Library-based Mapping of Application to Reconfigurable Array Architecture

Kyuseung Han and Kiyoung Choi

Abstract—**Reconfigurable array architecture is recently attracting much attention. It is a flexible hardware architecture, which can dynamically change its configuration to execute various functions while maintainning high performance. However, pursuing flexibility and performance at the same time leads to complexity, thereby makes the mapping of applications a difficult process. There have been attempts to use compiler or high level synthesis techniques to solve the problem. In this paper, we propose yet another method, which uses libraries for the mapping to provide an abstraction of the internal structure and at the same time to reduce the development time and efforts through the automated process. We have selected a JPEG decoder as an example to apply the proposed method. As a result, we obtained about 20% less performance compared to manual mapping but development time is dramatically reduced to less than 1%.**

Index Terms—**Reconfigurable array, architecture, mapping, parameterized library, multimedia**

I. INTRODUCTION

As the amount of computation required in an application increases, it becomes more and more difficult to satisfy the performance requirement of an embedded system with just software running on a processor. Therefore, many embedded systems or system-on-chips (SoCs) are equipped with one or more dedicated hardware IPs. Such hardware IPs may provide sufficient performance but lack flexibility and so they should be redesigned from the scratch even when the required functionality changes slightly. Moreover, since hardware can hardly be fixed after fabrication, it needs much more time and effort put into the verification of the design before fabrication. Even further, considering that it is a trend to support various multimedia applications such as music or video codec in a cellular phone or a portable multimedia player, developing an IP for each application takes too much cost, development time, and power consumption. One way to solve such a problem is to use a reconfigurable array architecture (RAA), which provides hardware-like performance through an array processing as well as software-like flexibility through reconfigureability.

Instead of developing a new IP for a new application, to implement an application on an RAA, one can just reconfigure the existing RAA such that the RAA provides the necessary functionality of the application. For this, internal code for the reconfiguration should be generated from the application, which is called a mapping process. However, the mapping of an application onto the RAA is more difficult than a simple compilation, since the hardware architecture is parallel -- an array of processing elements (PEs) -- and moreover the interconnection between the PEs are not fixed but they should also be reconfigured. The difficulty is the cost that one has to pay for achieving both flexibility and performance at the same time.

There have been previous researches to address this issue. One approach tries to take an RAA as a part of a VLIW processor and extend a compiler to include the mapping process [1]. However, since the architecture is not just a processor with fixed hardware, the optimization process is not easy and takes long time. Some other approaches combine compiler techniques with various techniques for the mapping of operations onto the array

Manuscript received Aug. 23, 2009; revised Nov. 1, 2009.
Design Automation Lab, Seoul National University
E-mail : darprin@poppy.snu.ac.kr

of PEs [2-4]. Such approaches typically take multiple steps of optimization from code optimization down to physical layout on the array, which may result in a less optimized mapping. Yet another approach adopts high level synthesis techniques to schedule the operations and bind them to the PEs in an RAA [5,6]. This approach is rather efficient but still takes time and may result in a less optimized mapping.

In this paper, we propose a fast and effective approach to mapping of an application to an RAA. First, in Section II, we describe the target architecture that we consider for this research on mapping. In Section III, we present our library-based mapping approach. In Section IV, we present experimental results to demonstrate the effectiveness of the proposed approach. We conclude this paper in Section V with some remarks on the future work.

II. TARGET ARCHITECTURE

In this work, we use FloRA [7], an RAA developed by Design Automation Lab, Seoul National University (Fig. 1). FloRA roughly consists of three parts: configuration cache, array of PEs, and frame buffer. Configuration cache contains multiple layers of configuration code which determines the instructions of the PEs as well as the interconnection. The configuration of the entire array can be changed within one clock cycle by fetching new configuration code from a different layer of the cache. The capability of changing the configuration dynamically (during runtime) is what we call reconfigurability. Frame buffer is an internal memory used for communicating with other blocks (e.g. processor core or main memory) of the platform. Array of PEs is the part that performs actual computation according to the configuration code. For the computation, the PEs fetch data from the frame buffer, process them, and write the results back into the frame buffer.

The previous design of FloRA provides a very simple but restricted communication mechanism between the frame buffer and the PEs such that a PE can access to only a fixed location in the frame buffer. This makes our library-based mapping approach very complicated or sometimes impossible to apply. Therefore, for a little more flexible communication mechanism, we add address generation units that can do striding. The cost is not high since they require only three 8-bit adders.

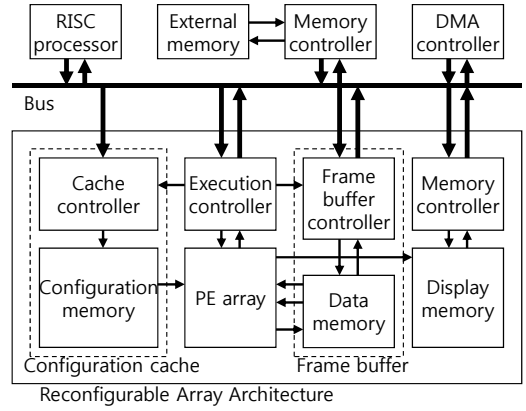

Fig. 1. Target architecture, FloRA.

III. LIBRARY-BASED MAPPING

1. Overview

In order to run an application kernel on FloRA for acceleration, we should map the kernel code on the FloRA architecture. It requires generating configuration code for the configuration of FloRA as well as control code that runs on the processor core to control the activation of the configuration code and the data communication through the frame buffer. Conventionally they are generated manually, making the process cumbersome, time-consuming, and error-prone. In this work, we try to solve the problem through a library. The library is a set of *library functions*, each of which is used to generate a *target function* that performs a basic operation such as vector operation, matrix operation, or other frequently-used operation. A target function contains both configuration code and control code needed to perform the corresponding operation on FloRA. Instead of manually writing code that accelerates the execution of a given kernel, we just decide which library function (or which combination of library functions) should be selected to generate an appropriate target function (or combination of target functions), and then replace the kernel with a call to the target function (calls to the target functions).

Fig. 2 shows the overall flow of library-based mapping of an application kernel onto FloRA. First, we analyze the given application software to extract compute-intensive parts which we call kernels. Among the kernels, we select ones that can be replaced by target functions generated by the proposed approach. Then we write *function-generating code* that calls the library functions

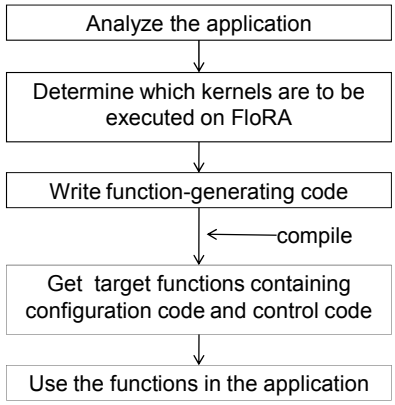

Fig. 2. Overall flow of library-based mapping.

with appropriate parameter values. When the function-generating code is compiled, the corresponding target functions are generated in the C language so that they can be used directly in the application code.

2. Parameterized library

The control code controls the computation on FloRA in three steps. First, it copies the input data in the main memory to the frame buffer. In the second step, it makes FloRA compute with the input data and write the output back to the frame buffer. Finally, it copies the output result back to the main memory.

We implement the whole process with a target function that contains the configuration code as well as the control code. The function requires only the location of input and output in the main memory as an argument, so provides a perfect abstraction of the internal architecture. The user (or programmer) does not need to know the details of FloRA.

The user should generate the target function before the use. He/she can generate it by writing code that calls library functions with the parameters set properly. But why not use the library function directly without generating the target function? Let's consider, for example, matrix multiplication operation. If a library function for matrix multiplication is used directly in the application but it can handle only a fixed matrix size like (8x8) x(8x8), then it may not be flexible enough to be used for arbitrary sized matrix multiplication such as (15x15)x (15,15) or more complex one like (3x5)x(5x6). Surely, it is almost impossible to have one library function that fits all sizes, and so we decide to take two phase approach. The user first writes function-generating code, which contains function calls to the library functions that have necessary configuration code and control code. Then the function-generating code is compiled to generate the actual target functions to be used in the application. In the example of matrix multiplication, once the user determines the size of the matrices, he/she write a code that calls the library function for matrix multiplication. Then the code is compiled to generate the target function to be called in the application software.

Following is the actual declaration of the library function for matrix multiplication.

genLibrary_Matrix_Mult(IN_TYPE, OUT_TYPE, SIZE_A, SIZE_B, SIZE_C, POLICY, MF_NAME);

Among the seven parameters, IN_TYPE and OUT_TYPE are respectively for the data types of input and output such as char, unsigned char, and short. Next three parameters determine the sizes of matrices and so the target function generated from this function will execute (SIZE_A x SIZE_B) x (SIZE_B x SIZE_C) matrix multiplication. POLICY can be used to decide whether the application prefers fast execution or less amount of configuration memory. MF_NAME just sets the name of the target function to be generated.

Each library function has templates of the control code and configuration code which are used to generate actual control code and configuration code to be included in the target function. Note that control code can vary according to the data types and sizes because they control the amount of data transferred between the frame buffer and the main memory. Configuration code is also changed by these parameters. Data types may affect some configuration code dealing communication between the frame buffer and PEs. Regarding the instructions to be executed in PEs, which is a part of the configuration code, the library function has a minimal set of instructions and expands it to the given data sizes by modifying or duplicating it.

There can be many other parameters besides the ones mentioned above. These features help generate target functions to meet the requirements of the user. However, the function generation efficiency heavily depends on the functionality and regularity. Moreover, some irregular parts of FloRA architecture make the function generation

very difficult. Some functions are generated in a very inefficient form, and some functions are impossible to generate. In these cases, we use fixed library functions which have parameters set to commonly used constants.

3. Library-based function generation

As mentioned early in this section, the user should write a function-generating code using the library functions given in the library. Fig. 3 shows an example of the function-generating code for the dequantization block and inverse discrete cosine transform (IDCT) block in the JPEG decoder. In this code, three library functions marked as ①, ②, and ③ are called with many parameters. Dequantization is separated into ① and ② because coefficients for luma and chroma are different. By compiling this code, one can generate the target functions as shown in the header file of Figure 4. By using these functions in the application code, the mapping process is completed.

When the function-generating code is compiled, all configuration code of FloRA as well as the control code are generated. During this process, similar parts among the mapping functions are shared so that the amount of memory used for the configuration is optimized.

Fig. 5 shows the function body of "deqaunt_Chroma", one of the target functions generated from the code in Fig. 3. There are four arguments, which provide information on where and how the input and output data are arranged in the main memory. The function body has five lines, ① through ⑤. In ①, the configuration cache of FloRA is loaded with the configuration code for the chroma dequnatization. Then in ②, the input data are copied from the main memory to the frame buffer. FloRA starts execution in ③, and the processor waits in ④ until FloRA finishes the computation. Then the results are transferred to the main memory in ⑤.

```
extern int dequant_coef_luma[8][8];
extern int dequant_coef_chroma[8][8];
void main()
{
genLibrary_Matrix_ConstScalarMult(
IO_TYPE_CHAR, IO_TYPE_SHORT,
8,8, dequant_coef_luma,
"deqaunt_Luma");                          … ①
genLibrary_Matrix_ConstScalarMult(
IO_TYPE_CHAR, IO_TYPE_SHORT,
8,8, dequant_coef_chroma,
"deqaunt_Chroma");                        … ②
genLibrary_InverseDCT(IO_TYPE_SHORT);     … ③
makeFloRAConfiguations ();
}
```

Fig. 3. Function-generating code using parameterized library functions.

```
void deqaunt_Luma(int srcAddrA, int saguA, int dstAddr,
int dagu);
void deqaunt_Chroma(int srcAddrA, int saguA, int dstAddr,
int dagu);
void FL_InvDct_Short(int srcAddrA, int saguA, int dstAddr,
int dagu);
```

Fig. 4. Declaration of the target functions generated from the code in Fig. 3.

```
void deqaunt_Chroma(int srcAddrA, int saguA,
int dstAddr, int dagu)
{
initConfigure_deqaunt_Chroma();                  … ①
mem2mem(srcAddrA, saguA, RC_FB_SET0_A,           … ②
AGU(2, 2, 1, 1), 16);
RAA_START(1, 3, 2, 3, 3, 0, 0, 0, 0, 0, 2);      … ③
raa_end();                                       … ④
mem2mem(RC_FB_SET0_C, AGU(4, 4, 1, 1),           … ⑤
dstAddr, dagu, 32);
}
```

Fig. 5. Function body of one of the generated target functions.

4. Part of code not covered by library

In general, the library does not cover the entire application code but covers only the kernels that are supported by the library functions. Therefore, after mapping the kernels onto FloRA, there can remain some kernels and glue code that are not covered by the library. If the system performance is already high enough, then the remaining part of the application code can be executed on the RISC processor. In this case, the designer does not need to worry about any side effect from the generated target functions running on FloRA, since they provide good encapsulation.

In the case that one needs to accelerate some of the remaining kernels to achieve even higher performance, he/she cannot help taking extra effort to make his/her own target functions. However, even in this case, the extra effort is minimized by utilizing the library development framework that we provide, which is beyond the scope of this paper. Once a new target

function is made, it can be added to the library for reuse by other applications.

IV. EXPERIMENTAL RESULTS

1. Library functions

In this subsection, we compare the quality of the configuration code generated for each of the basic library functions with the manually mapped one in term of performance and resource usage. The performance is measured by the number of cycles taken for FloRA to complete the operation, and the resource usage is measured by the amount of memory (in the unit of 4 bytes) required for caching the configuration code. As shown in Table 1, there is no difference between the proposed approach and the manually mapping. In general, manual mapping has a better result, but when only a small-sized function is considered, the proposed approach can also generate well optimized code.

Table 1. Comparison between manual mapping and library-based mapping

kernels	Number of data	performance (cycle)		Resource (memory in 4 bytes)	
		manual	library	manual	library
accumulate	8	4	4	4	4
	16	5	5	5	5
dot product	8	5	5	4	4
	16	15	15	6	6
transpose	8	7	7	8	8
	16	28	28	32	32
convolution	4	6	6	5	5
	8	6	6	6	6
matrix multiplication	4	47	47	25	25
	8	135	135	26	26

2. Application to JPEG decoder

We select JPEG decoder as an application and simulate this using SoC Designer 7.0. The platform consists of ARM7TDMI, SRAM, and FloRA, which are connected through an AMBA AHB bus. We adopt the flow introduced in section III.1 to map some functions of the JPEG decoder onto FloRA. By profiling the reference software of JPEG decoder, we find that the dequantization block and the IDCT block are compute-intensive so we decide to map them onto FLoRA. Dequantization can be replaced by matrix scalar multiplication, and IDCT can be replaced by 8x8 Chen DCT [8] library function. The function-generating code for these two library functions and the generated header file are shown in Fig. 3 and Fig. 4, respectively.

We show in Table 2 how much acceleration we can achieve for the JPEG decoder and compare it to the case of software only implementation and to the case of manual mapping onto FloRA. Compared to the software-only implementation, the library-based mapping provides more than twice the performance. If we consider the entire system, the performance improvement is about 23%. Compared to the manual mapping, the entire system performance is slower by 23%. It is because the communication overhead increases about 5 times. For the manual mapping, the outputs of dequantization are directly pipelined to the inputs of IDCT through the internal communication of FloRA. However, the current library functions do not support this, since they copy the outputs to the main memory before they are used by other functions. This is the main drawback of the proposed approach and should be improved in the future.

On the other hand, the computation time is reduced by 4.4%, and the resource usage increases by only 7.1%. This means that the generated function itself is very well

Table 2. Comparison of JPEG decoder between manual mapping and library-based mapping

Comparing list			SW only	manual	library-based mapping		
			#cycles	#cycles	#cycles	compared to	
						manual	SW only
JPEG decoder	dequant + IDCT	computation	7,709,984	70,092	67,716	96.6%	0.88%
		communication	-	580,441	3,331,358	573.9%	-
		total	7,709,984	650,533	3,399,074	522.5%	44.1%
	entire system		18,788,323	11,740,826	14,513,855	123.6%	77.2%
resource (memory size in bytes)			-	1,568	1,680	107.1%	-
mapping time			-	a day	ten minutes	<1%	-

optimized and so shows viability of the library-based mapping.

Most of all, library-based mapping has a great merit in the development time. When the mapping is done manually, all kinds of configuration code and the control code should be written manually taking long time. Debugging them also takes long time. Although IDCT is not a big example, mapping it takes more than a day. However, if our approach is used, 10 minutes is enough. All kinds of code are generated by just selecting the library functions and setting the parameters. After the function selection, the rest of the process is automated and the correctness is guaranteed.

V. CONCLUSIONS

We have presented a new technique for mapping an application kernel onto reconfigurable array architecture. The mapping based on a parameterized library makes the process very easy, and the development time and effort are reduced dramatically to under 1%. Furthermore, the library provides perfect abstraction so anyone can use it to program the system without any knowledge about the internal architecture of the reconfigurable array. However, there is a limitation in avoiding the overhead due to inefficient communications between the library functions mapped onto the same array, which is to be addressed in our future work.

ACKNOWLEDGMENTS

This work was supported in part by KOSEF under NRL Program Grant R0A-2008-000-20126-0 funded by MEST, Korea, in part by NIPA under ITRC Support Program NIPA-2009-(C1090-0902-0024) funded by MKE, Korea, and in part by Nano IP/SoC Promotion Group of Seoul R&BD Program 10560.

REFERENCES

[1] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, “DRESC: a retargetable compiler for coarse-grained reconfigurable architectures,” in *Proc. ICFPT*, 2002.

[2] J. Lee, K. Choi, and N.D. Dutt, "An algorithm for mapping loops onto coarse-grained reconfigurable architectures," in *Proc. ACM Workshop on Languages, Compilers, Tools for Embedded Systems*, pp.183-188, Jun. 2003.

[3] J. Lee, K. Choi, and N.D. Dutt, “Compilation approach for coarse-grained reconfigurable architectures,” *IEEE Design & Test of Computers*, vol. 20, pp. 26-33, Jan./Feb. 2003.

[4] M. Ahn, J. W. Yoon, Y. Paek, Y. Kim, M. Kiemb, and K. Choi, “A spatial mapping algorithm for heterogeneous coarse-grained reconfigurable architectures,” in *Proc. Design, Automation and Test in Europe*, pp. 363-368, Mar. 2006.

[5] T. Toi, N. Nakamura, Y. Kato, T. Awashima, K. Wakabayashi, and L. Jing, “High-level syntheis challenges and solutions for a dynamically reconfigurable processor,” in *Proc. ICCAD*, Nov. 2006.

[6] G. Lee, S. Lee, and K. Choi, “Automatic mapping of application to coarse-grained reconfigurable architecture based on high-level synthesis techniques,” in *Proc. ISOCC*, 2008.

[7] M. Jo, V.K.P. Arava, H. Yang, and K. Choi, “Implementation of floating-point operations for 3D graphics on a coarse-grained reconfigurable architecture,” in *Proc. IEEE International SOC Conference*, pp.127-130, Sep. 2007.

[8] W.-H. Chen, C.H. Smith, and S.C. Fralick, “A fast computational algorithm for the discrete cosine transform,” *IEEE Trans. on Communications*, vol. COM-25, pp. 1004–1009, Sep. 1977.

Kyuseung Han received the B.S. degrees in Electrical Engineering from Seoul National University, Seoul, Korea, in 2008. He is currently a graduate student under the unified M.S./Ph.D. degree program in the department of Electrical Engineering and Computer Science at Seoul National University. His research interests include reconfigurable architecture and automation of its soft-ware development.

Kiyoung Choi received the B.S. degree in electronics engineering from Seoul National University, Republic of Korea, in 1978, the M.S. degree in electrical and electronics engineering from KAIST, Republic of Korea, in 1980, and the Ph.D. degree in electrical engineering from Stanford University, Stanford, CA, in 1989. From 1989 to 1991, he was with Cadence Design Systems, Inc. In 1991, he joined the faculty of the Department of Electrical Engineering and Computer Science, Seoul National University. His primary interests include various aspects of computer-aided electronic systems design including embedded systems design, high-level synthesis, and lower-power systems design. He is also interested in computer architecture and especially in configurable and reconfigurable computer architecture design.