

Architectural Partitioning of Control Memory for Application Specific Programmable Processors

Wei Zhao and Christos A. Papachristou[†]

Department of Computer Engineering
Case Western Reserve University
Cleveland, OH 44106

Abstract

Because of programmability of Application Specific Programmable Processors (ASPPs), microcode-based control is effectively used to drive ASPP datapaths for different applications. In ASPPs, each application needs a separate microprogram resulting in large microcode memory. This paper proposes a distributed microcode memory model in which only distinct microcodes are stored in each separate memory module to save memory area. A hierarchical clustering approach is also proposed for the design of this distributed microcode memory. Experimental results indicate this approach is especially well suited for ASPP microcode memory design because of the existence of repetitive microcodes across multiple behaviors.

1 Introduction

1.1 Motivation

An Application Specific Programmable Processor (ASPP) is a programmable architecture which can be tuned to a number of different DSP applications[1]. We have developed an approach to employ on multiple behaviors[2]. By exploiting the similarities among the multiple behaviors, our approach can get an optimal datapath for all behaviors. As to ASPP's controller design, although there are many ways to implement the controller for ASICs, microcode seems to be the best candidate for ASPP's controller because programmability is required.

In most ASIC designs, "Hardwired" FSM is used to implement the controller; however, this can not meet ASPP's requirement. Once FSM control is decided, it can not be changed. In practice, some designers still use "hardwired" controllers to drive their programmable processor with a separate controller for each application. However, their applications are well defined before design and the number of applications is very limited, for example, just two. For this limited case, finite state machines(FSM) are more area efficient than microcode. However, these kind of processors are not real programmable processors.

In ASPPs, there are several microcode programs in the control memory, one for each application. Although sometimes these microprograms can be downloaded to microcode memory at run time, in most situations, switching from one application to another must be done in real time, thus this kind of program loading becomes prohibitive. This requires that all microprograms be in control memory leading to a very large memory. In addition, ASPP's microcode memory can not be off-chip, in that case memory is not a problem. The reason is ASPPs are high speed processors, thus off-chip microcode memory can not provide instructions in a timely fashion to sustain high speed computation. Also the datapath of ASPP is relatively more complex than that in ASICs, so the microcodes for ASPPs usually are much wider than that of ASICs, and this fact would increase the number of pins if the microcode memory would be off-chip. Thus it is

vital for ASPP designers to minimize the on-chip microcode memory area.

The similarity among all input behaviors not only leads us to a common datapath for all algorithms[2], but also provides a lot of similarities in control behaviors. For example, multiply-add is a common operation chain in many DSP applications; the control signals to this chain can be the same if there is a same binding for these multiply-add nodes. Apart from this explicit similarity, some partial similarity may exist among all or part of the input behaviors. Our objective is to expose and to make good use of these similarities leading to simple control unit design. Our work is motivated by these observations.

1.2 Our Work

In this paper, we propose a distributed microcode memory model in which only distinct microcodes are stored in each separate memory module to save area. To find out these commonly shared microcodes, we use a hierarchical clustering approach to merge those microcode fields that could lead to maximal decrease in total microcode memory in a bottom-up fashion. We use a closeness metric between two microcode fields to guide their merging.

1.3 Related Work

There has been a lot of research on microcode control unit design for ASICs. An efficient microcode compiler was developed in *CATHEDRAL II* [3] with emphasis on microprogram scheduling and memory allocation. Retargetable compiler techniques were studied in [4] and [5] for reconfigurable microarchitectures. Different microcode generation and optimization techniques are discussed in [6] [7]. As controller becomes more and more complex, some researchers use partitioning to make the design of controller more effective. A novel partition scheme was discussed in [8] [9] for the reduced area design of PLA-based control tables and PLA-based microcode. In [10], they propose a partitioning method for FSM implementation. By grouping the control sequences into classes, the method reduces the number of minterms and inputs, thus leads to a reduction in μ -controller area. [11] presents a new efficient algorithm to reduce the the width of microcode to save more microcode memory area. A detailed survey of microcode width minimization can also be found in [12]. The basic idea on microcode width minimization is to construct a compatibility graph of microoperations in each microinstruction and then to use various graph partitioning schemes to determine which microoperations are compatible. Encoding these compatible microoperations will lead to minimization of the width of microcode. Although various methods are different in their time and minimization efficiency, they all need additional logic to decode the encoded microcode. This may not be a serious problem in ASIC design; although there is a price to pay for this minimization in terms of increased gate delay. However, for ASPPs, additional decoding logic means more restriction to programmability. Even small changes to the any of the multiple behavioral descriptions require a complete redesign of the decoding logic and whole microcode memory.

We have seen some successful applications of hierarchical clustering in high-level synthesis [13]. Closeness of two operations in a control/data flow graph(CDFG) is defined in

[†]Support for this work has come from OAI Contract 94-1-015 and SRC Contract 94-DJ-527.

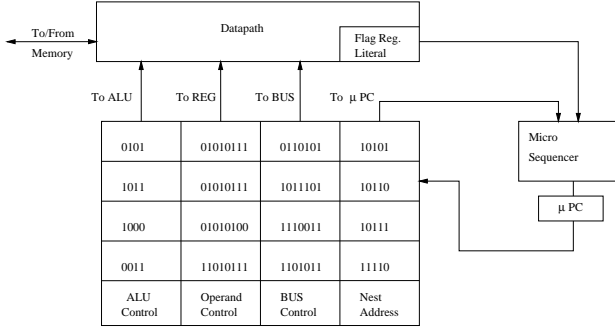


Figure 1: Microprogrammed Controller Model

BUD [14] [15]. By merging of operations which have small distance, a cluster tree can be constructed and then used to guide the unit selection in scheduling and binding. In APARTY [16] [17], different closeness definitions are used to cluster nodes in different stages. Different cutting lines give different styles of partition according to different design considerations. Although these works are very successful, they focus on the datapath design. In our work, we have found that hierarchical clustering is very useful not only in datapath design, but also in controller design.

2 Microprogram Controller Model

Microprogrammed controllers are discussed in detail in [18] [19]. Here we focus on the most standard and conventional model, which is built mainly around microcode memory. We base our discussion on this model because of its good programmability and simplicity. We show the diagram of this module in Figure 1.

In this model, the control signals for the datapath are saved in microcode memory in the form of microcodes. This microcode memory can be implemented in a RAM, so that by changing the microcode the whole architecture can be reprogrammed. A microsequencer can also be attached to produce the next address field as explained in [19] [20]. Here we focus just on the design of microcode memory, so we omit the details of the μ -sequencer in Fig. 1.

In microcode memory, the control signals are organized into time steps. One horizontal line of microcode represents the control signals at a time step and is called a microword. This format is very good to organize control, but from another point of view, we need to store redundant information which takes more memory area. To optimize the area cost, an approach is to store the distinct control signals only once. For a very long horizontal microcode, it is very difficult to find two identical microwords for this kind of compaction. Suppose each control bit has equal chance to be “1” or “0”. The probability of two microwords of width w being identical is $(\frac{1}{2})^w$, so it is very small. Although in practice the chance of certain sections of microcodes to be identical is higher than this, it is still too small to get any meaningful compaction. However, for ASPP applications, it is very likely that repetitive control signals will appear across all the microcodes. The reasons for us to believe this are:

1. DSP applications are computation intensive and some computation patterns occur frequently in many applications. Examples include multiplication/accumulation, etc.
2. All the applications of ASPP will share a common datapath, some datapath components are repeatedly used in the same way in many places across all applications.

Although it is difficult to provide a quantitative analysis of repetitive microcodes in multiple behaviors, we are convinced that the more similarity across multiple behaviors, the more repetitive microcode we can have. Figure 2 shows

Time Steps	Algorithms		
	DCT (-,+,*)	FFT (-,+,*)	IIR (-,+,*)
1	110	001	001
2	110	001	001
3	110	001	011
4	110	101	011
5	111	110	011
6	111	110	011
7	111	010	110
8	111		101
9	111		100
10	111		
11	011		
12	111		
13	111		
14	111		
15	011		
16	011		
Distinct	3	4	5
Subtotal	16	7	9
Distinct	7		
Total	32		

Figure 2: Repetitive Microcodes in A Real Example

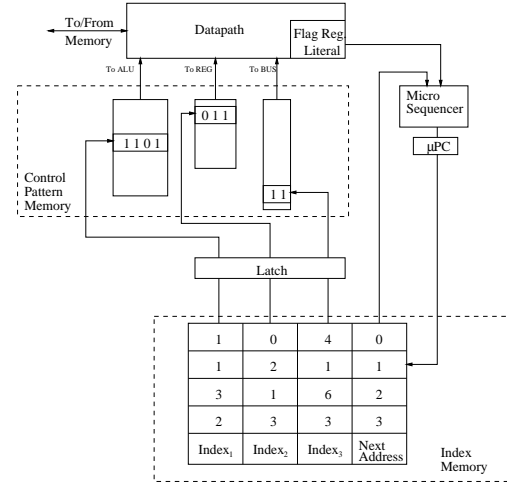


Figure 3: Distributed Microprogrammed Control Model

how the repetitive microcodes exist in a real example. The three algorithms are in the second case in our experiments.

Clearly, from our above observation, another way to increase the chances of repetitive microcodes is to reduce w . And the only way to do that is to partition the whole microcode memory horizontally across the fields into several small pieces resulting in a distributed microcode memory.

2.1 Distributed Microcode Memory

Figure 3 shows the distributed microcode memory model. The microcode memory is divided into two parts, one is labeled *index memory* and another is labeled *control pattern memory*; each separate memory in control pattern memory is labeled *microcode memory module*; the microcode in each microcode memory module is labeled *control patterns* hereafter. The index memory has a next address field and some additional index fields, each for one microcode memory module. The conventional microsequencer is attached to the index memory as usual. Actually, this memory works just as a conventional microcode memory as illustrated in Fig. 3. The only difference is instead of providing control signals to datapath, it just provides indices to the microcode memory modules where the control signals are stored. Each microcode memory module as shown in Fig. 4 on the right side represents a “slice” of the original control memory, it receives the indices provided by the index memory and then provides the control signals to the datapath. Two-level control has been used earlier in QM-1 [21] and M68000 but in a different way. This distributed microcode memory has a two-stage organization and operates in pipeline fashion us-

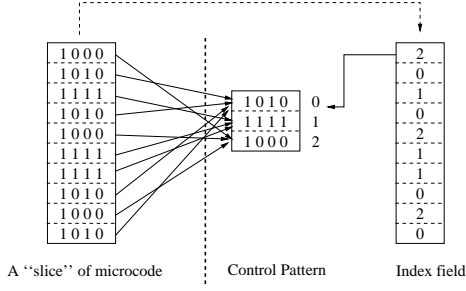


Figure 4: Mapping Relationship between Two Control Memory Models

ing latches between the two memory stages to keep the speed of the control unit with that of datapath, as shown in Fig. 3. While the control pattern memory provides the control signals to the datapath, the index memory is working on the next instruction address. This mechanism can makeup the speed loss in the two level memory design.

In each microcode memory module, only the distinct control patterns are stored. Fig. 4 shows the mapping relationship from conventional control memory to distributed control memory. The microcode memory module can save area by saving control patterns, while the control sequence is maintained in the index memory.

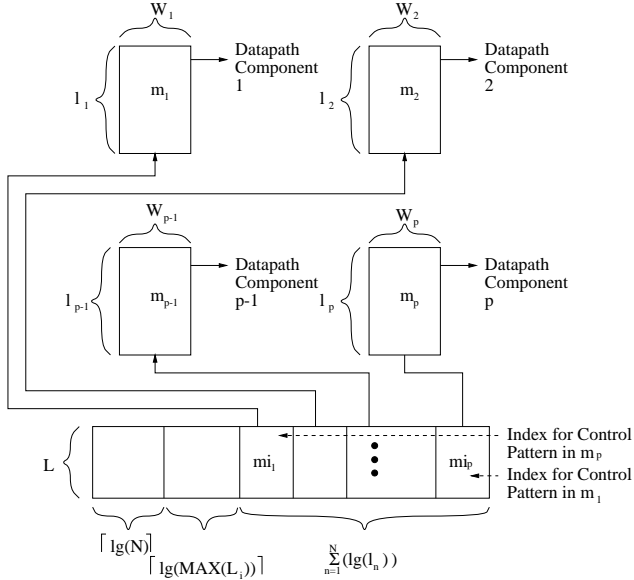


Figure 5: Analysis of Distributed Microprogrammed Control Model

2.2 Analysis of Distributed Microcode Model

In Figure 5 we illustrate the relevant parameters in the distributed microcode memory model. Here we can see that the cost for the compaction in each microcode memory module is that we must add some additional index fields in the index memory. The larger the number of microcode memory modules we have, the more compaction we might get in each microcode memory module, but we will also have more index fields in the index memory. Thus there is a tradeoff between these two factors. Sometimes, the additional index fields can forfeit the savings in each microcode memory module. The following analysis indicates their relationship.

Suppose :

- N the # of input applications
- L_n the # of microcodes in n th application
- L total # of microcodes of all the N input applications
- W total width of microcodes that go to the datapath
- w_p width of microcodes in p th microcode memory module
- l_p length of microcodes in p th microcode memory module
- m_p # of memory bits in p th microcode memory module
- c_p the compaction ratio, $c_p = \frac{l_p}{L}$
- mi_p # of memory bits in p th microcode index field
- P the total # of microcode memory modules
- M the total # of memory bits of the whole original microcode memory
- M_p the # of memory bits of the p th microcode memory module
- M_{index} the total # of memory bits of the additional index fields
- $M_{partition}$ the total # of memory bits after partitioning
- $M_{address}$ the total # of memory bits of next address field

Based on Figure 5 we have the following,

$$L = \sum_{n=1}^N L_n; \quad W = \sum_{p=1}^P w_p; \quad m_p = l_p * w_p; \quad mi_p = [\lg^*(l_p)] * L;$$

$$M_{address} = ([\lg(N)] + [\lg(MAX(L_n))]) * L$$

So the total memory bits before partitioning :

$$M = M_{address} + W * L \quad (1)$$

After partitioning, each microcode memory module for control patterns:

$$M_p = w_p * l_p = w_p * c_p * L$$

According to our previous analysis, after partitioning, we will need some additional fields to hold the indices to the control patterns. The number of memory bits required for these additional fields can be calculated as follows:

$$M_{index} = \sum_{p=1}^P mi_p = \sum_{p=1}^P ([\lg(c_p * L)] * L)$$

Therefore, the total number of memory bits after partitioning is:

$$M_{partition} = M_{address} + M_{index} + \sum_{p=1}^P M_p \quad (2)$$

Our goal is to use partitioning to reduce area, so we would like to have $(2) < (1)$. We compare (1) and (2), delete L on both sides and substitute c_p on left side by $c_{min} = MIN(c_p)$. Then we get the following:

$$P * [\lg(L * c_{min})] + W * c_{min} < W \quad (3)$$

We can further simplify (3) by substituting $1 - c_{min}$ by another coefficient k as follows :

$$P < k * \frac{W}{[\lg(L) + \lg(c_{min})]} \quad (4)$$

This tells us that there is an upper bound for the number of partitions. The upper bound is decided by three parameters that are W , L and c_{min} . If there are more repetitive microcodes in individual memory module, we can get a smaller c_{min} , thus a bigger k ; this means, when W and L

* $\lg(x) = \log_2(x)$ hereafter

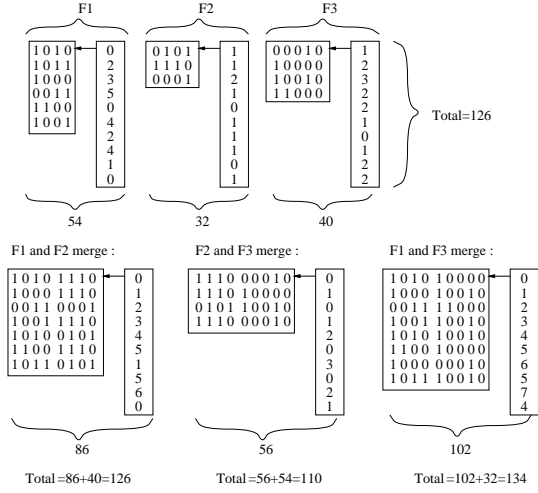


Figure 6: An Example : Merging of Microcode Memory Modules

are fixed, the more repetitive microcode in memory module, the bigger range for a possible area saving partition.

This distributed microcode memory model is not suitable for conventional microprocessors. The conventional microprocessor is built around one *ALU* with a relatively simple datapath. This makes the width of the microcode W not very big. As the functions performed by conventional microprocessors are much more complex than the functions performed by ASICs and ASPPs, these functions usually have long sequence of microcodes, meaning a very big L . The smaller W , and the relatively large L and c_{min} , make it impossible to use partitioning to save some area.

3 Microcode Field Closeness

Before introducing the concepts, let us first study an example shown in Fig. 6. Suppose we have three microcode fields whose microcode memory bits are 54, 32 and 40 for field $F1$, $F2$, and $F3$, respectively, as shown in Fig. 6. The width of $F1$ is 4, the length is 6. So the bits in memory module $F1$ are 24. In the index field of $F1$, we have the width $\lceil \lg(5) \rceil = 3$, and length is 10, so the bits in index field are 30 [†]. Thus we have 54 memory bits in $F1$. The total number of memory bits in $F1$, $F2$ and $F3$ is 126. We want to find which two of the three should be merged to reduce the total memory area. We try to merge any two of them as shown in Fig. 6. Comparing with 126 which is the number of the original memory bits before merging, we know that by merging fields 2 and 3, we can get a smaller microcode memory.

The goal of the memory closeness metric is to find out which two microcode fields will lead to smallest total number of microcode memory bits if they were merged together. Fig. 7 shows the merging process.

Suppose :

- M_i^D total # of memory bits of the i th microcode field that goes to datapath
- M_i^I total # of memory bits of the index field of the i th microcode field
- M_{ij}^D total # of memory bits of microcode field which is merged from microcode fields i and j
- M_{ij}^I total # of memory bits of the index field of microcode field ij

Based on the above definition and Figure 7, we have the

[†]Recall that index field is in another memory stage, please refer to Fig. 3.

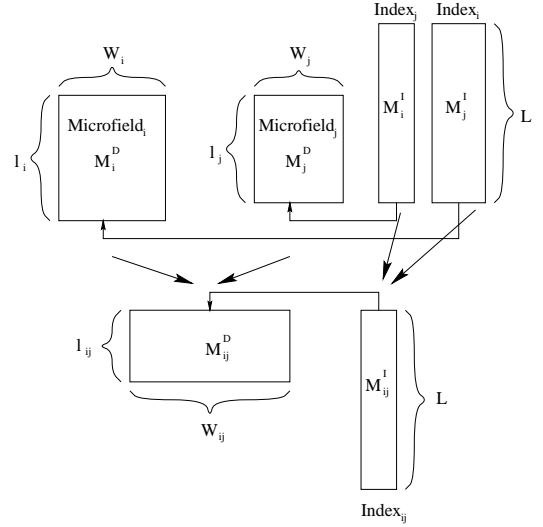


Figure 7: The Merge Process of Two Microcode Memory Modules

following for the size of total memory[†] before merging :

$$M = \sum_{p=1, p \neq i, j}^P (M_p^D + M_p^I) + (M_i^D + M_i^I) + (M_j^D + M_j^I)$$

After we merge microcode fields i and j , we get the following total memory bits:

$$M_{merge} = M - (M_i^D + M_i^I) - (M_j^D + M_j^I) + (M_{ij}^D + M_{ij}^I)$$

We know that after a merge, the total number of memory bits will change as indicated above. We simply define the closeness of two microcode fields as:

$$\chi(i, j) = -(M_i^D + M_i^I) - (M_j^D + M_j^I) + (M_{ij}^D + M_{ij}^I) \quad (5)$$

We allow the distance to be negative for the ease of understanding and computation. By merging two microcode fields, this definition can guarantee the greatest decrease in total memory area.

According to this definition, the distances between the fields in the example in Figure 6 are as follows :

$$\chi(1, 2) = 0; \quad \chi(2, 3) = -16; \quad \chi(1, 3) = 8;$$

We can further substitute Equation 5 by some parameters tagged on Figure 7. We then get the following :

$$\chi(i, j) = w_i * (l_j - l_i) + w_j * (l_i - l_j) + L * \lceil \lg(\frac{l_{ij}}{l_i * l_j}) \rceil \quad (6)$$

As ' l ' means the number of distinct microcodes, so the distinct number of microcodes in merged field w_{ij} will have the following bounds:

$$\max(l_i, l_j) \leq l_{ij} \leq l_i + l_j$$

In Equation 6, the first two terms will always be positive, and the third term may be positive, zero, or negative, because l_{ij} may be greater, less than or equal to $l_i * l_j$. Whether the value of $\chi(i, j)$ is positive or negative, depends largely on the third term in Equation 6. However the third term is not decisive; the first two terms must still be considered. This is why we can not simplify the $\chi(i, j)$ function further.

[†]In our later discussion, we exclude the address field in microcode which is $M_{address}$ because it remains constant during the partition process.

4 Partition Algorithm

First the control signals that go to the datapath are divided into basic fields. Each field controls one datapath component. These fields are used as the initial partitions. Then, they are input to our partitioner. The partitioner will iteratively merge those two fields that have the smallest distance based on the closeness metric between two fields, as defined in Equation 5. Finally the partitioner will merge all these fields into a unified one which is equivalent to the conventional microcode memory.

In each step, the partitioner decreases the number of partitions by 1. The partitioning process travels a path such that at each move, a locally optimal solution is picked. Many implementations of this kind of algorithm exist according to a survey in [13]. The time complexity is linear in the number of microcode fields in the original microcodes.

The following is a pseudocode description of the algorithm.

```

PARTITION(microcode[f], f) /* f is the total
                             number of microcode fields */
1 while ( f > 1 ) do
2   for i ← 1 to f do
3     for j ← 0 to f do
4       calculate closeness[i][j]
5       if (closeness[i][j] < temp)
6         m ← i; n ← j;
7         temp ← closeness[i][j];
8     endfor
9   endfor
10  merge_microcode_field(m,n);
11  f ← f - 1
12  calculate total microcode memory bits;
13  save the partition result;
14 endwhile;

```

5 Experiments

Our partitioner is written in the C programming language and is linked to SYNTTEST [22]. First, the algorithms are fed to a transformation process [2] for optimization[§], and then they are input to SYNTTEST[¶] for scheduling and allocation. Control synthesis is done by SYNTTEST incrementally with allocation. The final microcodes of each algorithm are then fed to our partitioner for partitioning. The partitioner finds the best partition and then generates VHDL code for each microcode memory. The VHDL descriptions are then fed to COMPASS [23] for layout synthesis.

We use our partitioner for three ASPP design cases. The first one was used as a benchmark during the development of the partitioner. The second and the third ones are real design cases whose datapath synthesis was reported in another paper [2].

We list the relative information of the three cases in Table 1. In this table, the third column lists the types and number of computation nodes that each behavior has. The fourth column lists the final datapath generated by SYNTTEST. The last three columns list the width, length and number of fields of the microcodes.

We illustrate the partitioning process for Case 1, 2 and 3 in Fig. 8. Each curve on the diagram represents the partitioning process in one case. One point on the curve corresponds to one partition result. The point's X-coordinate indicates the number of partitions. Its Y-coordinate indicates the total area in number of transistors.

In each curve, the partition process starts from the right-most point, that is, the largest number in X-axis and finally terminates at "1" on the X-axis which corresponds to merging all the microcode fields together. As our analy-

Case Number	Algorithm	Behavioral Description	DataPath	Microcode		
				Width	Length	Fields
Case 1	#1	* 4, + 5, - 4	*1,+1,-1	43	6	29
	#2	* 3, + 5, - 2	*1,+1,-1	43	5	29
	#3	* 2, + 5, - 4	*1,+1,-1	43	5	29
Case 2	DCT	*13,+16,-13	*1,+1,-1	59	16	42
	IIR (5-3order)	*8,+4,-3	*1,+1,-1	59	9	42
	FFT (radix-2)	*4,+3,-3	*1,+1,-1	59	7	42
Case 3	Bandpass	*13,+9,-6,0	*2,+1,-1/1	56	10	39
	Biquad	*6,+9,-5,0	*2,+1,-1/1	56	10	39
	FFT (radix-2)	*4,+3,-3,0	*2,+1,-1/1	56	5	39
	Sobel	*9,+11,-5,4	*2,+1,-1/1	56	20	39
	Hough	*3,+2,-2,0	*2,+1,-1/1	56	5	39
	Robert	*11,+7,-2,4	*2,+1,-1/1	56	17	39

Table 1 : The Algorithms and Their Microcodes in Case 1, 2, 3

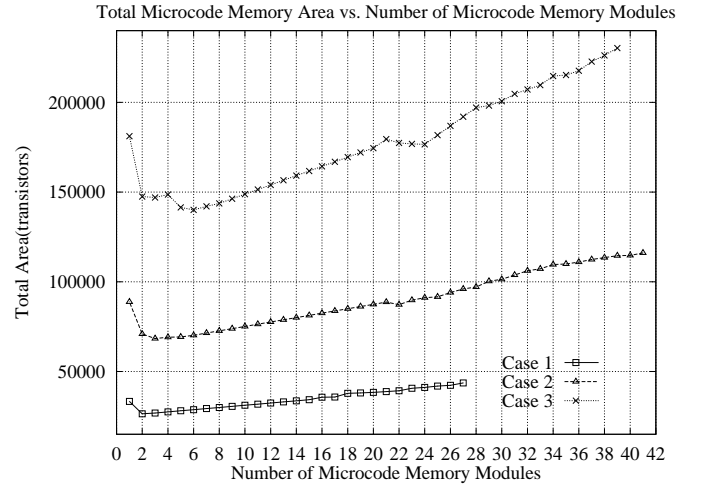


Figure 8: Partitioning Process in Case 1, 2 and 3

sis predicts^{||}, the best solution usually happens at a very small number of partitions. A large number of partitions will cause too many additional index fields, which will forfeit the savings brought by compaction in each microcode memory module. We can easily find the best partitions in the three cases. They are all much smaller in area compared with the left-most point in each curve, respectively, which represents the conventional microcode memory. We summarize the results in Table 2^{**}.

We also try to use our partitioner to optimize the microcode memory for each application algorithm in the three cases separately. The results are summarized in Table 3. In this experiment, the microcodes of each algorithm are obtained by separately scheduling and binding each algorithm according to each one's time constraints. We can see that the saving by this method varies from algorithm to algorithm; however, generally speaking, the results are not as good as those in ASPP cases. We believe the reason for this is there are not enough repetitive microcodes within a single algorithm. In ASPP cases, not only may there exist enough repetitive microcodes, but also there are more data-

[§]To get a minimum common datapath

[¶]Or any other high level synthesis system

^{||}Please refer to Equation 4

^{**}The CPU time in the last column is obtained on SUN SPARC-II with 52 MB memory.

	Central Microcode Memory (transistors)	Partitioned Microcode Memory (transistors)	Number of Partitions	Savings	CPU Time (s)
Case 1	32067	25167	2	21.52%	7
Case 2	88802	68335	3	23.05%	76
Case 3	181190	140062	6	22.70%	276

Table 2 : Summary of Experimental Results

Algorithm	Central Microcode Memory (transistors)	Partitioned Microcode Memory (transistors)	Number of Partitions	Savings	Microcode	
					Width	Length
#1	8093	8093	1	0%	29	6
#2	6744	6744	1	0%	29	5
#3	6744	6744	1	0%	29	5
DCT	33301	33301	1	0%	43	16
IIR (5-3order)	7284	6898	2	5.30%	22	9
FFT (radix-2)	8671	8671	1	0%	17	7
Bandpass	25052	20619	3	17.70%	56	10
Biquad	18115	16418	2	9.37%	40	10
FFT (radix-2)	7284	6898	2	5.30%	22	5
Sobel	37771	34374	9	8.99%	38	20
Hough	5202	4007	2	22.97%	20	5
Robert	32105	26555	4	17.29%	40	17

Table 3 : Partitioning Results on Each Algorithm in Case 1, 2 and 3

path components and the width of microcode is bigger than that in a single algorithm. All these factors contribute to the better savings in ASPP cases. This result shows that our partitioner is especially well suited for ASPP microcode memory designs.

6 Conclusion and Future Work

In this paper, we propose a distributed microcode memory model for ASPPs. Based on this model, we propose a hierarchical partitioning algorithm which can maximally exploit the repetitive control patterns in certain sections of the conventional microcodes. This leads to a reduction in total microcode memory size.

In the future, we will pursue the following aspects:

1. Simulate the two level control memory model in COM-PASS to ensure that speed loss will not be a problem.
2. Continue to study to programmability issue with this model, a preliminary result can be found in [24]
3. As our work is based on the assumption of repetitive code in multiple behaviors, we will extend our work to scheduling and allocation phases to create more repetitive control signals.

References

- [1] P. Paulin, "DSP Design Tool Requirements for Nineties: An Industrial Perspective", Sixth Intl. Workshop on High-Level Synthesis, Laguna Niguel, CA, Nov. 1992.
- [2] W. Zhao, C. Papachristou, "An Evolution Programming Approach on Multiple Behaviors for The Design of Application Specific Programmable Processors", Technical Report, TR-CES-95-06, Case Western Reserve University.
- [3] G. Goossens, J. Rabaey J. Vandewalle, H. De Man, "An Efficient Microcode-compiler for Custom DSP-processors", Proc. of IEEE/ACM ICCAD, pp.24-27, Nov, 1987.
- [4] P. Marwedel, "A New Synthesis Algorithm for the MIMOLA Software System", Proc. of 23rd Design Automation Conference, pp. 271-277, 1986
- [5] P. Marwedel, "A Retargetable Compiler for a High-Level Microprogramming Language", Proc. of 17th Annual Microprogramming Workshop(MICRO-17), pp. 267-276, 1984.
- [6] C. Liem, T. May, P. Paulin, "Register Assignment through Resource Classification for ASIP Microcode Generation", Proc. of IEEE/ACM ICCAD, pp.397-402, Nov, 1994.
- [7] S. Lin, C. Hwang, Y. Hsu, "Efficient Microcode Arrangement and Controller Synthesis for Application Specific Integrated Circuits", Proc. of IEEE/ACM ICCAD, pp.38-41, Nov, 1991.
- [8] C. Papachristou and A. Pandya, "A Design Scheme for PLA-based Control Tables with Reduced Area and Time-Delay Cost", IEEE Trans. on Computer-Aided Design of Integrated Circuits & Systems, Vol. CAD-8, No. 5, pp. 453-472, May 1990.
- [9] C. Papachristou and J. Reuter, "Microassembly and Area Reduction Techniques for PLA Microcode", 17th IEEE-ACM Microprogramming Workshop, pp. 86-94, November 1984.
- [10] G. Tarroux, B. Rouzeyre, G. Sagnes, "Optimization of Microcontrollers by Partitioning", Proc. of IEEE/ACM ICCAD, pp.368-373, Nov. 1991.
- [11] R. Puri, J. Gu, "An Efficient Algorithm for Microcode Length Minimization", Proc. of 29th Design Automation Conference, pp.651-656, 1992.
- [12] R. Puri, J. Gu, "Microword Length Minimization in Microprogrammed Controller Synthesis", IEEE Trans. on Computer-aided Design of Integrated Circuits and System, Vol. 12, No.10, pp.1449-1457, October 1993.
- [13] D. Gajski, N. Dutt, A. Wu, S. Lin, "High-Level Synthesis: introduction to chip and system design", Kluwer Academic Publishers, 1992.
- [14] M. McFarland, "Using Bottom-Up Design Techniques in the synthesis of Digital Hardware from Abstract Behavioral Descriptions", Proc. of 23rd Design Automation Conference, pp. 474-480, 1986.
- [15] M. McFarland, T. Kowalski, "Incorporating Bottom-Up Design into Hardware Synthesis", IEEE Trans. on Computer-aided Design of Integrated Circuits and System, Vol. 9, No.9, pp.474-480, September 1990.
- [16] E. Lagnese, D. Thomas, "Architectural Partitioning for System Level Synthesis of Integrated Circuits", IEEE Trans. on Computer-aided Design of Integrated Circuits and System, Vol. 10, No.7, pp.847-860, July 1991.
- [17] E. Lagnese, D. Thomas, "Architectural Partitioning for System Level Design", Proc. of 26rd Design Automation Conference, pp. 62-67, 1986.
- [18] D. Patterson, J. Hennessy, "Computer Organization and Design: the hardware/software interface", Morgan Kaufmann Publishers, Inc., 1994.
- [19] N. Tredennick, "Microprocessor Logic Design", Murray Printing Company, 1987.
- [20] C. Mead, L. Conway, "Introduction to VLSI systems", Addison-Wesley Publishing Company, 1980.
- [21] A. Salisbury, "Microprogrammable Computer Architectures", Elsevier North-Holland, Inc., 1977.
- [22] H. Harmanani, C. Papachristou, S. Chiu and M. Nourani, "SYN-TEST : An Environment for System-Level Design for Test", Proc. European Design Automation Conference (EURO-DAC), Sept. 1992.
- [23] COMPASS Design Automation, Inc., "VHDL for the ASIC Synthesizer User Guide", COMPASS Design Automation, Inc., Aug. 1994.
- [24] W. Zhao, C. Papachristou, "Architectural Partitioning of Control Memory for Application Specific Programmable Processors", Technical Report, TR-CES-95, Case Western Reserve University.