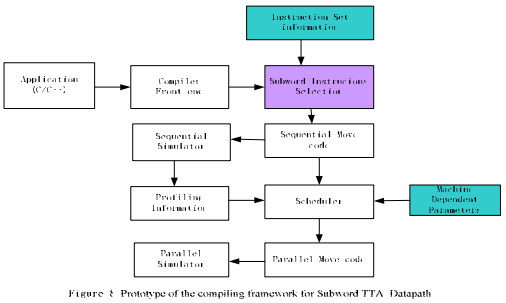

Figure 8. Prototype of the compiling framework for Subword TTA Datapath

V. CONCLUSIONS AND FUTURE WORK

Current compilers are poorly in automatically exploiting SWP. In this paper, we present an approach of compiling framework design for effective and automatic SWP exploitation for multimedia processing. Some novel techniques are explored, including autovectorization and bitwise dataflow analysis. Low-level architecture-specific factors are considered throughout the process of vectorizaiton. The framework is preliminarily experimented on GNU CC with example of MPEG-4. We now mature this framework by compiling media bench. In the next step, more commonly used multimedia processing kernels will be included and the execution time should be compared with that from conventional compilation techniques.

VI. ACKNOWLEDGEMENT

Authors would like to thank all reviewers for their helpful comments.

REFERENCES

[1] Diefendorff, K., Dubey, P.K., “How multimedia workloads will change processor design, “IEEE Computer, pp. 43 - 45, September 1997.

[2] Randy Allen, Ken Kennedy. Optimizing Compilers for Modern Architectures: A Dependence-Based Approach. Morgan Kaufmann, 2002.

[3] N. Sreraman, R. Govindarajan. A Vectorizing Compiler for Multimedia Extensions. In International Journal of Parallel Programming 28 (4), pp. 363-400, August 2000.

[4] Andreas Krall, Sylvain Lelait. Compilation Techniques for Multimedia Processors. In International Journal of Parallel Programming 28 (4), pp. 347-361, August 2000.

[5] R. Leupers, "Code selection for media processors with SIMD instructions", in Proc. DATE'00, 2000.

[6] Henk Corporaal. Microprocessor Architectures: from VLIW to TTA. John Wiley, 1998. ISBN 0-471-97157-X