Automated Specific Instruction Customization Methodology for Multimedia Processor Acceleration[*]

Kang Zhao[1], Jinian Bian[1], Sheqin Dong[1], Yang Song[2], Satoshi Goto[2]
[1]*EDA Lab, Dept. Computer Science & Technology, Tsinghua University, Beijing, P.R. China*
[2]*Graduate School of Information, Production and Systems, Waseda University, Kitakyushu, Japan*
Email: zhao-k04@mails.tsinghua.edu.cn

Abstract

To enhance the computing ability of the multimedia processor, this paper presents an automated specific instruction customization methodology. Specially, this methodology features a profiling stage which is equipped with a sub-graph matching algorithm. Furthermore, to support the features of multimedia systems, three special structures are integrated, such as fusion instructions, parallel and pipelining. To evaluate this methodology, a case study on H.264 encoder is adopted, and a H.264-JM8.0 system is built based on the Xtensa toolset from Tensilica Inc. We have verified it with a set of video benchmarks. The experimental results indicate that 67% enhancement can be obtained via specific instructions.

1. Introduction

Recently, the growing demand of multimedia has been leading to a development of embedded systems, which not only supports ever-increasing functionality, but also needs to be flexible enough [1]. The tradeoff between the efficiency and flexibility can be obtained through the use of application specific instruction-set processor (ASIP).

For many multimedia systems, ASIP can bring better effects than the methods based on only hardware or only software. For example, H.264 is a new standard of video codec. To accelerate its computation, many methods have been proposed, including both hardware implementation [2] and software optimization [3]. However, in the aspects that need both efficiency and flexibility, ASIP will be a better choice. However, using ASIP techniques in H.264 is still in its infancy. For example, Kim presented an ASIP approach for the H.264 system, which uses the combined instructions to enhance the performance [4].

However, the methods above only use the configurable architectures to accelerate the ASIP, which is similar to the hardware implementation. Hence, the potential of is not thoroughly extracted. Because the kernel of ASIP is specific instructions, the customizable instruction set will be used to accelerate the multimedia system in this paper.

Recently, many researches have been done on the specific instruction customization. In the literature [5], an algorithm based on a binary decision tree model was proposed. In addition, a similar approach was introduced by Yu [6], which reduces the exploration space of specific instruction customization by enumerating patterns based on the operations in corn structures. Furthermore, to solve the problem of instruction customization, Sun proposed a synthesis methodology [7] based on the Xtensa core [8].

However, the above techniques have two limitations: first, those methods focus on universal applications and cannot reflect the special requirements of multimedia applications; second, most techniques focus on a part of the whole customization process and specially do not present a methodology for the multimedia applications from the input programs to the final processor synthesis.

To address this issue, a methodology which focuses on the specific instruction customization is proposed in this paper. To satisfy the special requirements of multimedia, special structures are considered. This methodology is built on the Xtensa LX2 core, which is a configurable processor. Finally, to verify the proposed methodology, a case study on the H.264 encoder is presented.

The rest of the paper is organized as follows. In Section 2, the methodology of instruction customization is presented. Then in Section 3, a case study of the H.264 is given. Finally, a conclusion is drawn in Section 4.

2. Methodology

This section first formulates the problem with a data flow graph (DFG) description, and then presents the flow of instruction customization methodology. Finally, the detailed explanations for this workflow are given.

2.1. Problem Formulation

The specific instructions should be extracted from the objective application. Therefore, we first extract the data dependency information from the application programs and formulate it with DFG.

[*] This work was supported by National Natural Science Foundation of China under grant NSFC-90207017, and NSFC-90067001; National Basic Research Program of China (973) under grant 2005CB321605.

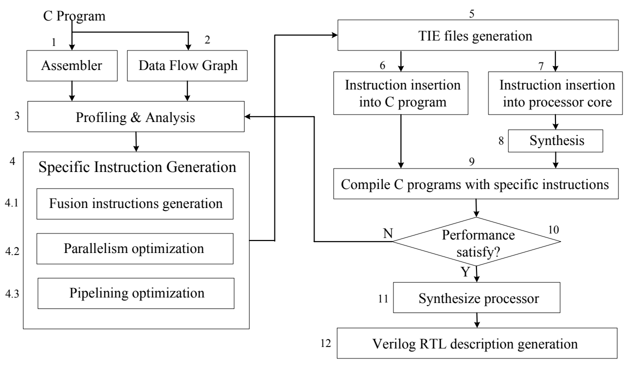
C Program
1
2
Assembler
Data Flow Graph
3
Profiling & Analysis
4
Specific Instruction Generation
4.1
Fusion instructions generation
4.2
Parallelism optimization
4.3
Pipelining optimization
5
TIE files generation
6
Instruction insertion into C program
7
Instruction insertion into processor core
8
Synthesis
9
Compile C programs with specific instructions
N
Performance satisfy?
10
Y
11
Synthesize processor
12
Verilog RTL description generation

Figure 1. Workflow for the specific instruction automated customization

Definition 1: $G(V, E, T)$ is a DFG, where V represents the set of nodes which denotes the operations, and E is the set of arcs which stands for data dependency relations. T is the set of parameters for each node and it represents the execution time of each operation.

Definition 2: For DFG G, the total execution time T_G will be equal to the sum of parameters on the longest path.

Because specific instruction is the combination of basic operations, in DFG it will be mapped to feasible sub-graphs which satisfy several constraints. So this problem can be described as follows.

Problem: Given a DFG $G(V, E, T)$, find a set of feasible sub-graphs $\{G_i'\}$, so that T_G is minimized.

2.2. Design Flow

To solve the problem mentioned above, the details of the methodology are presented. The input is the programs written with C/C++ language, and the final output is the synthesized processor, as shown in Figure 1.

Since the operation is on instruction level, the C code is first compiled and converted to assembler (Step 1), and then the data dependency relations are extracted (Step 2). Based on the assembler and DFG, a profiling process is adopted to find the features of the objective application (Step 3). In this process, a sub-graph matching algorithm is used. Then the specific instructions can be customized based on the profiling results. In this process, three types of special structures are employed: fusion instruction, parallelism and pipelining. The three types of instructions can reflect the features of multimedia programs.

After the specific instruction generation, the focus will be transferred to the method of inserting new instructions into the C code so that both the compiler and simulator can understand them. To settle this issue, an instruction language named TIE is used, which is defined in Xtensa and can be accepted by its compiler and simulator.

2.3. Sub-graph matching algorithm in profiling

The motivation of sub-graph matching is to find the operations which are used frequently. If the frequency is low, it will be unworthy to accelerate it via specific instructions with a cost enhancement. Each specific instruction is mapped to one sub-graph, so a sub-graph matching algorithm can be used to seek out the candidate specific instructions. As shown in Figure 2, the graph A and B both have the same operations "(&)|(>>)", so this format will be synthesized to a specific instruction.

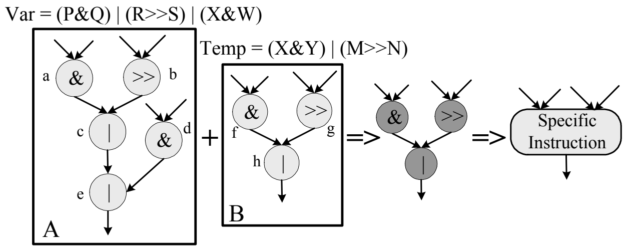

Figure 2. Example explaining the sub-graph matching process

However, the exploration space of this problem is huge. Assume the DFG G has n nodes, so the sub-graph number of graph G is 2^n. Then the number of mapping between the two sets of sub-graphs will be:

$$\underbrace{2^n \times 2^n \times \cdots \times 2^n}_{2^n} = \left(2^n\right)^{2^n}$$

To speed up the exploration and reduce the exponential space, a tool called *pattmatch* is used, which is supplied by Pattlib software [9]. This toolkit utilizes a heuristics isomorphism algorithm with a complexity of $O(n^{\log n})$ to compare different sub-graph patterns in DFG.

2.4. Specific Instruction Generation Strategy

Based on the profiling results, specific instructions will be customized. Three structures are used to achieve this object: fusion instruction, parallelism and pipelining. To explain them clearly, an example is shown in Figure 3(a).

First, fusion instruction is used to merge multiple basic operations into one instruction. Since specific instruction is viewed as hardware for special purpose, the motivation of fusion instruction is similar to hardware acceleration. For the multimedia systems, many complicated operations are used frequently. So we can consider synthesizing the fusion instructions to accelerate the computation. An example is shown in Figure 3(b). The limitation is that it will increase the cost of the final synthesized processor.

Second, the motivation of the parallelism process is to implement sets of operations simultaneously. In this process SIMD (Single Instruction Multiple Data) is used, which describes a set of operations where an identical computation is performed via a series of operands in parallel. In Figure 3(b), SIMD is used to unroll the loop in the C program. This feature can effectively reduce the cycles brought by the iterations. However, SIMD cannot extract the potential of parallelism when the complexity of each instruction is very high.

$$\begin{bmatrix} A_0 \\ A_1 \\ A_2 \\ A_3 \end{bmatrix} = \begin{bmatrix} M_0 & M_1 & M_2 & M_3 \\ M_4 & M_5 & M_6 & M_7 \\ M_8 & M_9 & M_{10} & M_{11} \\ M_{12} & M_{13} & M_{14} & M_{15} \end{bmatrix} \times \begin{bmatrix} B_0 \\ B_1 \\ B_2 \\ B_3 \end{bmatrix} + \begin{bmatrix} C_0 \\ C_1 \\ C_2 \\ C_3 \end{bmatrix}$$

```
void Matrix (A[], B[], C[], M[])
{ int temp[4];
  for (int i=0; i<=3; i++) temp[i]=0;
  for (int i=0; i<=3; i++){
      temp[0]=temp[0]+B[i]*M[i]+C[i];
      temp[1]=temp[1]+B[i]*M[i+4]+C[i];
      temp[2]=temp[2]+B[i]*M[i+8]+C[i];
      temp[3]=temp[3]+B[i]*M[i+12]+C[i];
  }//end for
  for (int i=0; i<=3; i++) A[i]=temp[i];
}//end Matrix
```

(a)

	(1) Fusion instruction	(2) Parallelism	(3) Pipelining
#SI	operation acc (inout x, in y, in z, in w) { x=x+y*z+w; }	operation mua (out x, in y[], in z[], in w) { int n= vector_size(y); // The size of the vector x=TIEmul(y[0:n-1], z[0:n-1])+w; // TIEmul: multiplication of vectors }	operation mul (out x, in y[], in z[]) { int n= vector_size(y); x=TIEmul(y[0:n-1], z[0:n-1]); } operation ada (inout x, in y) { x=x+y; }
#CP	void Matrix (A[], B[], C[], M[]) { int temp[4]; for (int i=0; i<=3; i++) temp[i]=0; for (int i=0; i<=3; i++){ acc(temp[0], B[i], M[i], C[i]); acc(temp[1], B[i], M[i+4], C[i]); acc(temp[2], B[i], M[i+8], C[i]); acc(temp[3], B[i], M[i+12], C[i]); } for (int i=0; i<=3; i++) A[i]=temp[i]; }//end Matrix	void Matrix (A[], B[], C[], M[]) { mua(A[0], B[0:3], M[0:3], C[0]); mua(A[1], B[0:3], M[4:7], C[1]); mua(A[2], B[0:3], M[8:11], C[2]); mua(A[3], B[0:3], M[12:15], C[3]); }//end Matrix	void Matrix (A[], B[], C[], M[]) { mul(A[0], B[0:3], M[0:3]); ada (A[0], C[0]); mul(A[1], B[0:3], M[4:7]); ada (A[1], C[1]); mul(A[2], B[0:3], M[8:11]); ada (A[2], C[2]); mul(A[3], B[0:3], M[12:15]); ada (A[3], C[3]); }//end Matrix

(b)

Figure 3. (a) Example of matrix multiplication. (b) Detailed results for the three-step instruction customization, which are based on the example in figure (a). The three columns present the results after fusion instruction generation, parallelism and pipelining processes respectively. Here "#SI" means specific instructions, and "#CP" means the corresponding C programs.

Third, a pipelining process is utilized to disassemble some specific instructions and run via pipelining. As shown in Figure 3(b), the speed is raised through SIMD instruction *mua*, however, there will be a high complexity for the units to implement this instruction. To reduce the complexity, we can divide it into two parts and run them via pipelining. So *ada* is released and the clock frequency is raised. Besides, due to the pipelining process on *mul* and *ada* instructions, the whole latency is reduced.

3. Case Study of Intra Prediction in H.264

To evaluate the proposed methodology, a case study on the intra prediction application of H.264 is presented.

3.1. Experimental Setup

To customize specific instructions, the experimental platform should satisfy three conditions: first, it should have an analyzer on C programs so that it can analyze which part of the code should be synthesized to specific instructions; second, there must be a configurable core which can be changed along with the generation of new instructions; third, both the compiler and simulator can identify the new instructions inserted in C programs.

Tensilica's Xtensa RB2006.0 toolset for Xtensa LX2 is chosen as the experimental platform, which just satisfies the three conditions above. Its compiler is the subset of gcc, and Xtensa integrates TIE into its library so that new instructions can be compiled and simulated directly. In this experiment, the JM8.0 of H.264 is used as the inputs [10]. To evaluate the proposed methodology, we have implemented a H.264-JM8.0 system based on the Xtensa platform. Then, a configurable core is configured as our baseline processor. This core is equipped with a memory of 1G and runs with a frequency of 319 MHz.

3.2. Profiling and Analysis

Table 1 shows a summary of the profiling results for intra prediction in H.264 encoder. The first column is the function names. The second column displays the number of committed cycles for each function, and the third column under "#rate" shows the percentage of the total cycle count which is spent in executing this function.

From the profiling results in Table 1, we can find that the execution time is typically spent in several functions, such as *Mode_Decision_for_4x4IntraBlock* and *dct_luma*. Based on further analysis on these functions via sub-graph matching, the results indicate that high time consumption is brought by two aspects: complex operations and nesting loops. To optimize these two structures, our strategy is to synthesize fusion instructions first, and parallelism and pipelining optimizations are utilized to unroll the loops and speed up the computation.

3.3. Specific Instruction Customization Results

The customization process includes two steps. First, the complex operations which are used with a high frequency will be synthesized to fusion instructions. This technique is similar to the reference [4]. For example, in the function *intrapred_luma*, many types of complex operations are used frequently, which are partially classified in Table 2. After the fusion instruction generation, the cycle count of function *intrapred_luma* is reduced by 4.1%. Second, the parallelism and pipelining optimizations are adopted to increase the computing speed. Because many of loops are used in this application, there are multiple operations on a set of vector data. Therefore, the acceleration potential can be extracted via parallel and pipeline structures.

To present the effect of the specific instructions, we tested the H.264-JM8.0 system by a set of video benchmarks [11]. Table 3 presents the experimental results, where the first column illustrates the benchmark names, the column with "#frame" denotes the number of frames, and "#reference" means the number of previous frames used for inter motion search. The column under "#original" presents the cycles under basic configurations,

Table 1.
Profiling results for the functions related with intra prediction

Function name	#cycle	#rate
intrapred_luma	60916074	0.41%
intrapred_luma_16x16	44688038	0.30%
dct_luma	1587575203	10.8%
dct_luma_16x16	228965298	1.55%
dct_chroma	269122886	1.83%
OneComponentLumaPrediction4x4	37376669	0.25%
LumaPrediction4x4	38242006	0.26%
IntraChromaPrediction4x4	17028900	0.11%
ChromaPrediction4x4	10247912	0.06%
IntraChromaPrediction8x8	8004903	0.05%
Mode_Decision_for_4x4IntraBlocks	430339090	2.92%
RDCost_for_4x4IntraBlocks	192749556	1.31%
RDCost_for_8x8blocks	19664179	0.13%
RDCost_for_macroblocks	128803212	0.87%
getNeighbour	54966920	0.37%

"#fusion" shows the results after fusion instruction generation, and "#optimization" presents the results after the parallelism and pipelining processing. In addition, "#cycle" means the total number of cycles, and "#imp" represents the percentage of improvements compared to the system without specific instructions.

From Table 3, we can find that specific instructions can achieve about 67% improvements than the core under basic configurations. This case study on the H.264 has offered a powerful proof for the proposed methodology.

4. Conclusion

In this paper, a new specific instruction customization methodology for the multimedia processor acceleration is proposed. It integrates an automated profiling, three-step instruction generation and automated synthesis processes. To evaluate this methodology, a case study of H.264 is employed. The final experimental results indicate that the proposed three-step instruction customization method can effectively reduce the latency of the H.264 system.

Table 2.
Fusion instruction examples for the function *intrapred_luma*

Operation format	#count
reg1=reg2=reg3=reg4=const	634032
reg0=(reg1+reg2+const) / const	462672
reg0= (reg1+reg2+reg3+const) / const	68544
reg0= (reg1+reg2*const+reg3+const) / const	1062432

5. References

[1] F. Sun, and *et al*, "Application-specific heterogeneous multiprocessor synthesis using extensible processors," *IEEE Trans. CAD*, Vol. 25, Issue 9, pp. 1589-1602, 2006.
[2] C. Y. Chen, and *et al*, "Analysis and architecture design of variable block-size motion estimation for H.264 AVC," *IEEE TCAS-I*, Vol. 53, Issue 3, pp. 578-593, 2006.
[3] F. Pan, and *et al*, "Fast mode decision algorithm for intra prediction in H.264/AVC video coding," *IEEE TCAS for Video Technology*, Vol. 15, Issue 7, pp. 813-822, 2005.
[4] S. D. Kim, J. H. Lee, C. J. Hyun and M. H. Sunwoo, "ASIP approach for implementation of H.264/AVC," *Proc. ASPDAC*, Yokohama, Japan, pp. 758-764, 2006.
[5] K. Atasu, and *et al*, "Automatic application specific instruction-set extensions under micro-architectural constraints," *Proc. DAC*, CA, pp. 256-261, 2003.
[6] P. Yu and T. Mitra, "Scalable Custom Instructions Identification for Instruction-Set Extensible Processors," *Proc. Conference on Compiler, Architectures and Synthesis for embedded systems*, USA, pp. 69-78, 2004.
[7] F. Sun, S. Ravi, and *et al*, "Custom-instruction synthesis for extensible-processor platforms," *IEEE Trans. CAD*, Vol. 23, Issue 2, pp. 216–228, 2004.
[8] Tensilica Inc.: http://www.tensilica.com/
[9] Pattlib: http://sourceforge.net/projects/pattlib/
[10] H.264 reference: http://iphome.hhi.de/suehring/tml/
[11] Xiph Test Media: http://media.xiph.org/video/derf/

Table 3.
Experimental results on a set of video benchmarks for H.264

Benchmarks	#frame	#reference	#original	#fusion		#optimization	
			#cycle (×10^9)	#cycle (×10^9)	#imp	#cycle (×10^9)	#imp
foreman	300	5	14.690870364	11.913480837	18.91%	3.475919012	76.34%
akiyo	300	3	9.269737320	7.246989328	21.82%	3.290705323	64.50%
container	300	4	10.137229051	8.261425415	18.50%	3.405389364	66.41%
bowing	300	3	9.335069590	7.698927509	17.53%	3.173851207	66.00%
carphone	382	2	9.947224374	8.891701875	10.62%	3.338589347	66.44%
claire	494	3	9.688303345	8.403139177	13.27%	3.279317640	66.15%
coastguard	300	4	13.350951331	13.040796922	2.33%	4.183072978	68.67%
hall_monitor	300	4	9.896111243	7.918765618	19.98%	3.341251978	66.24%
husky	250	3	12.361467688	11.278290829	8.77%	4.010371098	67.56%
miss_am	150	5	9.442842319	8.636607660	8.54%	3.197977006	66.13%
suzie	150	5	10.020138039	8.759141504	12.58%	3.361292739	66.45%