Run-Time Adaptive Flexible Instruction Processors

Shay Seng[1], Wayne Luk[1], and Peter Y.K. Cheung[2]

[1] Department of Computing, Imperial College, London, UK.
[2] Department of EEE, Imperial College, London, UK.

Abstract. This paper explores run-time adaptation of Flexible Instruction Processors (FIPs), a method for parametrising descriptions and development of instruction processors. The run-time adaptability of a FIP system allows it to evolve to suit the requirements of the user, by requesting automatic refinement based on instruction usage patterns. The techniques and tools that we have developed include: (a) a run-time environment that manages the reconfiguration of the FIP so that it can execute a given application as efficiently as possible; (b) mechanisms to accumulate run-time metrics, and analysis of the metrics to allow the run-time environment to request for automatic refinements; (c) techniques to automatically customise a FIP to an application.

1 Introduction

This paper explores adapting Flexible Instruction Processors (FIPs) [7] at run time. Previously we concentrated on compile-time issues and described the use of FIPs for the systematic customisation of instruction processor design and implementation. The features of our approach include: a modular framework based on "processor templates" that capture various instruction processor styles, such as stack-based or register-based styles; enhancements of this framework to improve functionality and performance, such as hybrid processor templates and superscalar operation; compilation strategies involving standard compilers and FIP-specific compilers, and the associated design flow; technology-independent and technology-specific optimisations, such as techniques for efficient resource sharing in FPGA implementations.

Our research helps designers to tune hardware implementations to the run-time characteristics of a system over a period of time. Factors such as keeping to power consumption or area constraints have to be traded-off with better performance and the ability to perform a wider range of functions. For instance, in an embedded communications device, we might wish to encrypt, compress and compute a checksum before transmission. A fast solution involves chaining three pieces of hardware that perform these functions together. However, it is likely that further control hardware will be required, for instance to negotiate the transmission protocols or control other peripherals. A flexible and small design is to utilise an instruction processor, and in order to achieve acceptable performance, we may have to run the processor at a high clock speed.

The FIP approach enables incorporation, at design time, of instructions that will accelerate encryption, compression and checksum generation, thereby improving performance by doing more per cycle, instead of increasing the clock speed. Our design-time system allows easy customisation of various styles of processors and executable code to be compiled for application-specific processors. A FIP can be further improved based on run-time characteristics. For example, an embedded device might be used in an area with high interference, requiring frequent retransmission of the data. In this case, the FIP could evolve to incorporate a more robust error correction code capable of detecting and correcting more error bits, while maintaining similar performance and size, by trading off general functionality for more targeted functionality.

Run-time reconfiguration is often used to gain either functionality or performance. Unfortunately, the time taken for reconfiguration often incurs performance penalty. The increasing density of FPGAs further exacerbate this penalty. The FIP approach provides a way to tune the frequency of reconfiguration. If long reconfiguration time is unacceptable, the application can be executed less efficiently with the instruction processor; otherwise we can reconfigure to a different FIP to run more efficiently.

To summarise, the main contributions of our approach include: (a) a run-time environment that manages the reconfiguration of the FIP so that it can execute a given application as efficiently as possible; (b) mechanisms to accumulate run-time metrics and analysis of the metrics to allow the run-time environment to request for automatic refinements; (c) techniques to automatically customise a FIP to an application.

The rest of the paper is organised as follows. Section 2 describes FIPs and motivates their run-time adaptation. Section 3 addresses point (a) and outlines our design and run-time flow. Section 4 deals with run-time optimisations and addresses (b). Section 5 introduces ideas for custom instruction generation which covers (c). We then use the AES algorithm as an example in Section 6.

2 FIPs and Run-Time Adaptation

Flexible Instruction Processors, or FIPs, consist of a processor template and a set of parameters [7]. Different processor implementations can be produced by varying the template parameters. FIP templates provide a general structure for creating processors of different styles: e.g. stack- or register-based processors. The processor templates can be further enhanced with features found in high performance processors, such as superscalar architectures and pipelining. Various Java Virtual Machines and MIPS style processors have been implemented.

Our FIP generation tool allows us to explore FIP designs with different speed, area and functionality trade-offs. FIPs provide a well-defined control structure that facilitates varying the degree of sharing of resources. FIPs also provide a systematic method for supporting customisation by allowing user-designed hardware to be accommodated as new instructions. By adding customisations or eliminating unused resources, we can tune an instruction processor to make use of available area efficiently and provide a good range of functionality.

Run-time adaptation allows us to further fine tune our FIPs to run-time changes by exploiting the upgradability of FPGAs. Our framework simplifies the process by providing a means of adapting a FIP and creating its executable code at compile and run time.

Hand-crafted implementations provide fast performance but once it has been manufactured and deployed, there is little scope for improvement. Instruction processors, on the other hand, provide a solution that is easily upgradable and flexible. However this flexibility is often provided at the expense of performance. FIPs provide a way to explore the design space between these two extremes. For instance, custom instructions can be included into a design to speed up their operation at the expense of increasing area and power consumption.

The ability for a FIP system to adapt to changing behaviour of applications is a powerful feature, but there are significant challenges involved. Such challenges include: (a) creating a collection of FIP designs at compile time or at run time, (b) managing these FIPs and (c) ensuring that performance of the system is not degraded by its flexibility. To meet these challenges, we develop an approach that contains the following components: (1) a design tool to facilitate the creation of customised FIPs at compile time, (2) a scheme to keep track of available FIP designs and machine code, (3) a run-time system to manage FIP state and configuration, (4) a metric used to decide if reconfiguration is a suitable option at a given time, (5) a specification of what run-time statistics are needed for refinement analysis, (6) a monitor to accumulate run-time statistics, (7) techniques for generating custom instructions.

Component (1) is outlined in [7]. Components (2) and (3) will be described in Section 3, while components (4), (5),(6) and (7) will be discussed in Section 4 and 5.

3 Design and Run-Time Flow

This section describes our proposed framework. Figure 1 shows the design flow and the run-time flow. The flow diagram is essentially divided into two parts: the design environment and the run-time environment.

Details of the design environment is covered in a previous paper [7]. During design time, source code is profiled to extract information that can help in customising the FIP. FIP generation includes producing domain-specific FIPs and the corresponding FIP-specific compiler to generate executable code. The design environment also produces the run-time environment. Users can determine the capability of the run-time environment at compile time. For example, the user can decide whether reconfiguration or automatic adaptation is required. This will affect the complexity of the run-time environment.

The run-time environment is responsible for the execution and management of the system and maintains a database of available FIPs, their associated executable code and a decision condition library, which contains information about

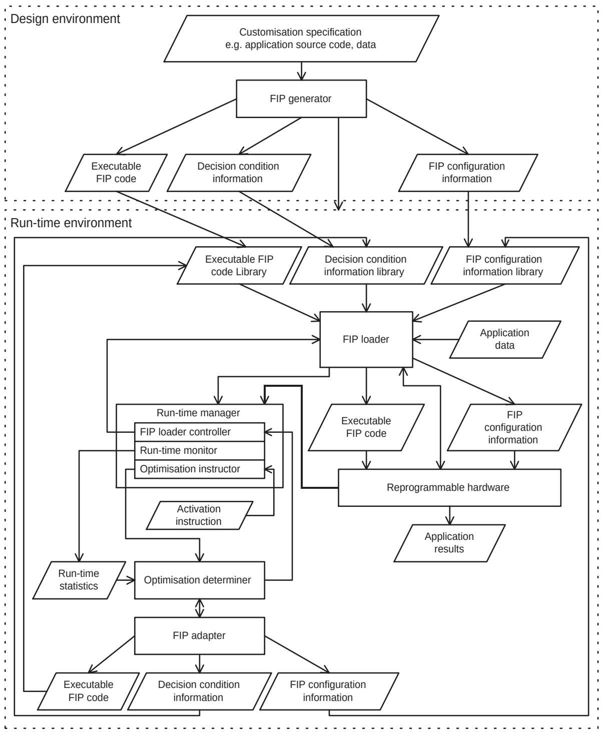

Fig. 1. Design flow and run-time flow. Optimisation analysis and automatic refinement steps are optional and are included when their overheads can be tolerated. The run-time environment is generated by the FIP generator. The run-time manager within the run-time environment can be run locally or on an external server.

when certain FIPs should be loaded. When an application is required, the FIP loader loads the appropriate executable from the code library and if necessary, a new FIP configuration. During execution, a FIP can keep track of run-time statistics, such as the number of times functions are called or the most frequently used opcodes. These run-time statistics can be sent to the run-time manager.

The run-time manager does not have to be local to the run-time environment. Run-time statistics can be uploaded to a central server where a run-time manager may reside. Based on run-time statistics, the optimisation determiner (Figure 1) can decide to dedicate more resources to functions that are used most frequently, by creating custom instructions for these functions. It can also suggest various optimisations, such as using faster operations for more frequently-used instructions, or changing instruction cache size or stack depth. Further issues regarding the optimisation determiner will be discussed in Section 4.2.

The FIP adapter creates a new FIP, executable code and decision condition information. The run-time environment's databases are updated and the new

FIP can be used next time the application is executed. Sections 4.3 and 5 will detail more of the reconfiguration analysis and automatic refinement process.

4 Run-Time Optimisation of FIPs

Several issues have to be addressed in order for a system to be able to dynamically optimise implementations for performance, and to request and perform upgrades via reconfiguration. We need to: (1) decide what statistics to collect, (2) set how frequently to sample the statistics collected, (3) collect the statistics from the application, (4) analyse the collected statistics and decide what to optimise, (5) perform the optimisation, (6) decide when to reconfigure the implementation.

Should data collection and analysis take place during run time, their impact on performance can be reduced if they run concurrently with the FIP. Such activities can be implemented in software, on a PC or on a programmable system-on-chip device.

4.1 Collection and Analysis of Data

There has been research addressing issue (1) and (2) above [8]. The frequency with which such statistics is collected, is pertinent. If statistics are recorded frequently, the available data will more accurately describe run-time characteristics, although this is at the expense of design area. If statistics are analysed frequently, the area required for the storage of these statistics can be reduced. However, frequent analysis may affect the performance of the FIP by either taking processing cycles to analyse the data, or consuming bandwidth involved in downloading the data off-line to be analysed. Furthermore, a short sampling of data before analysis may yield results that do not accurately reflect the run-time characteristics. A full analysis of these issues is beyond the scope of this paper.

Our FIP templates allow users to easily incorporate statistic monitors into their FIP designs. In our implementation, we collect information on the frequency of procedure calls, sampled over an application's run time.

Analysis of the data will depend on the statistics collected. The user could choose to monitor the frequency of use, for certain native or custom instructions. Based on this information we can, for example, increase the performance of a frequently-used multiplier circuit, while reducing the area and performance of the less frequently-used operations. We provide a level of flexibility in the optimisation analysis because of the domain specificity.

Custom instructions are created according to the results of the above analysis. The techniques used in creating custom instructions will be discussed in Section 4.2. Once custom instructions have been generated, the optimisation analyser (Figure 1) performs analysis similar to that done in the design environment. Optimisations based on analysis of congestion and constraints satisfaction of speed, area and latency can also be carried out.

This analysis is necessary because the original request for customisations may contain too many new custom instructions, and may not satisfy area or latency constraints. In that case, the optimisation analyser can decide to remove custom instructions, reduce the number of native opcodes supported, or downgrade the performance of opcodes that are not used frequently.

4.2 Reconfiguration

If configuration occurs too frequently, the overall performance of the system can suffer. The FIP approach provides a way to fine tune the frequency of reconfiguration by allowing an application to run less efficiently with the instruction processor if the reconfiguration time is unacceptable. A metric can be used to decide if it is beneficial to reconfigure. For instance, if we aim to improve performance, then a new, faster FIP should only be adopted if the reduction in run time is greater than the reconfiguration time involved in replacing the old FIP by the new one.

Consider a software function $f()$ which takes C_{sw} clock cycles with time for each cycle T_{sw}. As a custom instruction, it takes C_{ci} cycles with cycle time T_{ci}. The function $f()$ is called F times over the time period we are investigating, in this case one execution of the application. The reconfiguration time for the device is T_r, which includes time for collecting and analysing data. The time spent executing the software function (t_{sw}) can be shown to be $C_{sw}T_{sw}F$ and the time spent executing the custom instruction (t_{ci}) to be $C_{ci}T_{ci}F$. We define the reconfiguration ratio R as follows:

$$R = \frac{t_{sw}}{t_{ci} + T_r} = \frac{C_{sw}T_{sw}F}{C_{ci}T_{ci}F + T_r} \tag{1}$$

More generally, with n custom instructions, R becomes:

$$R = \left(T_{sw} \sum_{j=1}^{n} C_{sw,j}F_j \right) / \left(T_{ci} \sum_{j=1}^{n} (C_{ci,j}F_j) + T_r \right) \tag{2}$$

The point where $R = 1$ is the threshold: if $R > 1$, reconfiguration will be beneficial. Figure 2 demonstrates the effect of varying different parameters R on our designs. The horizontal axis measures the number of times an application is executed. The vertical axis shows R values. The curve with the circular disks corresponds to the base FIP, the R values of which is calculated by making $C_{ci}T_{ci} = C_{sw}T_{sw}$. The value of R for the base FIP will never reach 1, unless reconfiguration time is less than or equal to zero. The following discusses the effects of various FIP features on R.

Number of Custom Instructions. The general form of R, as shown in equation 2, shows that as we include more custom instructions, the reconfiguration threshold can be reached with fewer execution of the application. This is demonstrated by the two dashed curves in Figure 2. As more custom instructions are added and generic instructions are removed, the shape of the reconfiguration

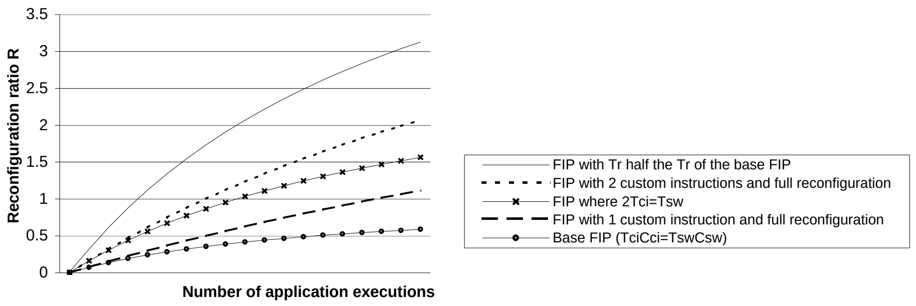

Fig. 2. This graph shows the effects of varying different parameters in the general reconfiguration ratio equation. When $R > 1$ reconfiguration should be attempted. Here we see the effects of increasing the number of custom instructions, reducing the cycle time of the custom instruction FIP and the reconfiguration time.

curve will tend towards that of a direct hardware implementation. The rate of this improvement is expected to decrease as more custom instructions are added.

Changing FIP cycle time. FIPs with custom instructions often require fewer clock cycles to complete an operation, but they may operate at a lower clock speed. The curve with crosses shows the reconfiguration ratio of a FIP with two custom instructions operating at half the clock speed of the base FIP. Although the performance is still better than the design with one custom instruction, the gain is not as much as one would expect.

Changing reconfiguration time. Changing the value of T_r may also affect R. The topmost curve shows a FIP with two custom instructions and half the reconfiguration time of the base FIP. Halving the reconfiguration time increases the initial gradient of the curve and reduces the number of application execution required to reach the threshold.

We have assumed the use of full reconfiguration until now. The reconfiguration time, T_r, can be rewritten as the product of t_r and n_r, the reconfiguration cycle time and the number of cycles needed to reconfigure the design. By utilising partial reconfiguration we can reduce n_r [8], and hence reduce the total impact reconfiguration has on R. n_r may also be reduced through improvements in technology and architectures that support fast reconfiguration through caches or context switches [10].

5 Generation of Custom Instruction

Compared to direct hardware implementations, instruction processors have the added overhead of instruction fetch and decode [7]. VLIW and EPIC architectures are attempts to reduce the ratio of fetches to execution. Customising instructions is also a technique for reducing the fetch and execute ratio to increase the performance of the instruction processor. The idea of incorporating custom instructions in an instruction processor has been reported [6]. Custom instructions are typically hand-crafted and incorporated at the processor instancing stage. While hand-crafted custom instructions provides the best performance, they are difficult to create, and require a skilled engineer with good knowledge of the system.

Opcode chaining can be used to reduce the ratio of the time spent on the control path and the data path of a FIP. Directly connecting up the data path of the sequence of opcodes that make up a procedure reduces the time spent on fetching and decoding instructions. Further, by converting a procedure call to a single instruction, the overhead of calling a procedure can be avoided; such overheads include preamble and postamble housekeeping routines, like storing program counters and shared registers, and refilling prefetch buffers.

Custom instructions have their own dedicated data path, making them suitable to exploit parallelism that may exist in opcodes; such as in instruction folding, up to four Java opcodes can be processed concurrently [3].

When creating custom instructions, we allow the available resources to grow as is needed to exploit as much data-level parallelism as possible. The introduction of a new custom instruction could introduce new possibilities for instruction-level parallelism in the FIP.

Other optimisations include device-specific optimisations such as using look-up table implementations. This exploits the high on-chip RAM capacity found in devices like Xilinx Virtex-E chips. Generalised look-up table based custom instructions can also be created, for instance; the same instruction can be used for cosine, sine and tangent operations, by reconfiguring the block RAM.

Streaming style instructions can also be incorporated. For example, an IP core that encrypts or decrypts a stream of data can be added as a custom instruction. Before a streaming instruction can be executed, it has to be initialised with relevant data such as the start position, length of the data stream and the location to write back to. The IP core is then activated via control channels that can be controlled by executing the custom instruction associated with the IP core. The IP core may be on a different clock domain from the FIP, in which case the relevant handshake protocols are used to communicate. Once the streaming instruction is activated, the FIP can then proceed to process other instructions. Buffers can also be used to store intermediate data if the core is not fully pipelined. This will allow the memory resources to be multiplexed between the core and FIP.

There are also optimisations related to reducing the overheads of run-time reconfiguration by reducing the amount of configuration storage required to store multiple designs and the time taken to reconfigure between these designs [9]. The following section will exemplify the process of creating a custom instruction.

	Java opcodes	Chained opcodes in custom instruction
a	iload_1	paramreg=TOS();
b	iconst_1 ishl	tempreg1=paramreg≪1;
c	iload_1 sipush 0x80 iand	tempreg2=param & 0x80
d	ifeq 0x9	if (tempreg2==0) jump 3;
e	getstatic 0x4f	tempreg2=0x1b;
f	goto 0x4	jump 2;
g	iconst_0	tempreg2=0;
h	ixor	tempreg2=tempreg2 ^tempreg;
i	int2byte	int2byte
j	ireturn	ireturn

Table 1. Sequential implementation of *FFmulx* in Java opcodes and chained opcodes. The Java opcode version takes 26.5 clock cycles on average plus another 4 cycles for the procedure preamble and post amble. The chained opcode version takes 8.5 clock cycles on average.

6 AES Example

We illustrate our approach by the AES (Rijndael) algorithm which is an iterated block cipher with variable block and key length. We have implementations of the cipher in two forms, a straight implementation (i) where we write code for the different component transformations of the cipher and a more efficient implementation (ii) where the component transformations are optimised into look-up tables.

In the first approach, the most frequently executed function, *FFmulx*, is a Galois matrix multiplication. The second column of Table 1 shows the Java opcodes required to implement the function. Depending on the outcome of the conditional branch `ifeq` opcode, this implementation takes between 25 to 28 clock cycles to execute.

The rightmost column of Table 1 shows the result of both opcode chaining and instruction folding. Opcode chaining involves storing intermediate results in temporary registers. By removing the need to push and pop values from the stack, the sequential structure imposed by the stack is eliminated. Next, instruction folding is applied. This allows several opcodes to be combined or folded into one instruction. In other words, several stack-based instructions are converted into one register-based instruction. Furthermore, since *FFmulx* is replaced by a single instruction, there is no longer a need to perform the preamble and postamble routines necessary for procedural calls. These techniques reduce the number of clock cycles in each application execution from 30 to 8.5 cycles.

Table 1 can be further optimised by identifying instructions that can execute in parallel, instruction b and c for instance. By introducing predicate registers, instructions d,e,g and h can be executed in parallel. Using this implementation, the original software *FFmulx* function has been optimised from 30 to 6 cycles, producing a 5-fold speedup. Similarly, the FIP (i) implementation is augmented by a single custom instruction involving direct connection of the data paths for the individual component transformations. This achieves an encryption of 128 bits of data with a 128 bit key in 99 cycles. Another implementation, FIP (ii), utilises lookup tables and achieves an encryption of 128 bits of data with a 128-bit key in 32 cycles.

Implementations	Cycles/Block	Hardware resources	Mbps/MHz	Flexible
Software[1] (C/C++)	340		0.4	Yes
FIP (i)	99	1770 Slices 2 BRAMs	1.3	Yes
FIP (ii)	32	1393 Slices 10 BRAMs	4	Yes
Hardware[5] (Spartan II 100-6)	11	460 Slices 10 BRAMs	11.5	No
Hardware[4] (Virtex-E 812-8)	1	2679 Slices 82 BRAMs	129.6	No

Table 2. Various AES implementations. Blocks are 128 bits with 128 bit keys. The C/C++ implementation runs on a 933MHz Pentium III. FIP implementations are written in Java and run on a sequential JVM implemented on a Spartan II 300E-6. The Spartan design is latency optimised and runs at 0.52 Gbps (45MHz). The Virtex-E design runs at a data rate of 7 Gbps (54MHz).

Table 2 compares different implementations of the AES algorithm. The fastest reported C/C++ implementation, by Gladman [1], achieves an encryption speed of about 350Mbps on a 933MHz Pentium 3. Running at 40MHz, FIP(i) encrypts at 51.7Mbps and FIP(ii) at 160 Mbps. FIP(ii) performs ten times better than software, in a Mbps per MHz comparison. Hand-placed hardware implementations provide good performance, but cannot be used for general computations. This flexibility has been compromised to improve performance. However these hardware implementations can be incorporated into FIPs as custom instructions, as outlined in Section 5.

The FIP approach provides a convenient compromise of trading off speed, flexibility and area. It provides the flexibility afforded by instruction processors and can be augmented with custom hardware to improve performance. Our proposed design-time and run-time system provides a means of customising these processors. It also provides a mechanism for these processors to adapt to environmental conditions, depending on usage patterns.

7 Concluding Remarks

We have described our work on run-time adaptive Flexible Instruction Processors, and the associated design and run-time environment. Run-time adaptability allows our system to automatically evolve and automatically refine the implementation to suit run-time conditions.Current and future work includes refining run-time statistics collection and reconfiguration strategies. The quality of the statistics gathered should allow the system to produce more optimal refinements. However the complexity of the monitoring hardware could affect FIP performance. We intend to obtain a better understanding of this trade-off between the quality of statistics and performance. We also continue to investigate strategies for reconfiguration, and to improve scalability [2] of our approach.

Acknowledgement. Thanks to A. Derbyshire and S. McKeever for their comments. The support of Celoxica Limited and the UK Engineering and Physical Sciences Research Council (Grant number GR/N 66599) is gratefully acknowledged.

References

1. B. Gladman. *Implementations of AES (Rijndael) in C/C++ and Assembler.* http://fp.gladman.plus.com/cryptography_technology/rijndael/.
2. J. A. Fisher. Customized instruction sets for embedded processors. In *Proc. 36th Design Automation Conference*, 1999.
3. H. McGhan and M. O'Connor. PicoJava: a direct execution engine for Java bytecode. *IEEE Computer.* October 1998.
4. M. McLoone and J. McCanny. Single-chip FPGA implementation of the Advanced Encryption Standard algorithm. In *Proc. FPL*, LNCS 2147. Springer, 2001.
5. N. Weaver and J. Wawrzynek. *Very high performance, compact AES implementations in Xilinx FPGAs.* http://www.cs.berkeley.edu/~nweaver/sfra/rijndael.pdf.
6. R. Razdan and M. D. Smith. A high-performance microarchitecture with hardware-programmable functional units. In *Proc. MICRO-27.* 1994.
7. S. Seng, W. Luk and P. Cheung. Flexible instruction processors. In *Proc. CASES.* ACM, 2000.
8. N. Shirazi, W. Luk and P. Cheung. Run-time management of dynamically reconfigurable designs. In *Proc. FPL*, LNCS 1482. Springer, 1998.
9. N. Shirazi, W. Luk and P. Cheung. Framework and tools for run-time reconfigurable designs. *IEE Proc.-Comput. Digit. Tech.*, May 2000.
10. S. Trimberger, D. Carberry and A. Johnson. A time-multiplexed FPGA. In *Proc. FCCM.* IEEE Computer Society Press, 1997.