

# A New Processor Architecture with a New Program Driving Method

Xiaobo Li, Ke Luo, Xiangdong Cui, Lalin Jiang, Xiaoqiang Ni,  
Chiyuan Ma, Jingfei Jiang, Huiping Zhou, Zhou Zhou  
School of Computer and Communication Engineering  
Changsha University of Science and Technology  
45 Chiling Road, Changsha, Hunan, P.R. China

**Abstract** - *This paper proposes a new type of processor architecture using a new program driving method which makes it possible for more programs to run in a single kernel processor concurrently without using interrupt technique. The main idea of the method is to collect all the program/instruction driving elements/factors to form a driving vector, called Program Driver (PD) for short, so that more than one PD can drive its own program to run in a single kernel processor concurrently.*

**Keywords:** processor, program, instruction, parallelism, dependency.

## 1 Introduction

Many programs and/or tasks are required to run in one single kernel processor can be found in many real life applications. For example, while a PC user is editing a file, the PC's system routines, such as an anti-virus routine, the routines for monitoring/processing keyboard/mouse, and so on, are busy in doing its work. Traditional processors use the time-sharing technique and the interrupt technique to run more than one program in one processor "concurrently". Obviously, these programs are not in a real parallel way. In order to exploit the parallelism, more attentions are then paid to exploiting the parallelism within one program, such as pipeline technique, VLIW technique, multi-threading technique and so on [1,2,3,4,5,6,7]. However, much dependency among the instructions and/or threads of one program often limits the parallelism that can be gained and the cost is often high.

In the other hand, the dependency among the instructions from different programs is much smaller, and the parallelism is more easily achieved. In order to exploit the parallelism of the instruction-level among programs, a single kernel processor has to be able to run more than one program in a real parallel way. However, almost all the traditional processors use hardware PCs to control programs to execute, named "hardware PC+". This driving method is obviously unable to run more than one program in a single kernel processor in a real parallel way, and therefore is unable to exploit the parallelism of instruction-level among programs. Multi-threading, multi-CUs', multi-

processors, and multi-kernel processors can be classified as the type of "the fixed number of hardware PC+". This sort of driving methods is also unsuitable for exploiting the parallelism of instruction-level among programs due to the limitation of the fixed number of PC's.

This paper proposes a new type of computer processor (called Concurrent Multiple Program Processor CMPP) architecture using a new program driving method which makes it possible for more programs to run in a single kernel processor concurrently without using the interrupt technique. The next section of this paper gives the main idea of the new driving method, the third section presents the processor architecture based on the new driving method, the 4th section show how the proposed processor works, and the last section concludes this paper.

## 2 Program Driving Mechanism

The key elements for driving a program to execute are Program Counter, Instruction Register, Program State Word, Intermediate Result Buffers, and so on. If we put all of the key driving elements together to form an information vector, we then create "something" like a program's "soul/spirit" since it can make the program to be "alive", that is, to execute the instructions of the program, just like there is something (usually called the soul/spirit) inside the body of a person which makes the person to be alive. Furthermore, just like many of such souls/spirits can control many persons to live in the world concurrently, many program's "souls/spirits" can also drive many programs to execute concurrently in a processor.

If we put all of the key driving elements/factors together to form an information vector, we then create "something" like a program's "soul/spirit" since it can make the program to be "alive", that is, to execute the instructions of the program, just like there is something (usually called the soul/spirit) inside the body of a person which makes the person to be alive. Furthermore, just like many of such souls/spirits can control many persons to live in the world concurrently, many program's "souls/spirits" can also drive many programs to execute concurrently in a processor.

Based on what has been mentioned above, this paper proposes a new driving method for executing programs. There are two main points of the new drive method: the one is to collect all of the key driving elements/factors to form an information vector, called Program Driving Vector (PDV) or Program Driver (PD) for short. And the other is to design the functional units of a processor to act as service providers so that program drivers can behave the way as service clients.

Let's consider one of the functional units of a processor: the data storage control unit. Just like a bank in real life is open for the service of depositing and/or withdrawing money, a data storage control unit could also be designed to provide the service of reading and/or writing data. And just like a person goes to a bank to request the service of withdrawing money when he/she needs money to buy something, a PD can also "go" to a data storage control unit to request the service of reading data when it needs to read an operand from the data storage. After the service of reading an operand has been done, the PD may "go" to the next functional unit to request the next service according to its next execution step of the current instruction, for instance, "go" to ALU to request the service of "+" operation. Obviously, there can be more than one PD "living" in a single kernel processor, which means many programs can be executed in a single kernel processor concurrently. A PD can drive its program to run independently. Also, a PD may cooperate with other PDs via some synchronized mechanism.

In order to make the new driving method to work, special improvements should be made to the design of processor architectures. Firstly, for each instruction executing step, such as fetching instruction step, decoding step, fetching operand step, operating step, writing-back step, and PC+ step, a certain processor's functional unit should be designed to carry out the given operation. Secondly, the functional units should be designed in the way by which the units can work as service providers so that a PD can behave as a service client. Thus, a PD, according to the certain sequence of the instruction executing steps, "travels" around the processor's functional units to request the services provided by the units, that is, the PD may "go" to an instruction fetching unit to fetch an instruction, and then the PD may go to a decoding unit to decode the instruction, ..., and then the PD may go to a PC+ unit to have its PC renewed, and then the PD may go back to the instruction fetching unit to fetch the next instruction, and so on. This is why we say a PD is like a program's "soul/spirit" which "lives" in a processor and drives the program to run.

### 3 CMPP Architecture

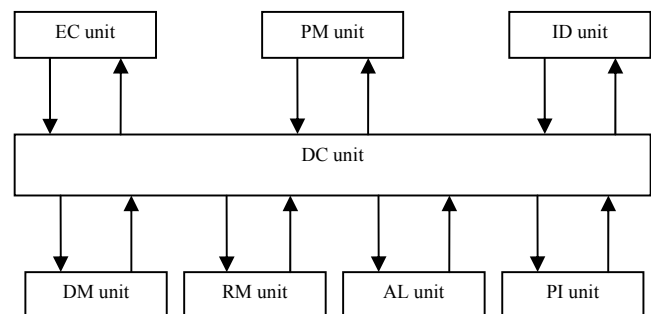
In this section, we firstly give the definition of a program driver PD, which is as follows.

UN0/OP0, UN1/OP1, ...	P N	PC	P S W	TB0	TB1	TB2
--------------------------	--------	----	-------------	-----	-----	-----

Here, a PD is made up of seven fields (information blocks). The descriptions for each field are:

- The field named "UnitNo/Operation Sequence" (UN/OPseq for short) is used to keep a sequence of pairs as UN0/OP0, UN1/OP1, ... The UN0/OP0 is called the first UN/OP pair, and so on. The sequence means the PD will first go to the functional unit named UN0 to request the service named OP0, then go to the functional unit named UN1 to request the service named OP1, and so on.
- The field named PN is used to keep the number of a program, or the name of a program.
- The field named PC is used to keep the program counter of a program, that is, the current instruction address of the program.
- The field named PSW is to keep the program state word of a program, such as carry bit, etc.
- The fields named TB0 – TB2 are used to keep the intermediate results. Note that, a PD in real life might need more temporary buffers.

In order to provide a means for a program driving PD to work, there are eight functional units in the proposed processor CMPP. The descriptions for each functional unit are as follows.



There are eight functional units in the proposed processor. The descriptions for each functional unit are as follows.

- Delivering Control Unit (DC) is designed to provide the service of delivering PDs from one unit to the other. The destination address is UN0 in the field "UN/OPseq" of PDs.
- Program Storage Management Unit (PM) is to provide the services related to program storages.
- Instruction Decode Unit (ID) is designed to provide the service of instruction decoding.
- Register File Management Unit (RM) is designed to provide the services related to the register file, such as reading one or more operands from the registers, or

write a result into a register.

- Data Storage Management Unit (DM) is designed to provide the services related to data storages.
- Arithmetic/Logical Operation Unit (AL) is designed to provide the services related to the arithmetic/logical operations, such as +, -, \*, etc.
- Peripheral Interface Control Unit (PI) is designed to provide the service related to the management of peripheral equipments and/or its interfaces.
- Execution Control Unit (EC) is designed to provide the services related to the execution management like creating a new PD, deleting a PD, modifying the PC field of a PD, etc.

Every functional unit of the processor has an input queue and/or an output queue. When a PD reaches a functional unit, the PD will enter the input queue to wait for being served. After a PD has obtained the required service and is about to leave the unit, the PD will enter the output queue of the unit to wait for being delivered to its next destination.

If a functional unit is in an idle state or after it has just finished its last serving operation, the unit will then check if its input queue is empty or not. The unit will enter an idle state if its input queue is empty. Otherwise, it will remove a PD from the head place of the input queue and do the required operation for that PD. After the required service has been done, the unit should delete the first UN/OP pair in the UN/OPseq field of the PD, which makes the second UN/OP pair to be now the first UN/OP pair and lets the PD known where to go and what to do next. And finally the unit puts the PD into its output queue to wait for leaving. The main function of a delivering control unit DC is to deliver a PD from the output queue of the one unit to the input queue of the other. There can be more DCs working concurrently in a processor to ensure all PDs to be delivered in time.

## 4 Execution Mechanism

When the proposed processor is reset, the Execution Control unit (EC) will create a PD with the PD number to be 0, named PD[0]. PD[0] is used to drive the user's main program to run. The layout of PD[0] is as follows.

UN/OPseq	PN	PC	PSW	TB0	TB1	TB2
PM/R0, ID/D0	0	0	0			

The PN field is 0 standing for the number of the main program is 0. The value of the field named UN/OP Sequence is "PM/R0, ID/D0", which means the PD will firstly go to the PM unit, request a reading instruction operation and save the fetched instruction into the field TB0, and then the PD will go to the ID unit to request a decoding operation. Note that a functional unit will delete the first UN/OP pair of the field named UN/OPseq of a PD

after the unit has finished the service for that PD, so the PD is able to know where it should go and what service it should request next.

Note that a functional unit will delete the first UN/OP pair of the field named UN/OPseq of a PD after the unit has finished the service for that PD, so the PD is able to know where it should go and what service it should request next. To make shorter, this paper only select one normal instruction as an example to show how the ID unit decodes an instruction for a PD. Two special instructions will be also presented to show the parallel mechanism of the CMPP.

Suppose an PLUS instruction is of the form: ADD R6,R5,R4 which means  $R6 \leftarrow [R5] + [R4]$ , and a PD has gone to the PM unit requesting a instruction fetching service and returned with a PLUS instruction in its TB0 field, and now the PD reaches the ID unit requesting a decoding service. After the ID unit has done the decoding service for the PD, the result of the PD fields is as follows.

UN/OPseq	PN	PC	PSW	TB0	TB1	TB2
RM/R12, AL/1←1+2, RM/W0←1, EC/PC+, PM/R0, ID/D0	0	0	0	6	5	4

The explanation of the value of the field UN/OPseq is as follows.

- The first pair of UN/OP is RM/R12 which means the PD should firstly go to the RM unit to request the service of  $TB1 \leftarrow RM[PN][TB1]$  and  $TB2 \leftarrow RM[PN][TB2]$ . Since  $PN=0$ ,  $TB1=5$ , and  $TB2=4$ , the requested service is actually  $TB1 \leftarrow RM[0][5]$ ,  $TB2 \leftarrow RM[0][4]$ .
- The second pair of UN/OP is AL/1←1+2 which means the PD should then go to the ALU unit to request the service of  $TB1 \leftarrow TB1 + TB2$ .
- The 3rd pair of UN/OP is RM/W0←1 which means the PD should then go to the RM unit to request the service of  $RM[PN][TB0] \leftarrow TB1$ .
- The 4th pair of UN/OP is EC/PC+ which means the PD should then go to the EC unit to request the service of  $PC \leftarrow PC + 1$ .

In order to run more than one program in a single kernel processor, the proposed processor allows more than one PD to be generated via an CREATE instruction of the form: CREATE #PN,#PC which means create a new PD whose program number is #PN and starting address is #PC. For example, after the ID unit has done the decoding service for the PD currently carried the CREATE instruction of the form: CREATE #5,#1234, the result of the PD fields is as follows.

UN/OPseq	PN	PC	PSW	TB0	TB1	TB2
EC/CrePC0PN1, PM/R0, ID/D0	0	8	0	123 4	5	

The first pair of the field UN/OPseq is EC/CrePC0PN1 which means the PD should next go to the EC unit to request a creating PD service. Here, we suppose the PD that currently carries a CREATE instruction is PD[0] and its current PC is 8. When the PD[0] reaches the EC unit, the EC unit will do the following for the PD[0] since its required service is CrePC0PN1:

- Generate a new PD: the EC unit generates a new PD, sets the new PD's PC field to be the value of PD[0]'s TB0 field (PC←1234), sets the new PD's PN field to be the value of PD[0]'s TB1 field (PN←5, and the newly created PD is now PD[5]), sets the PD[5]'s UN/OP Sequence field to be "PM/R0,ID/D0", and finally sends the PD[5] to its output queue to wait for leaving. Afterwards, the PD[5] will go to the PM unit to request the instruction fetching service. The layout of PD[5] waiting in the output queue is as follows.

UN/OPseq	PN	PC	PSW	TB0	TB1	TB2
PM/R0, ID/D0	5	1234	0			

- Process PD[0]: the EC unit renews the PC field by PC+1, deletes the first pair of the UN/OP Sequence field, which causes the second pair now becomes the first pair, and finally sends the PD[0] to its output queue to wait for leaving. Afterwards, the PD[0] will go to the PM unit to request the instruction fetching service. The layout of PD[0] waiting in the output queue is as follows.

UN/OPseq	PN	PC	PSW	TB0	TB1	TB2
PM/R0, ID/D0	0	9	0			

From now on, the PD[0] and PD[5] are both on its way in controlling its programs to run in the proposed processor concurrently. By means of an END instruction, a PD can be deleted.

By means of an END instruction, a PD can be set to "death". This is the case when a program ends. An END instruction is used to delete a PD. An END instruction is of the form END. For example, after an ID unit has done the decoding service for the PD[5] currently carried an END instruction, the result of the PD[5] fields is as follows.

UN/OPseq	PN	PC	PSW	TB0	TB1	TB2
EC/End	5	123 9	0			

In the UN/OP Sequence field of PD[5], there is only one pair of UN/OP which tells the PD[5] to go to the EC unit to request an END service. When the EC unit receives

PD[5] from its input queue and finds that PD[5] is requesting an END service, the EC unit simply discards the PD (sends no PD to its output queue). Since then, there will be no PD[5] in the processor at all.

## 5 Conclusion

This paper proposes a new type of CMPP architecture using a new program driving method which makes it possible for more programs to run in a single kernel processor concurrently without using interrupt technique. The main idea of the new driving method is to collect all the program/instruction driving elements/factors to form a driving vector, so that more than one PD can drive its own programs to run in a single kernel processor concurrently. Although the CMPP and the new driving method is only described in principle in this paper due to the space limitation of the paper, the authors have spent years working on the project, and have developed a simulator using C++ and a FPGA machine using VERILOG based on the new program driving method. In the FPGA machine, about 200 instructions, including nearly all sorts of instructions, have been realized and as many as 32 programs can be run concurrently in the machine.

## 6 References

- [1] Atsushi Mizuno, et al. "Design Methodology and System for a Configurable Media Embedded Processor Extensible to VLIW Architecture". In: Proceedings of the 2002 IEEE ICCD 2002.
- [2] Gurindar S.Sohi, Amir Roth. "Speculative Multithreaded Processors". IEEE Compu. 2001.4.
- [3] Dean M. Tullsen, et al. "Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor". In Proceedings of the 23rd ISCA, 1996.
- [4] A. Moshovos et al., "Dynamic Speculation and Synchronization of Data Dependence", Proc. Of the 24th Int'l IEEE Solid-State Circuits Conf., IEEE Press, Piscataway, N.J., 2000.
- [5] S. Arya, H. Sachs, S. Duvvuru, "An Architecture for High Instruction Level Parallelism", Proc. Of the 28th Annual Hawaii Int'l Conf. on System Scienc, 1995.
- [6] G.S. Sohi, S.E. Breach and T.N. Vijakumar, "Multiscalar Processors", In Proc. Of Int'l Symp. On Computer Architecture, 1995.
- [7] N.P. Jouppi, D.W. Wall, "Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines", 1989 ACM 0-89791-300-0/89/0004/0272.