Proc. International Workshop on Synthesis And System Integration of MIxed Technologies (SASIMI '97)
Osaka, Japan, August 1997

Instruction Code Compression for Application Specific VLIW Processors Based on Automatic Field Partitioning

Nagisa Ishiura[†]
ishiura@ise.eng.osaka-u.ac.jp

[†] Dept. Information Systems Engineering
Osaka University
Suita, Osaka, 565 Japan

Masayuki Yamaguchi[†,††]
masa@edag.ptdg.sharp.co.jp

[††] Precision Technology Development Center
Sharp Corporation
Tenri, Nara, 632 Japan

This paper presents a method of reducing the instruction memory size of application specific VLIW processors. Given an object code of a processor, we try to compress it by recoding each instruction word with a smaller number of bits. The problem of finding a good coding which achieves large reduction and yet curbs the size and the delay of the decoder is reduced to the problem of finding a good field partitioning. By establishing a relation between a field partition and the cost of the program and decoders, we formulate an optimum field partitioning problem in which a field partition minimizing the cost of the program and the decoder is sought. We have developed an approximation algorithm to solve this problem. In a preliminary experiment on three sets of practical VLIW processors and application programs, the total objective cost, which is defined as the sum of the total bits of the coded program and the size of the ROMs to implement the decoder, is reduced into 46%∼60% of the original ones.

I. Introduction

A VLIW architecture is becoming increasingly popular in embedded processor design, for the simultaneous activation of multiple hardware resources enables fast execution of softwares in audio and video applications. A drawback of this architecture is that the memory space to store the instruction code becomes larger. The long instruction word, whose every bit may not be utilized in every execution step, leads to large instruction memory size. Since the size of the memory is directly related to the hardware cost and the power consumption, the reduction of the instruction memory size will be beneficial to the overall cost-performance of the system.

One resource we can exploit for this purpose is the fact that processors are application specific. By making certain modifications to the instruction set or on the instruction formats taking the characteristics of the given program into account, we may be able to reduce necessary bits to store the program.

Unlike the methods of redesigning both the instruction set and the datapath architecture, or the methods of synthesizing an instruction set and a datapath architecture from a given set of benchmark programs [Hua93], we start from where a datapath architecture and an application program to be executed on it are given. We assume a hardware designer has "done a good job" in designing datapath architecture taking the characteristics of the program and various constraints on the hardware into consideration, using support tools such as an architecture evaluator [Yam97]. Thus what we try to do here is to compact the program without touching the datapath.

One method of doing this is to store the program in a compressed form (Figure 1). When the processor fetches the next instruction, it is supplied by way of a decoder that expands the compressed code. So far, a cache-based method [Koz94] and a code-by-code method [Yos97] have been proposed. The former encodes the program by the unit of a cache block. It achieves high compression ratio with the use of variable length coding, but the delay and the cost of the decode engine will be significantly large. Moreover, it does not fit in the architectural scheme if the processor does not employ an instruction cache. The latter encodes each instruction word by fixed length coding. If the program consists of M different code words, then each word can be encoded with $\lceil \log M \rceil$ bits. The problem again is the the size of the decoder. Even in the code-by-code approach, the decoder size, measured in ROM bits to implement the decoder, is $2^{\lceil \log M \rceil} \times$(instruction word size) bit large and could offset the gain obtained by the code compression.

In this paper, we propose an improved code-by-code method which can reduce the program size while curbing the decoder size. The idea is to introduce *field partitioning*. As is usually done in designing vertical microinstruction sets from horizontal microinstruction sets, the original instruction word is partitioned into several fields so that the decoder for each field stays within a reasonable size. In order to obtain a good field partition automati-

cally, we formulate a field partitioning problem and give an approximate solution to this problem.

This method is applicable to all kinds of processors with fixed instruction word length, but we think it is especially effective for VLIW processor. In a preliminary experiment on three sets of industrial VLIW processors and application programs, the total objective cost, which is defined as the sum of the total bits of the coded program and the size of the ROMs to implement the decoder, is reduced into 46%∼60% of the original ones.

In the following sections, we first formulate the optimum field partitioning problem and then show an approximation algorithm. After presenting the experimental results, we discuss pros and cons of our approach.

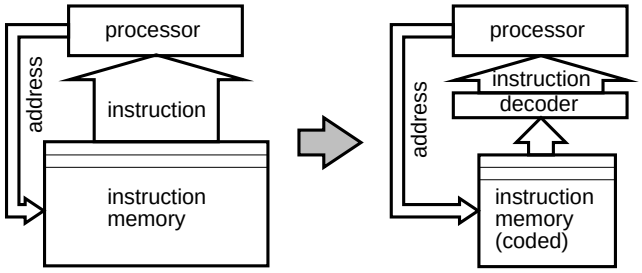

Figure 1: Configuration of a processor and an instruction memory

II. Optimum Field Partitioning Problem

A. Formulation

Let a given application specific program consist of N instructions, each of which is w bit long. Let $B = \{b_1, b_2, \cdots, b_w\}$ be the set of bit positions in an instruction word. Then a *field partition* $F = \{f_1, f_2, \cdots, f_n\}$ is a partition of B where $f_i \subseteq B$, $f_1 \cup \cdots \cup f_n = B$, $f_i \neq \phi$, and $f_i \cap f_j = \phi$ for $i \neq j$. We call each f_i a field. Let $M(f_i)$ be the number of different bit patterns appearing in field f_i of the program. For example, if a program consists of only ADD and SUB instructions, then the number of different bit patterns that appears in the field specifying the operation is two. Let $m(f_i)$ be the number of bits to encode the $M(f_i)$ patterns of field f_i. Since we assume the minimum number of bits, we have

$$m(f_i) = \lceil \log M(f_i) \rceil.$$

The size of the (coded) program for partition F, denoted by $P(F)$, is expressed as

$$P(F) = \sum_{f_i \in F} N \cdot m(f_i).$$

The decoder size $D(F)$ for partition F is

$$D(F) = \sum_{f_i \in F} d(f_i)$$

where $d(f_i)$ is the decoder size for field f_i. There are various ways of defining a decoder cost, but here we define it as the size of the ROM, in terms of bits, which implements the decoder. Thus $d(f_i)$ is defined as

$$d(f_i) = \begin{cases} |f_i| \cdot 2^{m(f_i)} & (m(f_i) \geq 2) \\ 0 & (m(f_i) \leq 1) \end{cases}.$$

The reason why $d(f_i)$ is 0 for $m(f_i) \leq 1$ is that decoder is implemented with only wires or inverters. Our objective cost $T(F)$ is the sum of the program size and the decoder size.

$$T(F) = P(F) + D(F).$$

Finally, an *optimum field partitioning problem* is a problem to find a field partition F that minimizes $T(F)$ for a given program.

1	2	3	4	5	6	7	8	9	10
0	0	1	0	0	0	1	0	0	1
0	0	1	0	0	0	1	0	1	0
0	0	1	0	0	0	1	1	0	0
0	0	1	0	0	0	1	1	0	0
0	0	1	0	0	0	1	1	0	0
0	0	1	0	0	0	1	1	0	0
0	0	1	0	0	1	0	0	0	1
0	0	1	0	0	1	0	0	1	0
0	0	1	0	0	1	0	0	1	0
0	0	1	0	0	1	0	0	1	0
0	0	1	0	1	0	0	0	1	0
0	0	1	0	1	0	0	1	0	0
0	0	1	1	0	0	0	0	0	1
0	0	1	1	0	0	0	0	0	1
0	1	0	0	0	0	1	0	0	1
0	1	0	0	0	0	1	0	1	0
0	1	0	0	0	0	1	1	0	0
0	1	0	0	0	0	1	1	0	0
0	1	0	0	0	0	1	1	0	0
0	1	0	0	0	1	0	0	0	1
0	1	0	0	0	1	0	0	1	0
0	1	0	0	1	0	0	0	1	0
0	1	0	0	1	0	0	1	0	0
0	1	0	1	0	0	0	0	0	1
1	0	1	0	0	0	1	0	0	1
1	0	1	0	0	1	0	0	0	1
1	0	1	1	0	0	0	0	0	1
1	1	0	0	0	0	1	0	0	1
1	1	0	0	0	1	0	0	0	1
1	1	0	1	0	0	0	0	0	1

Figure 2: Sample program.

B. Examples of Field Partitioning

Let us illustrate how field partitioning works using a simple example in Figure 2. The instructions are sorted for easier understanding. It consists of 30 instructions and each instruction word is 10 bit long. The original program size is $10 \times 30 = 300$ bits.

In the 30 instructions, we have only 22 different instructions, for some instructions are repeatedly used. Then we can code the 22 patterns using $\lceil \log 22 \rceil = 5$ bits. By this coding, the program size becomes $5 \times 30 = 150$ bits, the half of the original one. However, we must count the

cost of the decoder. If it is implemented by an ROM, the necessary bits are $2^5 \times 10 = 320$. The total cost of $150 + 320 = 470$ bits is even larger than the original program.

Now we see a few ways of field partitioning. The first three bits (bits 1–3) may correspond to a field that chases ALU operations. In this example, we see only four different patterns on this three bit field. Then we can recode them using two bits. Similar y bits 4–7 that presents only four different one-hot-code patterns can be coded with two bits. This kind of patterns may be generated in arbitrating bus access or selecting operands. Since the final bits 8–10 shows only three distinctive patterns and coded with two bits, the coded instruction word will consist of $2 + 2 + 2 = 6$ bits (see Figure 3 (a)). The decoder sizes are

$$2^2 \times 3,\ 2^2 \times 4, \text{ and } 2^2 \times 3$$

which accumulate to 40 bits. Total hardware cost with this field partitioning is $6 \times 30 + 40 = 220$ bits.

Depending on programs, it is sometimes effective to make the field larger. By grouping bits 4–10 into a single field, we find only eight patterns out of possible twelve patterns. This corresponds to the case where the patterns of operand references are somehow limited. If we code this with three bits, as shown in Figure 3 (b), the instruction word becomes 5 bit long. Since the decoder size is $2^2 \times 3 + 2^3 \times 7 = 68$, the total cost is $5 \times 30 + 68 = 218$ bits and slightly better than the previous one.

We need not restrict ourselves to the knowledge of a relation between the bits and the hardware control. We could detach the bit 1 from bits 2–3 and join it with bit 8–10. The resulting field partition (Figure 3 (c)) also produces an instruction format of 5 bit long. The decoder cost of $0 \times 2 + 2^2 \times 4 + 2^2 \times 4 = 32$ bits and the total cost becomes $150 + 32 = 182$ bits. In this way, we can make good use of a correlation between a specific operation and its operand reference patterns.

f_1

1 2 3	
0 0 1	0 0
0 1 0	0 1
1 0 1	1 0
1 1 0	1 1
	1 2
	f_1'

f_2

4 5 6 7	
0 0 0 1	0 0
0 0 1 0	0 1
0 1 0 0	1 0
1 0 0 0	1 1
	3 4
	f_2'

f_3

8 9 10	
0 0 1	0 0
0 1 0	0 1
1 0 0	1 0
	5 6
	f_3'

(a) Field partitioning (1).

f_1

1 2 3	
0 0 1	0 0
0 1 0	0 1
1 0 1	1 0
1 1 0	1 1
	1 2
	f_1'

f_2

4 5 6 7 8 9 10	
0 0 0 1 0 0 1	0 0 0
0 0 0 1 0 1 0	0 0 1
0 0 0 1 1 0 0	0 1 0
0 0 1 0 0 0 1	0 1 1
0 0 1 0 0 1 0	1 0 0
0 1 0 0 0 1 0	1 0 1
0 1 0 0 1 0 0	1 1 0
1 0 0 0 0 0 1	1 1 1
	3 4 5
	f_2'

(b) Field partitioning (2).

f_1

2 3	
0 1	0
1 0	1
	1
	f_1'

f_2

4 5 6 7	
0 0 0 1	0 0
0 0 1 0	0 1
0 1 0 0	1 0
1 0 0 0	1 1
	2 3
	f_2'

f_3

1 8 9 10	
0 0 0 1	0 0
0 0 1 0	0 1
0 1 0 0	1 0
1 0 0 1	1 1
	4 5
	f_3'

(c) Field partitioning (3).

Figure 3: Examples of field partitioning.

III. Approximation Algorithm

We have developed an approximation algorithm to solve the field partitioning problem, for it would take enormous computation time to find exact solutions.

Our algorithm tries to search an optimal partition by merging fields starting from the primitive partitioning $F = \{\{b_1\}, \{b_2\}, \cdots, \{b_w\}\}$ where each field consists of a single bit. Merges of the largest cost gains are applied repeatedly until we have no more cost improvement.

This scheme is, unfortunately, easily trapped by local optima. Merges which would produce good field partition in the future are often rejected because the cost gain at hand is zero. To counter this problem, we introduce auxiliary costs $T'(F)$ and $T''(F)$ which can express the quality of the merges more precisely.

Let us consider the example of Figure 4 (a). If we merge 1, 2, and 3, the number of the different patterns on this

1 2 3
0 0 0
0 0 1
1 1 1
1 0 1

(a)

1 2	3 4	5
0 0	0 0	0
0 0	0 0	1
0 1	0 0	0
0 1	0 1	1
1 0	1 0	0
1 0	1 0	1
1 1	1 0	0
1 1	1 1	1

(b)

Figure 4: Fraction of a program.

field will be four. The field can be recoded with two bits and we can save one bit. However, in order to merge the three bits by our algorithm, we should first merge either two bits of them. The merge of any two bits of the three results in three different patterns, which requires two bits. Since the merge does not reduce the necessary bits for the instruction word, the merge of any two bits is rejected and we have no way to merge the three bits.

This type of rejection of merges is due to the fact that the cost is not improved until the number of different patterns decreases below 2's power. The merge which largely reduce the number of different patterns should be selected even if the cost gain on the spot is zero, for it may lead to significant reduction of the cost in the future. For this purpose, we use exact value of $\log M(f_i)$, instead of $\lceil \log M(f_i) \rceil$, to represent the necessary bits to encode the field. In the example above, the necessary bits to recode the new field obtained by merging bits 1 and 2 is evaluated to be $\log 3 = 1.58$, instead of $\lceil \log 3 \rceil = 2$, which makes the gain positive. The approximate cost function $T'(F)$ is created in this way, which is formally defined as follows:

$$
\begin{aligned}
m'(f_i) &= \log M(f_i), \\
d'(f_i) &= |f_i| \cdot 2^{m'(f_i)}, \\
P'(F) &= \sum_{f_i \in F} N \cdot m'(f_i), \\
D'(F) &= \sum_{f_i \in F} d'(f_i), \\
T'(F) &= P'(F) + D'(F).
\end{aligned}
$$

Unlike in the exact cost $T(F)$, the decoder cost $d'(f_i)$ is not defined as zero even for $m'(f_i) \leq 1$. This is due to the same reason why we do not round up the value of $m'(f_i)$.

The other cost function $T''(F)$ attempts to evaluate the quality of the merges even more precisely than $T'(F)$. Consider an example in Figure 4 (b). Both the merge of bits 1, 2 and the merge of bits 3, 4 result in four different patterns and thus present the same cost gains in the previous cost functions. However, the distribution of patterns is less uniform in the field (3,4) than (1,2). Since the merges resulting in biased distribution can lead to the reduction of different patterns in future merges, it should be given a higher priority. (Compare the merge (1,2,5) and (3,4,5)). The bias is well-known to be numerated as entropy. It fits into our scheme very well, because the entropy of an information source works as a lower bound of the number of bits to encode the symbol of the source. Let $H(f_i)$ be the entropy of the set of patterns appearing in field f_i. Then the cost $T''(F)$ is defined as follows:

```
F ← {{b_1}, {b_2}, ···, {b_w}};
repeat {
    find (f_i, f_j) ∈ F × F that maximize G(F, f_i, f_j);
    if (G(F, f_i, f_j) > 0) merge f_i and f_j;
    else {
        find (f_i, f_j) ∈ F × F that maximize G'(F, f_i, f_j);
        if (G'(F, f_i, f_j) > 0) merge f_i and f_j;
        else {
            find (f_i, f_j) ∈ F × F that maximize G''(F, f_i, f_j);
            if (G''(F, f_i, f_j) > 0) then merge f_i and f_j;
        }
    }
} until (no more merge occurs)
```

Figure 5: Approximation algorithm for field partitioning.

$$
\begin{aligned}
m''(f_i) &= H(f_i), \\
d''(f_i) &= |f_i| \cdot 2^{m''(f_i)}, \\
P''(F) &= \sum_{f_i \in F} N \cdot m''(f_i), \\
D''(F) &= \sum_{f_i \in F} d''(f_i), \\
T''(F) &= P''(F) + D''(F).
\end{aligned}
$$

Our approximation algorithm is constructed using the three cost functions as listed in Figure 5. Here, $G(F, f_1, f_2)$ is the gain of the cost T obtained by merging f_1 and f_2 in field partition F. Let $F_{(f_1,f_2)}$ denote the new field partition obtained by merging f_1 and f_2 in field partition F, then $G(F, f_1, f_2) = T(F) - T(F_{(f_1,f_2)})$. $G'(F, f_1, f_2)$ and $G''(F, f_1, f_2)$ represent the gains of costs T' and T'', respectively, which are defined in the same way. Starting from an initial partitioning where each field consists of a single bit, merges of two fields are tried until there are no candidates. If there exist field pairs that improves $T(F)$, then a merge of the largest gain is adopted. If no field pairs improve $T(F)$, then a merge that shows the largest gain of $T'(F)$ is adopted. Furthermore, if we have no field pairs that improve $T'(F)$, then a merge of the largest gain in $T''(F)$ is adopted. Finally, when even the improvement of $T''(F)$ is impossible, the algorithm finishes.

In order to find a pair of fields (f_i, f_j) that maximize $G(F, f_i, f_j)$, $G'(F, f_i, f_j)$, and $G''(F, f_i, f_j)$, we must compute the gains for all the possible pairs at the beginning. Since it takes $O(N)$ to compute the gains for a pair, the computation time for all the pairs is $O(Nw^2)$. Every time a pair is merged, we must have the gains for the pairs associated with the new fields created by the merge. This costs $O(Nw)$ time for each creation of a new field, which accumulates to $O(Nw^2)$ throughout the computation. Thus the total computation time of this approximation algorithm is $O(Nw^2)$.

Table 1: Experimental results.

		code length (bit)	program P (bit)	decoder D (bit)	total T (bit)	%
P1	vliw	80	227,360	0	227,360	100.00
	min	11	31,262	163,840	195,102	85.81
	auto	26	73,892	31,233	105,124	46.24
P2	vliw	52	183,404	0	183,404	100.00
	min	11	38,877	122,880	161,877	88.15
	auto	26	74,067	36,360	110,427	60.21
P3	vliw	52	249,392	0	249,392	100.00
	min	11	52,756	122,880	175,636	70.43
	auto	20	95,920	24,072	119,992	48.11

IV. Experimental Result

We implemented an experimental version of a code compression program based on the method described so far. We applied it to three industrial programs P1, P2, and P3 for embedded VLIW processors. P1 and P2 are programs for audio encoding/decoding and P3 for video encoding/decoding. The size of the programs are 2,842, 3,527, and 4,796 instructions, respectively.

Table 1 shows the evaluation of the code length, program size P, decoder size D, and sum of the program and decoder T for three field partitioning. "Vliw" is the initial (one-bit per one-field) partitioning which corresponds to the original VLIW code. "Min" is the partitioning where all bits are grouped into a single field. This corresponds to the encoding method [Yos97] without field partitioning. "Auto" is the field partition obtained by our algorithm.

It turned out that our algorithm sometimes tend to do excessive merge of fields which makes decoder cost significantly large. In order to avoid this, we rejected the merge which makes the ratio of $D(F)$ to $P(F)$ greater than a constant r. (The value of r is set to 0.7 in this experiment.)

Although "min" reduces the program size the most, the decoder costs are so large that the total costs remain 70% $\sim$ 85% of the original ones. Our method, on the other hand, reduces the cost to 46% $\sim$ 60%.

The program is currently implemented in perl and the computation time for P1, P2, P3 on UltraSPARC (200MHz) is 235 sec., 180 sec., and 169 sec., respectively.

V. Conclusion

We presented a method of reducing size of object codes of application specific VLIW processors.

The most simple way of applying this method to embedded system design is to store coded program in the instruction memory and to expand them by a decoder attached to a processor core. We may be able to reduce the chip area for programs, the power consumption, and also the bandwidth between the instruction memory and the processor.

If the decoding delay is critical, we may have to change control pipeline scheme or redesign (resynthesize) the decoder of the processor. We may be able to reduce the overall decode delay by merging our decoder to the original decoder of the processor.

Another application we have in mind is to use this technique to design assistance or automatic synthesis of instruction sets. Our program gives us information on bit correlation which may be utilized in design of instruction format. If we can use compilers from high-level language to horizontal instruction codes, we may directly apply the code compression to the horizontal code, skipping the instruction set design.

One demerit of our scheme in practical system design is that we can not start the implementation of the decoder and a part of control unit until we finish the software. Software design with compiler is a requisite to our scheme.

Finally, it is also obvious that our approximation algorithm leaves much to be improved. It should be reconsidered so that it can find better solutions in shorter amount of time.

Acknowledgement

Authors express their appreciation to Prof. Isao Shirakawa and Mr. Takao Onoye of Osaka University, Dr. Takashi Kambe and Dr. Tetsuya Fujimoto of Sharp Corporation for their helpful advice and discussion.

References

[Hua93] I.-J. Huang, B. Holmer, and A. Despain: "ASIA: Automatic Synthesis of Instruction-Set Architectures," in *Proc. SASIMI '93*, pp. 15–22 (Oct. 1993).

[Koz94] M. Kozuch and A. Wolfe: "Compression of Embedded System Programs," in *Proc. IEEE Int. Conf. Computer Design (ICCD '94)*, pp. 270–277 (Oct. 1994).

[Yam97] M. Yamaguchi, A. Yamada, T. Nakaoka, T. Kambe and N. Ishiura, "Architecture Evaluation Based on the Datapath Structure and Parallel Constraint," *IEICE Trans. Fundamentals of Electronics, Communications and Computer Sciences*, vol. E97-A, no. 10 (Oct. 1997).

[Yos97] Y. Yoshida B. Y. Song, H. Okuhata, T. Onoye, and I. Shirakawa: "Low-Power Consumption Architecture for Embedded Processor," in *Proc. 2nd International Conference on ASIC*, pp. 77–80 (Oct. 1996).