A Compiler Directed Approach to Hiding Configuration Latency in Chameleon Processors

Xinan Tang, Manning Aalsma, and Raymond Jou

Chameleon Systems, Inc.
161 Nortech Parkway
San Jose, CA 95134
tang@chameleonsystems.com

Abstract. The Chameleon CS2112 chip is the industry's first reconfigurable communication processor. To attain high performance, the reconfiguration latency must be effectively tolerated in such a processor. In this paper, we present a compiler directed approach to hiding the configuration loading latency. We integrate multithreading, instruction scheduling, register allocation, and prefetching techniques to tolerate the configuration loading latency. Furthermore, loading configuration is overlapped with communication to further enhance performance. By running some kernel programs on a cycle-accurate simulator, we showed that the chip performance is significantly improved by leveraging such compiler and multithreading techniques.

1 Introduction

With the rapid progress of reconfigurable computing technology, the new generation of reconfigurable architectures support runtime configuration to execute general-purpose programs efficiently [10, 7, 13, 6]. Runtime configuration becomes an essential feature to allow reconfigurable machines to compete with the main-stream RISC, VLIW, and EPIC machines.

However, the runtime reconfiguration latency can be significant. To maximize program execution performance, such loading overhead must be minimized. Various techniques have been proposed to reduce/tolerate the configuration latency. Configuration caching [10], prefetching [8], and compression [9] are named as a few. In this paper, we propose a compiler directed approach that exploits chip hardware to tolerate the configuration latency. We believe that effectively hiding the configuration latency is the key to achieving high performance on Chameleon like processors.

In our approach, four major techniques [11], *multithreading*, *instruction scheduling*, *register allocation*, and *prefetching* are leveraged to *hiding* the configuration loading latency. Our experimental results show that such an integrated

approach can double performance and it is very effective in hiding the reconfiguration latency.

In Section 2, we briefly introduce the Chameleon chip and the compiler environment. In Section 3, we formulate and explain the latency-tolerance problem. In Section 4, we present the compiler based integrated solution. In Section 5. we report experimental results and analyze the performance impacts. Finally, related work is reviewed in Section 6, and future work is discussed in Section 7.

2 Chameleon Hardware Model and Software Environment

The Chameleon reconfigurable chip is a processor based reconfigurable architecture. We briefly describe the Chameleon hardware model and the software environment in this section.

2.1 Chameleon Architecture Model

The Chameleon chip provides a platform for high-performance telecommunication and datacommunication applications[4]. It is a processor-based reconfigurable architecture, in which a RISC core, the reconfigurable fabric, a fast bus, the memory system, and IO are built in a single chip. Fig. 1(a) gives an abstract architecture model of Chameleon CS2112 chips.

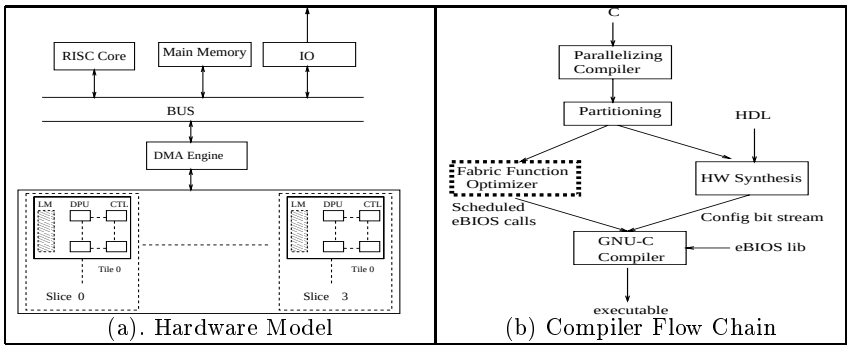

Fig. 1. Abstract hardware and software models for Chameleon chips

A 32-bit RISC core is used as a host processor. It schedules computation-intensive tasks onto the programmable logic. The programmable logic (fabric) is an array of 108 data path units (DPU). Each DPU can be dynamically reconfigured to execute one of eight instructions. The 108 DPUs are divided into four *slices* and each slice is further partitioned into 3 *tiles*. In each tile there are nine DPUs, of which seven are 32-bit ALUs and two are 16-bit multipliers. The inputs to a DPU can be changed on the cycle base. All instructions are executed in one cycle. Each DPU can have maximum 8 instructions stored in the control unit (CTL), and the next instruction is determined by the CTL within the same cycle. By loading a new configuration bit stream, the reconfigurated DPUs can

perform new computation. In addition, there are 8Kbyes of *local memory* (LM) for each slice and a DPU can read/write 32 bits in two cycles. Inside the chip, a high-speed bus links the core, the fabric, the main memory, and other IO devices together.

A config bit stream is stored in the main memory. It is loaded onto the fabric at runtime by DMA. Each slice has two planes for bit streams. An *active* plane executes the working bit stream and a *back* plane contains the next config bit stream. Switching from the back plane to the active one takes one cycle. Therefore, the back plane can be effectively used as cache for loading configuration.

Since the CS2112 chip is a loosely coupled reconfigurable architecture, it requires to move a large chunk of data from the main memory to the fabric local memory to cover the configuration and communication latency. Therefore, the granularity of program to be executed on the fabric is best at the function level. Unlike fine-grained reconfigurable architectures such as PRISC [12] and Chimaera [7], the Chameleon fabric is very efficient at executing medium-grained functions that contain tens or even hundreds of instructions. At such a medium-grain level, loading a config bit stream may take up to hundreds of cycles if there is a cache miss. Thus, effectively to tolerate the loading latency is the key to attaining high performance on the Chameleon chip.

2.2 Chameleon Software Environment

Fig.1(b) outlines the Chameleon software environment. Like other compilers for reconfigurable architectures [3, 5, 16, 2], it mainly consists of two compilers. An optimizing compiler takes a C program and partitions the program into two parts: (1) code suitable to run on the RISC core; (2) code profitable to execute on the reconfigurable fabric. The RISC part is then compiled by a GNU C compiler and the fabric one is passed to a hardware (HW) synthesis compiler in an intermediate form. For the efficiency reason, the HW compiler also takes in programs written in Verilog and generates config bit streams. The config bit stream can then be linked with other compiled code to form an executable.

To launch a function onto the fabric, an eBIOS (runtime system) is designed to support: (1) multithreaded execution between the core and the fabric; (2) communication and synchronization between the core and the fabric. To hide the configuration latency, the sequence of eBIOS calls must be carefully scheduled to exploit parallelism. The fabric-function optimizer is an optimization module that performs (1) eBIOS call scheduling; (2) static resource allocation to make full use of available hardware resources. In the following, we will focus on the latency tolerant techniques employed in the fabric-function optimizer.

3 Problem Statement

To run a function on the fabric, the call is replaced by a series of equivalent eBIOS calls to perform: (1) loading a config bit stream; (2) moving data in DMA; (3)

firing the fabric. The scheduling problem studied in this paper is to place a series of calls into a proper order so that (1) control and data dependences are obeyed. (2) total program execution time is minimized. To minimize total execution time, the configuration overhead must be minimized. Thus, if the configuration overhead is reduced, the total execution time is also effectively reduced.

Formally, we can formulate the problem as follows: *given a series of fabric function calls $F = f_1, ..., f_n$, each f_i has its parameter list l_i, and its corresponding config bit stream c_i. Find a schedule S that consists of a series of eBIOS calls so that the total execution time on the processor is minimized.*

In this paper, F can be a *fork/join* series, and function f_i has only one config bit stream c_i. Therefore, partially reconfiguration is not considered. Fig. 2(a) lists a series of four function calls, f_1, f_2, f_3 and f_4. In the series, functions f_2 and f_3 can run in parallel.

```
f1;
fork
  f2;
  f3;
join;
f4
```
(a) sequence.

```
1| #pragma cmln mac(\
2|  in  x[N],in y[N],\
3|  out *z)
4|
5|    mac(x, y, &z);
6|
```
(b) C call

```
1| LOAD_CONFIG(mac_bits);
2| WAIT_FOR_CONFIG();
3| DMA_MOVE(X, 4*N, LM_1);
4| WAIT_FOR_DMA();
5| DMA_MOVE(Y, 4*N, LM_2);
6| WAIT_FOR_DMA();
7| FIRE_FABRIC();
8| WAIT_FOR_FABRIC();
9| SCALAR_MOVE(&z, DPU_1);
```
(c) schedule 1

```
1| LOAD_CONFIG(mac_bits);
2| DMA_MOVE(X, 4*N, LM_1);
3| DMA_MOVE(Y, 4*N, LM_2);
4| WAIT_FOR_CONFIG();
5| WAIT_FOR_DMA();
6| FIRE_FABRIC();
7| WAIT_FOR_FABRIC();
8| SCALAR_MOVE(&z, DPU_1);
```
(d) schedule 2

Fig. 2. Fork/join program representation and eBIOS schedules.

Let's see an example that shows the scheduling impacts. Fig. 2 (b), a C call is indicated to the scheduler in the pragma line (lines 1-3), which says function `mac` has two input arrays and one output scalar. Two eBIOS schedules are listed in Fig. 2 (c) and (d) respectively.

Schedule 1 listed in Fig. 2 (c) first loads the config bit stream of `mac` (lines 1-2). Then two input arrays are sent to the fabric using DMA (lines 3, 4, 5, and 6). Next, the fabric is fired (line 7). After the completion of running (line 8), the scalar result is retrieved (line 9). However this schedule is not an effective one. First, it does not issue any DMA operation after configuration loading; Second it does not pipeline the DMA issuing operations. Schedule 2 listed in Fig. 2 (d) is better by pipelining two type of operations: loading a configuration and moving data in DMA (lines 1-3). Moreover, it reduces one synchronization (WAIT_FOR_DMA). Thus a better schedule will have significant performance impacts. In the following, we will discuss how a 'good' schedule can be found out.

4 Scheduling Algorithm and Compiler Optimizations

Since the problem formulated is NP-*Complete*, feasible algorithms require the use of heuristics. We will first introduce the multithreaded eBIOS, and then present our heuristic scheduling algorithm. Finally a series of compiler optimizations are also described.

4.1 Multithreaded Runtime System

The eBIOS is a runtime system that supports the *fork/join* style of parallelism. A master thread runs on the RISC core and other slave threads can run concurrently on the fabric. To support such an execution model, the eBIOS must support *split-phased* (asynchronized) transactions between the core and the fabric [15]. Particularly, the following operations are *non-blocking* in terms of the RISC core execution: (1) **LOAD_CONFIG:** loading a config bit stream onto the fabric; (2) **DMA_MOVE:** moving data between the main memory and the fabric; (3) **FIRE_FABRIC:** activate computation on the fabric.

However, such multithreaded execution inadvently adds to the programming complexity. It is essential for the scheduling algorithm to find a good schedule to guarantee: (1) the 'right' combination of eBIOS calls that is dead-lock free; (2) the 'best' schedule that has minimized execution time.

4.2 Heuristic Scheduling Algorithms

A two-level heuristic is used to solve the scheduling problem. First, we aggressively schedule eBIOS calls that belong to the same function call. Second we hoist up certain operations between two neighbor function calls to exploit parallelism.

Fig. 3(a) gives a list scheduling based algorithm that arranges the eBIOS calls within the same function-call boundary. Given an input function f_i, parameter list l_i, and config bit stream c_i, The algorithm works as follows. After issuing load config bit stream (line 2), the algorithm sorts the input parameter list l_i into four sublists (line 3): (1) l_i^1 is an input array list; (2) l_i^2 is an output array list; (3) l_i^3 is an input scalar list; (3) l_i^4 is an output scalar list. The purpose of such sorting is to facilitate handling data dependences between parameters and the function call. For list l_i^1, we further sort it into a *decreasing* order according to the length of input array (line 5). Then, we take each input array from the sorted list and issue DMA_MOVE operation one by one (lines 7 and 8). The reason of issuing longer DMA operations earlier is to use their execution time to cover the DMA setup costs of shorter DMA operations.

Next, operation WAIT_FOR_CONFIG is issued to guarantee that the config bit stream arrives on the fabric. After that scalar parameters can be sent onto the fabric (lines 11-13). The reason of DMA first and scalar second is that a config bit stream may modify some DPU registers while DMA operations can run in parallel with loading configurations. Then, we check whether the previously issued DMAs have finished (line 15). Afterwards, we can start fabric computation (line 16), and wait for its completion at line 17. Then we continue to process output parameters accordingly (lines 18-29 shown in the right).

In summary, the heuristics used in the scheduling algorithm are as follows: (1) overlap loading a config bit stream and DMA operations (lines 2-9); (2)

```
1 | RTS_schedule(f_i, l_i, c_i) {
2 |   select(LOAD_CONFIG) for c_i;
3 |   (l1,l2,l3,l4) = sort_parameter_list(l_i)
4 |   if (|l1| > 0) { /* Input arrays */
5 |       sort l1 into a non-ascending order;
6 |       foreach array input in sorted l1 {
7 |         select (DMA_MOVE, READ);
8 |       }
9 |   }
10|   select(WAIT_FOR_CONFIG);
11|   foreach scalar input in l3 {
12|       select(SCALAR_MOVE, READ);
13|   }
14|   if (|l1|)
15|       select(WAIT_FOR_DMA);
16|   select(FIRE_FABRIC);
17|   select(WAIT_FOR_FABRIC);
18|   if (|l2| > 0) { /* output arrays */
19|     sort l2 into a non-ascending order;
20|     foreach array output in sorted l2 {
21|       select (DMA_MOVE, WRITE);
22|     }
23|   }
24|   foreach out scalar in l4 {
25|     select(SCALAR_MOVE, WRITE);
26|   }
27|   if (|l2| > 0)
29|     select(WAIT_FOR_DMA);
30| }
```

Fig. 3. Scheduling algorithms for eBIOS calls

pipeline DMA operations to cover up their setting up costs (line 4-9, and 18-23); (3) use DMA operations to hide scalar operations (lines 11-15, and 24-29). The time complexity of the algorithm is $O(nlogn)$ assuming $|l_i| = n$. Most of time is spent on sorting l_1^i and l_2^i into a proper order.

For scheduling eBIOS calls from multiple function call sites, the resource conflict analysis should be done first. Two concurrent functions f_i and f_j are resource free if (1) the combined number of slices used is less than 4; (2) the intersection of the slice set is empty; (3) the intersection of the DMA set is empty. Otherwise two functions have to be executed sequentially due the resource constraints.

The minimum scheduling cluster (MSC) of function f_i is defined as:

$$MSC_i = \begin{cases} \{f_{i+1}, \ldots, f_{i+k}\} \text{ if } free(f_i, f_{i+1}) \ \& \ \ldots \& free(f_i, f_{i+k}) \ \& \ conflict(f_i, f_{i+k+1}) \\ \{f_{i+1}\} \qquad \text{otherwise} \end{cases}$$

Ideally, function f_i should be scheduled together with other functions within the same MSC_i. To make the scheduling tractable, only two neighbor function calls are considered, $MSC_i = \{f_{i+1}\}$. This is based on the fact that there are only two config planes on the Chameleon CS2112 chip. Furthermore, we only hoist up the config loading operation in between the firing fabric and the waiting for its completion. Thus, we try to use fabric computation time of one function to overlap configuration loading for another function.

4.3 Compiler Optimizations

In addition to scheduling, other compiler optimizations are also applied to reduce execution time.

Function inlining is a technique to replace a call with the function body to reduce the stack manipulation overhead. By inlinging the original C call with a series of eBIOS calls, the actual parameters are directly bound to the eBIOS calls and the program execution time is reduced significantly.

Static resource allocation is a technique similar to register allocation. The resources that a fabric function needs are slices and DMA channels. If resources for a fabric function can be statically allocated for a fabric function, the overhead of dynamic resource allocation such as address computation can be eliminated completely.

Synchronization between the core and the fabric must be done to enforce certain order. For example, WAIT_FOR_CONFIG waits for the config loading to finish and WAIT_FOR_DMA waits for DMAs to finish. There is an *autonomous* working mode in the CS2112 chip in which synchronization is done by hardware automatically. Our scheduling algorithm can identify such a case and eliminate unnecessary synchronization when the autonomous mode can be applied.

5 Experimental Results

To test the efficacy of heuristic algorithms, we use kernel benchmarks to measure the effectiveness of our scheduling algorithm and compiler optimizations. The major results are as follows: (1) the eBOS scheduling algorithm can dramatically increase program execution performance up to 60% (see Section 5.1); (2) the prefetching algorithm can boost performance up to 30% (see Section 5.2); (3) by using the autonomous mode, performance can be further increased by up to 15% for certain benchmarks (see Section 5.3);

Table 1 lists main characteristics of benchmarks used. Benchmarks `fht`, `fir24`, and `pngen` are kernels for CDMA systems. The DMA is used in all benchmarks. The length of a config bit stream is given in Kilo bytes. We expect that the longer of a config bit stream, the more effective of our scheduling algorithms.

The simulator used is a commercial cycle-accurate simulator, ModelSim. Since we simulate the entire chip at the RTL level, timing information is guaranteed to be cycle accurate. During the experiment, function-inlining is always applied since it definitely enhances performance.

Table 1. Benchmark Description

Name	Description	Bit Stream Length	DMA Channels	
			input	output
mac	vector product	0.9K	2	0
addvec	vector addition	1.1K	2	1
fht	hadamard function	2.1K	1	1
fir	FIR filter	2.1K	8	0
fir24	24 tap	$\simeq 5.0K$	> 1	≥ 1
pngen	PN generator	$\simeq 5.0K$	> 1	≥ 1

5.1 Effects of eBIOS Scheduling

Table 2 lists program execution performance for programs generated by different scheduling algorithms. $O0$ means a naive scheduling algorithm in which each

Table 2. Performance of the eBIOS scheduling algorithm.

	schedule	mac	addvec	fht	fir	fir24	pngen	ave.
time	O0	3303	3577	3518	6007	4705	8021	
	O1	3074	3326	3529	4375	4323	6955	
	O2	1612	2048	2446	2472	3069	5535	
imp(%)	0,1	7	7	-0	27	8	13	10
	1,2	48	38	31	43	29	20	35
	0,2	51	43	30	59	35	31	42

config loading and DMA operation is issued sequentially. $O1$ means applying the scheduling algorithm. $O2$ means using the static slice allocation on top of O1. The first three rows give corresponding execution times measured in cycles and the next three rows list the improvement rate, computed by

$$Imp_{i,j} = (T_{Oi} - T_{Oj})/T_{Oi} \tag{1}$$

From Table 2, we can see that on average *imp* is 10% when the scheduling algorithm is applied ($imp_{0,1}$ row). Benchmark `fir` has the highest improvement rate (27%) since the number of DMAs used is the biggest (8). However benchmark `fht` has a negligible negative impact. The reasons are two fold. First, the number of DMAs are smaller (1 input and 1 output). Second, pipelining config loading and DMA operations may cause the bus contention. This result indicates that the scheduling algorithm will have a big performance impact if more DMAs are used and the bus contention is not an issue.

When applying the static resource allocation ($imp_{1,2}$ row), the performance of all benchmarks is increased. The reason is that after the static resource allocation is applied the critical path of entire program execution is reduced. Therefore, the optimization impact becomes more visible.

In general, the combined improvement rate is 42%. This shows that the combined scheduling algorithm has a significant impact on performance.

5.2 Effects of Prefetching

Table 3(a) lists the performance impacts of the the prefetching based scheduling algorithm on the two combined benchmarks, `mac-addvec` (mac+addvec) and `fir24-pngen` (fir24+pngen). The experiment was done by turning on/off option $-O3$. When option $-O3$ is *off*, all functions in two test cases run sequentially. When option $-O3$ is *on*, `mac-addvec` runs in parallel since there is no resource conflict but `fir24-pngen` is forced to run sequentially since there is resource conflict. However loading config bit stream for `pngen` is prefetched to the back planes. Therefore, loading config bit stream for function `pngen` runs in parallel with the execution of function `fir24` on the fabric.

In Table 3(a), row `off` corresponds to program execution time when the prefetching algorithm is not applied. Row `on` corresponds to execution time when the algorithm is applied. Row `conflict` indicates whether there is resource conflict. Row `imp` gives the performance improvement rate.

Table 3. Performance of Compiler Optimizations.

O3	mac-addvec	fir24-pngen	ave.
off	3301	8231	
on	3048	5812	
conflict	no	yes	
imp(%)	7.66	29.39	18.53

(a) prefetching

Auto	mac	addvec	fht	fir24	Ave.
no	1612	2048	2446	3069	
yes	1510	1739	1966	2735	
imp(%)	6.0	15.0	20.0	11.0	13.0

(b) autonomous mode

From Table 3(a), we can see that on average performance is increased by almost 19% when the prefetching based scheduling algorithm is applied. Comparatively, `fir24-pngen` has bigger performance improvement over `mac-addvec`, 29.39% vs. 7.66%. The reasons are two fold. First, the fabric running time of `fir24-pngen` is longer than that of `mac-addvec`. Second, the length of the config bit stream of `fir24-pngen` is also longer that that of `mac-addvec`(See Table 1). Therefore, prefetching the config bit stream of `fir24-pngen` is more rewarding.

5.3 Effects of Using the Hardware Feature

Table 3 (b) lists program execution performance before and after the fabric autonomous execution feature is applied. This experiment is done based on $O2$ optimization. The *yes/no* rows in Table 3 (b) give execution time of a program *using/not using* the hardware feature correspondingly. Only four benchmarks are qualified for such an optimization. On average, execution performance is improved by 13%. The improvement rate of the benchmarks that have output DMAs (`addvec`,`fht`, and `fir24`) is bigger than the one that does not have (`mac`). The reason is that an extra synchronization for output DMAs was also eliminated. This suggests that the hardware feature should be exploited whenever possible.

6 Related Work

In compiling for reconfigurable architectures, most of work focuses on the code partitioning and parallelizing loops [3, 5, 16, 2]. On these machines [10, 7, 13, 6], the processor usually stalls when a configuration is loaded. The performance impact of prefetching has been studied by Hauck et. al. [8]. However the communication overhead is not considered in the study. Viswanath et. al. [14] did a quantitative case study on the effects of the communication overhead, and they identified the importance of reducing such overhead. Bondalapati et. al. [1] proposed a general model mapping loops onto the reconfigurable architecture. Our problem formulation is different from theirs by considering the fork/join tree and communication cost. In this paper, we propose a compiler-directed approach to hiding the 'interface' latency, including the reconfiguration and communication latencies. To the best of our knowledge, this is the first integrated effort to leverage compiler and multithreading techniques to solve this problem. We believe that our approach can also be applied to other similar architectures.

7 Conclusions and Future Work

We have developed a compiler-directed approach, combining compiler optimization and multithreading techniques, to hide the configuration loading latency. We have implemented the list scheduling based algorithm that find a 'best' schedule for a series of eBIOS calls. The experimental results are very encouraging, and performance has been significantly improved by applying the integrated method. The future work will be: 1) continue to improve the scheduling algorithms; 2) design advanced resource allocation schemes; 3) investigate better prefetching algorithms. We will also experiment on our chips with real applications.

References

1. K. Bondalapati and V. K. Prasanna. "mapping loops onto reconfigurable architectures". In *Proc. of Inter. Workshop on Field Programmable Logic and Applications*, Sep. 1998.
2. M. Budiu and S. C. Goldstein. "fast compilation for pipelined reconfigurable fabric". In *Proc. of ACM/SIGDA Inter. Symposium on FPGA*, 1999.
3. T. J. Callahan and F. John Wawrzynek. "instruction level parallelism for reconfigurable computing". In Hartenstein and Keevallik, editors, *Inter. Workshop on Field-Programmable Logic and Applications*. Lecture Notes in Computer Science, LNCS 1482,Springer-Verlag, Aug. 1998.
4. Chameleon Systems, Inc. http://www.chameleonsystems.com/, 2000.
5. M. Gokhale and J. Stone. "NAPA C: Compiling for a hybrid risc/fpga architcture". In *Proc. of the IEEE Symposium on FCCM*, Apr. 1998.
6. S. C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. R. Taylor, and R. Laufer. PipeRench: A coprocessor streaming multimedia acceleration. In *Proc. of ISCA-26*, pages 28–39, Atlanta, Geor., May 1999.
7. S. Hauck, T. W. Fry, M. M. Hosler, and J. P. Kao. "the chimaera reconfigurable functional unit". In *Proc. of the IEEE Symposium on FCCM*, Apr. 1997.
8. S. Hauck, T. W. Fry, M. M. Hosler, and J. P. Kao. ""configuration prefetch for single context reconfigurable coprocessors"". In *Proc. of ACM/SIGDA Inter. Symposium on FPGA*, Feb. 1998.
9. S. Hauck, Z. Li, and E. J. Schwabe. "configuration compression for the xilinx xc6200 fpga". In *Proc. of the IEEE Symposium on FCCM*, Apr. 1998.
10. J. R. Hauser and J. Wawrzynek. "garp: A mips processor with a reconfigurable coprocessor". In *Proc. of the IEEE Symposium on FCCM*, Apr. 1997.
11. S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.
12. R. Razdan. *PRISC: Programmable Reduced Instruction Set Computers*. PhD thesis, Harvard University, Division of Applied Sciences, Boston, 1994.
13. C. R. Rupp, M. Landguth, T. Garverick, E. Gomersall, H. Holt, J. M. Arnold, and M. Gokhale. "the napa adaptive processing architecture". In *Proc. of the IEEE Symposium on FCCM*, Apr. 1998.
14. S.K.Rajamani and P.Viswanath. "a quantitative analysis of the processor-programmable logic interface". In *Proc. of the IEEE Symposium on FCCM*, Apr. 1997.
15. X. Tang and G. R. Gao. Automatically partitioning threads for multithreaded architectures. *Journal of Parallel and Distributed Computing*, 58(2):159–189, Aug. 1999.
16. M. Weinhardt and W. Luk. "pipeline vectorization for reconfigurable systems". In *Proc. of the IEEE Symposium on FCCM*, Apr. 1999.