Flexible instruction processors

Shay Ping Seng
Imperial College
Department of Computing
180 Queen's Gate,
London SW7 2BZ,
England
sps@doc.ic.ac.uk

Wayne Luk
Imperial College
Department of Computing
180 Queen's Gate,
London SW7 2BZ,
England
wl@doc.ic.ac.uk

Peter Y.K. Cheung
Imperial College
Department of Electrical & Electronic Engineering
Exhibition Road, London SW7 2BT, England
p.cheung@ic.ac.uk

ABSTRACT

This paper introduces the notion of a Flexible Instruction Processor (FIP) for systematic customisation of instruction processor design and implementation. The features of our approach include: (a) a modular framework based on "processor templates" that capture various instruction processor styles, such as stack-based or register-based styles; (b) enhancements of this framework to improve functionality and performance, such as hybrid processor templates and superscalar operation; (c) compilation strategies involving standard compilers and FIP-specific compilers, and the associated design flow; (d) technology-independent and technology-specific optimisations, such as techniques for efficient resource sharing in FPGA implementations. Our current implementation of the FIP framework is based on a high-level parallel language called Handel-C, which can be compiled into hardware. Various customised Java Virtual Machines and MIPS style processors have been developed using existing FPGAs to evaluate the effectiveness and promise of this approach.

General Terms

Design

Keywords

High-level synthesis, ASIP, Instruction processors

1. INTRODUCTION

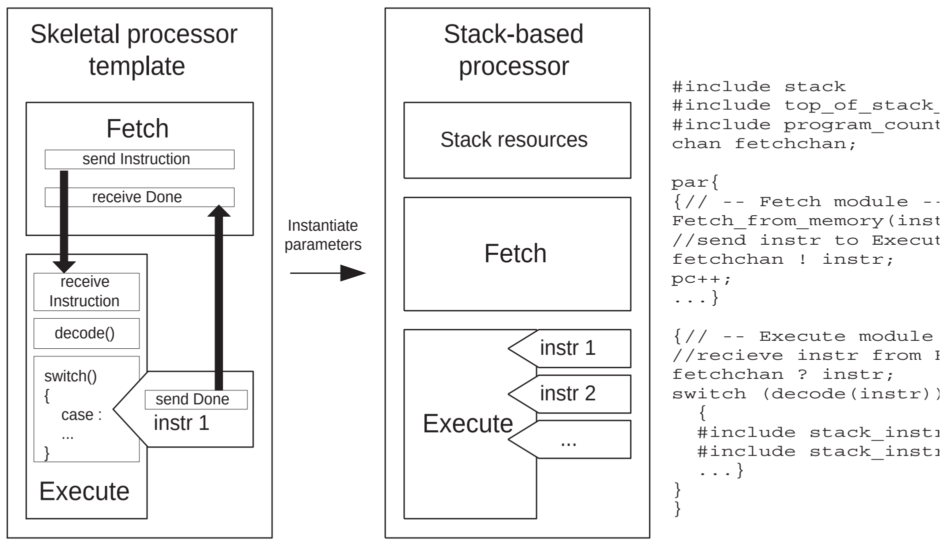
General-purpose instruction processors have dominated computing for a long time. However, they tend to lose performance when dealing with non-standard operations and non-standard data that are not supported by the instruction set formats [20]. The need for customising instruction processors for specific applications is particularly acute in embedded systems, such as cell phones, medical appliances, digital cameras and printers [7].

One way of supporting customisation is to augment an instruction processor with programmable logic for implementing custom instructions. Several vendors are offering a route to such implementations [2, 23, 27]. The processors involved are usually based on existing architectures, such as those from ARM, IBM and MIPS. These fixed instruction processor cores are interfaced to programmable logic, which provides the resources that implement a set of custom instructions for a given application.

Another way of supporting customisation of instruction processors is to implement them using existing FPGAs [8]. In this case, it is possible to customise the entire instruction processor at compile time [26] or at run time [6, 25]. While many of such instruction processors have been developed manually, Page [16] advocates an automatic method for instruction processor design and optimisation based on capturing the instruction interpretation process as a parallel program. His approach has been used in developing a number of instruction processors [5, 15, 24], although performance comparison with other techniques has not been reported.

Recent work on application-specific instruction processors (ASIPs) demonstrates the benefits of their customisation [9, 12, 22]. The trade-offs involved in designing application-specific processors differ from those of general-purpose processors. Similarly, trade-offs involved in ASIC implementations of application-specific processors differ from those of FPGA implementations.

This paper introduces the notion of a Flexible Instruction Processor (FIP) for systematic customisation of instruction processor design and implementation. The unique features of our approach include: (a) a modular framework based on "processor templates" that capture various instruction processor styles, such as stack-based or register-based styles; (b) enhancements of this framework to improve functionality and performance, such as hybrid processor templates and superscalar operation; (c) compilation strategies involving standard compilers and FIP-specific compilers, and the associated design flow; (d) technology-independent and technology-specific optimisations, such as techniques for efficient resource sharing in FPGA implementations. Inspired by Page's research [16], we have implemented the FIP framework using a parallel language called Handel-C, which can be compiled into hardware [3]. Various Java Virtual Machines and MIPS style processors have been developed to evaluate the effectiveness and promise of this approach, using existing FPGAs.



```
#include stack
#include top_of_stack_
#include program_count
chan fetchchan;

par{
{// -- Fetch module --
Fetch_from_memory(inst
//send instr to Execut
fetchchan ! instr;
pc++;
...}

{// -- Execute module
//recieve instr from F
fetchchan ? instr;
switch (decode(instr))
   {
   #include stack_instr
   #include stack_instr
   ...}
}
}
```

Figure 1: A skeletal processor template. The Fetch module fetches an instruction from external memory and sends it to the Execute module, which waits until the Execute module signals that it has completed updating shared resources, such as the program counter. This diagram also shows the instantiation of a skeletal processor into a stack processor, and the Handel-C description of the stack processor.

2. FIPS

A FIP consists of a processor template and a set of parameters. The template can be used to produce different processor implementations by varying the parameters for that template, and by combining and optimising existing templates. Our work is intended to provide a general method for creating processors of different styles. When compared with a direct hardware implementation, instruction processors have the added overheads of instruction fetch and decode. However, there are also many advantages:

- FIPs allow customised hardware to be accommodated as new instructions. This combines the efficient and structured control path associated with an instruction processor with the benefits of hand-crafted hardware. The processor and its associated op-codes provide a means to optimise control paths through optimising compilers. Conventional ad hoc sharing regimes are harder to optimise. Non-standard datapath sizes can also be supported.

- Critical resources can be increased as demanded by the application domain, and eliminated if not used. Instruction processors provide a structure for these resources to be shared efficiently, and the degree of sharing can be determined at run time.

- Our FIP approach enables different implementations of a given instruction set with different design tradeoffs. It is also possible to relate these implementations by transformation techniques [16], which provide a means of verifying non-obvious but efficient implementations.

- FIPs enable high-level data structures to be easily supported in hardware. Furthermore, they help preserve current software investments and facilitate the prototyping of novel architectures, such as abstract machines for exact real arithmetic and declarative programming [15].

Some of the above points will be explained later. Standard general processors are highly optimised and are implemented in custom VLSI technology. However, they are fixed in architecture and they only represent a point in a spectrum of possible implementations. FIPs provide a way of traversing the entire spectrum to create customised processors that are tuned for specific applications.

FIPs are assembled from a processor template with modules connected together by communicating channels. The template can be used to produce different styles of processors, such as stack-based or register-based styles. The parameters for a template are selected to transform a skeletal processor into a processor suited for its task (Figure 1). Possible parametrisations include addition of custom instructions, removal of unnecessary resources, customisation of data and instruction widths, optimisation of op-code assignments, and varying the degree of pipelining.

When a FIP is assembled, required instructions are included from a library that contains implementations of these instructions in various styles. Depending on which instructions are included, resources such as stacks, different decode units are instantiated. Channels provide a mechanism for dependencies between instructions and resources to be mitigated.

The efficiency of an implementation is often highly dependent on the style of the processor selected. Specialised processor styles, such as the Three Instruction Machine (TIM) [15], are designed specifically to execute a specific language, in this case a functional language. Even processor templates designed to be general, such as the stack-based Java Virtual Machine or register-based MIPS, are more efficient for different tasks. Hence, for a given application, the choice of the processor style is an important decision. Issues such as availability of resources, size of device and speed require-

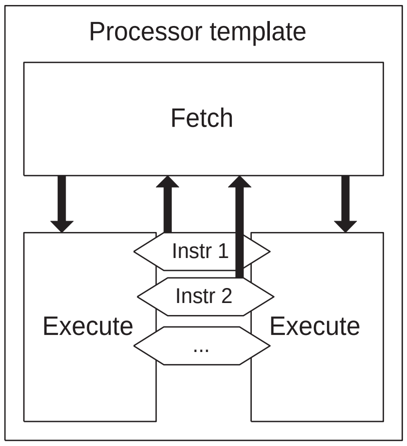


Figure 2: Superscalar or hybrid processor template. Several Execute modules are composed together using communicating channels that maintain the integrity of shared resources. This also provides a mechanism for creating hybrid processors. Different Execute modules may implement different styles of processors: stack-based or register-based styles.

ments are affected by the decision.

The FIPs in our framework are currently implemented in Handel-C version 2.1 [3], a C like language for hardware compilation. Handel-C has been chosen because it keeps the entire design process at a high level of abstraction, which benefits both the design of the processor and the inclusion of custom instructions. Handel-C also provides a quick way to prototype designs. Our focus is to provide FIPs that are customised for specific applications, particularly light-weight implementations for embedded systems. Using a high-level language, such as Handel-C, simplifies the design process by having a single abstract description and can provide a mechanism for demonstrating the correctness of the FIP [10, 16].

3. ENHANCING PROCESSOR TEMPLATES

The processor template framework introduced previously is sufficient for describing a basic instruction processor. Modern instruction processors contain many features that enhance their efficiency: examples include superscalar architecture, pipelining, interrupts and memory management units. This section outlines superscalar and hybrid processors as examples of more advanced processor designs. It also explains how our framework can exploit run-time reconfigurable capabilities of the implementation medium.

The key issue to resolve in supporting superscalar architectures in the processor template framework is that of scheduling. Scheduling of instructions can take place either at compile time or dynamically at run time. If scheduling takes place at compile time, the associated compiler for that processor should be responsible for scheduling. Otherwise, the Fetch module in a FIP should incorporate a scheduling algorithm.

In general, the Fetch module keeps track of resources used and issues multiple instructions to its array of Execution modules when appropriate. A superscalar processor provides a way for multiple resources to be used concurrently.

There is an opportunity to augment this framework with hybridisation. Hybrid processors are FIPs that can execute more than one style of instructions. Current complex processors can often be considered as hybrid processors. Intel x86 processors, for example, employ a register-based approach for most instructions, while floating-point instructions work on a stack. Hybridisation provides a systematic method that combine advantages of various styles into a FIP.

Consider the code for a hybrid FIP. It is well known that instructions for different styles of processors have different characteristics. For instance, register-based processors tend to have longer instructions and require more instructions in a program, compared with the instructions for a stack-based processor. Register-based instructions make it easier for parallelism to be exploited, while stack-based instructions tend to have more dependencies and often run sequentially. The possibility of combining multiple instruction formats into a single format for a hybrid FIP allows the user to trade-off speed with code size, which may be important for embedded devices with limited storage.

The binary description given to a hybrid FIP may contain instructions packed in the styles of different processors. The Fetch module of a hybrid FIP has to incorporate an additional level of decoding which determines an appropriate style, and channel the instruction to the corresponding Execute module.

For example, a hybrid processor may contain a MIPS and a TIM Execute module, composed in the same way as superscalar processors. This hybrid FIP can run MIPS code, but it is augmented by its ability to support functional languages.

It is also possible to produce multiple processors. In this case, different instruction streams can feed into each individual FIP that may communicate with one another via channels.

The modular design of FIPs provides a means for run-time reconfiguration to be incorporated. Conventional processors are designed to perform best in the most common case. Run-time reconfiguration allows FIPs to dynamically target applications.

Other advanced processor modules can be incorporated into the processor in a similar way. Modules for interrupt handling or memory management can be composed with the standard template using channels. Opportunities exist for these channels to be optimised and these are discussed in section 5. Speculative execution can also be supported by simultaneously executing both paths of a branch until a guard condition is determined. Pipelining of instructions can be composed with channels. Pipeline communications can be simplified if we know that a hazard will not arise. Profiling the application domain can provide this information. Section 4 will discuss more about profiling and our compilation strategy.

4. COMPILATION STRATEGY

The source code for an application can be in several forms: C, Java, data flow graphs, for example. The compilation from source code to hardware consists of two steps. A FIP has to be created and the code running on the FIP has to be generated.

Figure 3 shows the two possible compilation paths in our framework. The right path involves an available compiler to compile the source code. This compiler can be a standard compiler or a compiler created from a previous FIP

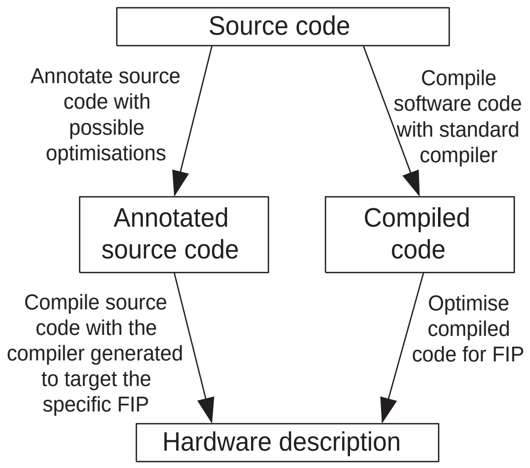

Source code
Annotate source code with possible optimisations
Compile software code with standard compiler
Annotated source code
Compiled code
Compile source code with the compiler generated to target the specific FIP
Optimise compiled code for FIP
Hardware description

Figure 3: Compilation path. Two paths are described. The left approach uses a specially generated compiler that targets the FIPs natively, the right approach uses a standard compiler.

design. Our design environment evaluates the compiled code to determine possible optimisations for the FIP. It also reorganises the code to exploit instruction-level parallelism and other optimisations. This is similar to the idea of Just-In-Time compilation (JIT) for Java virtual machines. The advantage of this strategy is that existing compilers can be used and precompiled code can execute on the processor. Since it is often difficult to identify possible optimisations in compiled code, this approach may yield a less optimum solution than using a FIP-specific compiler.

In the left path of Figure 3, the source code is annotated with relevant information, such as the frequency of the use of instructions, common instruction groups and shared resources. This step transforms standard source code into source code that includes specific information that is useful for optimisation of both the compiled code and the FIP. This step corresponds to the profiling step in Figure 4. A FIP specific compiler is then used to target this FIP. The advantage of this strategy is that no information is lost during the entire design flow, enabling the optimisation process to be as effective as possible.

A detailed description of this path is shown in Figure 4. The source code is profiled to extract information that can help to identify what styles of FIPs are suitable, or the user may specify the style directly. Recommendations may be made depending on the style of the source code. For instance, a stack-based processor is often a good choice for descriptions with a large number of small procedures in object-oriented style programming. The profiling step also collects information about the possible degree of sharing and frequency of certain combination of op-codes. At this stage, user constraints such as latency and speed can be specified.

Once a FIP template has been decided, the design flow is split into analysis and FIP instantiation. The analysis step involves analysing sharing possibilities and introducing custom instructions. Run-time reconfigurable possibilities are also explored. The FIP instantiation step involves technology independent analysis on congestion, scheduling, speed, area and latency. These two steps produce a domain specific FIP and the corresponding FIP specific compiler.

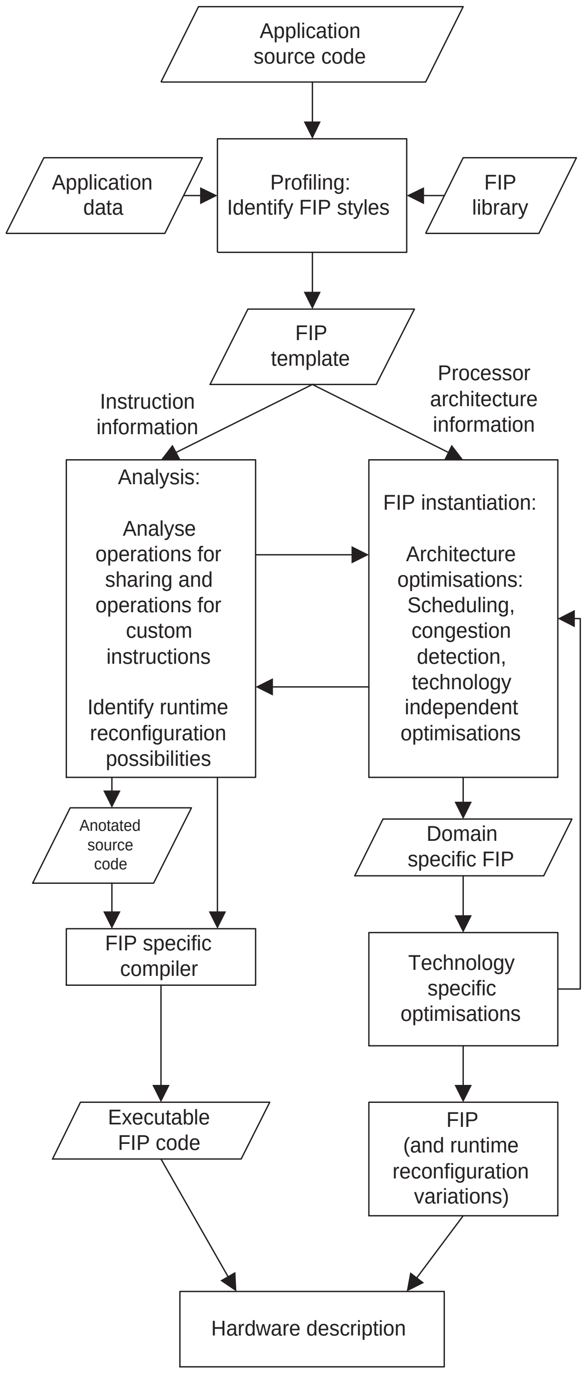

Application source code
Application data
Profiling: Identify FIP styles
FIP library
FIP template
Instruction information
Processor architecture information
Analysis:
Analyse operations for sharing and operations for custom instructions
Identify runtime reconfiguration possibilities
FIP instantiation:
Architecture optimisations: Scheduling, congestion detection, technology independent optimisations
Anotated source code
Domain specific FIP
FIP specific compiler
Technology specific optimisations
Executable FIP code
FIP (and runtime reconfiguration variations)
Hardware description

Figure 4: FIP design flow.

Next, the FIP goes through technology specific optimisations such as resource binding and constraint satisfaction. Following that, a FIP and its possible run-time reconfiguration variations can be produced.

Potentially, the design iterations between the blocks labelled "FIP instantiation" and "technology specific optimisations" in Figure 4 may be time consuming, because of the overheads associated with placement and routing. We are currently focusing on methods to reduce the design time by incorporating techniques to model technology specific constraints, such as routing and operator binding.

The FIP specific compiler is used to compile the annotated source code. A hardware solution is thus provided in the form of a domain-specific FIP and compiled FIP code, ready to be executed by the FIP.

Many of our current implementations are developed following the right path of Figure 3; these include the JVMs described in Section 6. We are also refining the methods and tools for the left path of Figure 3. At present, the analysis and FIP instantiation steps in Figure 4 are largely manual, and we are working on the automation of these steps to complete our tool chain.

5. FIP OPTIMISATIONS

Optimisations for FIPs can occur at two levels. Both the software and the processor can be optimised. We focus on the optimisation of the processors.

Advances in optimising compilers and instruction processor designs can be adapted for use in FIP architectures and compilers. We describe how these techniques can be modified for use with FIP systems. Optimisations can be categorised into four groups:

Technology independent

- Remove unused resources and instructions
- Customise datapaths and instructions
- Optimise op-code assignments
- Optimise channel communications between modules

Technology dependent (for FPGA implementation)

- Deploy special resources available: fast-carry chains, embedded memory
- Introduce congestion management to reduce delays due to routing

Processor style specific

- Processor type, such as JVM, MIPS, TIM
- Superscalar architecture, pipelining

Compiler specific

- Instruction level parallel scheduling
- Op-code re-ordering
- Loop unrolling, folding
- Predicated execution

Some of these optimisations have been presented in previous work [16]. The following describes custom instructions and technology dependent optimisations.

Direct hardware implementations of specific data paths can be incorporated into FIPs to be activated by custom instructions. This improves performance as it reduces the number of fetch and decode instructions. However, the more custom instructions, the larger the FIP. Hence, the improvement in speed is accompanied by an increase in size. The choice of the type and the number of custom instructions is important; this choice should also depend on how frequent a particular custom instruction is used. Section 6 evaluates the trade-offs in more detail.

Optimisations specific to certain processor styles are also possible. These are often related to device dependent resources. For example in a JVM, if multiple banks of memory exist, stack access could be enhanced so that the top two elements of a stack can be read concurrently. Device dependent resources can be exploited by using technology-specific hardware libraries [14] and vendor provided macros, such as Xilinx's Relationally Placed Macros [28] or Altera's Megafunctions [1].

In FPGAs, unlike ASICs, registers are abundant but routing can incur a large delay penalty as well as increase the size of a design. This feature of FPGAs places restrictions on template designs. Routing congestions occur when a resource is used extensively by many operations. Criteria such as the size of the resource or the routing density of neighbouring modules may also affect the routing of a FIP. We have identified three possible solutions. The first and simplest solution is to pipeline the routing. The second solution is to arrange the decoding network, which controls the activation of a resource, as a pipelined tree. This results in a shorter cycle time and a smaller logic to routing delay ratio, at the expense of larger area and more complex circuitry. The third solution is to replicate the resources. Resources should only be shared when it is profitable to do so. For example, instructions frequently require temporary registers for intermediate results, so sharing of these resources is inefficient. For shared operations, we are able to trade-off area and speed with latency. For instance, if the shared resource is a single-cycle multiplier, it can be replaced by several digit-serial multipliers, where parallel to serial converters are placed at locations to reduce the routing congestion. However, if the replicated resource is a shared storage, care needs to be taken to ensure the consistency of the state information. We are currently working on a method to automatically provide a combination of resources that will optimally trade-off congestion with latency and speed.

Using the first two techniques outlined above, we have optimised a JVM containing 40 instructions, all of which access the stack. Preliminary studies show promise: for instance, the optimisation has reduced the propagation delay of the critical path by half.

6. IMPLEMENTATION AND EVALUATION

This section describes various implementations of a FIP for the Java Virtual Machine, and compares their performance against software and ASIC implementations. The performance of a FIP for a MIPS style processor is also discussed.

6.1 Implementation

We have implemented a FIP based on the JVM specification [13]. Many parametrisations and optimisations have been investigated, including removal of unnecessary resources, customisation of data and instruction widths, optimisation of op-code assignments, and variation of the degree of pipelining. The first version of the JVM involves segregated re-

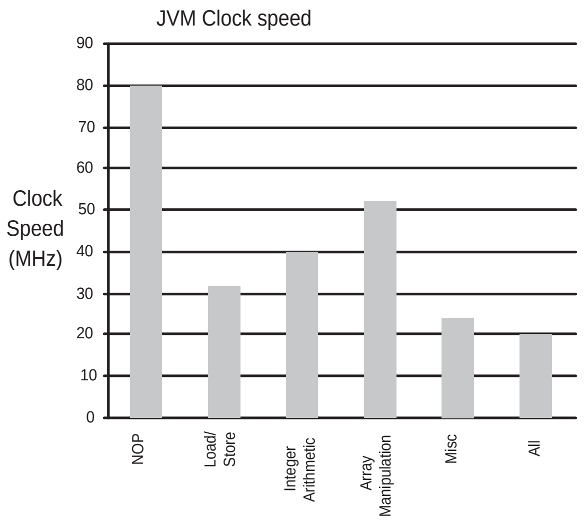


Figure 5: Registered performance of a JVM on a Virtex XCV1000 device with different available instructions.

sources that are shared. This provides good area utilisation at the expense of speed, because of routing congestion.

The second version of the JVM introduces two stages of pipelining and only shares irreplaceable resources, such as the stack and main memory. Stack-based processors are intrinsically sequential. Speed optimisation of the JVM tends to introduce parallelism that manifests as register style implementations of instructions.

The third version of the JVM incorporates deeper pipelines for certain instructions and 'register' style improvements such as having top-of-stack registers. The top-of-stack registers are replicated. Instructions can be read from different top-of-stack registers but are written back to the stack directly. Replicated registers are updated during the fetch cycle. Most instructions are processed by four pipeline stages, although certain instructions, such as the instruction for invoking functions, require deeper logic and their implementations have been partitioned into five or six pipeline stages. Routing has also been pipelined to reduce the effects of congestion.

The evolution of the three versions of the JVM demonstrates trade-offs between the possible parametrisations. Maximising sharing methods through conventional resource sharing may introduce significant routing overheads. Congestion management is necessary to identify the optimal degree of sharing: when the amount of routing is beginning to dominate the implementation medium.

Pipelining is useful for reducing clock cycle time. However, resources such as stacks may have operation dependencies that limit the amount of overlapping between instructions, and they introduce latency when pipelined. Many of our customised JVMs have been successfully implemented using the RC1000-PP system [4].

6.2 Efficiency of FIP-JVM

The third version of the JVM described earlier is used in the following evaluations. Figure 5 shows the theoretical upper bound for this implementation to be roughly 80MHz, when only the NOP instruction is supported. This shows that the fetch-decoding structure is reasonably efficient. The current program counter and data path size are 32 bits. The clock speed can be further increased by reducing the program counter size or improving the adder.

Devices	CaffineMark Score	Speedup	
Software JVM (Pentium II, 300 MHz)	502	1	
FIP JVM (Xilinx Virtex, 33MHz)	1019	2x	1
Pipelined FIP JVM (estimated)	3665	7x	3.5x
GMJ30501SB Java Processor (ASIC, 200 MHz)	13332	27x	13x

Figure 6: CaffineMark 3.0 Benchmark scores for different implementations of the JVM. The software JVM is Sun Microsystems' JVM version 1.3.0.

The performance of the FIP-JVM is compared with a JVM running on an Intel Pentium II machine and on a Java Processor [11] by Hyundai Microelectronics. The GMJ3050-1SB is based on the picoJava 1 core [21] from Sun Microsystems. The CaffineMark 3.0 [18] Java benchmark has been used to measure performance. The CaffineMark benchmark is a set of tests used to benchmark performance of JVMs in embedded devices. These include tests on the speed of Boolean operations, execution of function calls and the generation of primes.

Figure 6 shows the benchmark scores of our FIP processor together with software and ASIC. Our processor compares favourably with software, and a version with a deeper pipeline is estimated to run seven times faster. While the ASIC runs fastest, there are however two points to keep in mind. First, the ASIC processor is running at 200MHz, compared to our JVM at 33MHz. Second, the ASIC processor has fixed instructions, while we are able to incorporate custom instructions. The speedup provided by our FIP-JVM is expected to increase towards that shown by the ASIC processor as more custom instructions are added. In the following, we demonstrate the trade-offs of providing custom instructions.

Direct hardware implementations have been developed to manipulate a link list structure with separate circuits to support different access procedures, such as inserting a link and searching for a value. These direct implementations are clocked between 40MHz to 70MHz, and can be incorporated as data paths for custom instructions in the FIP-JVM.

Link lists may be used to organise emails or phone numbers in embedded systems such as cell phones. An insertion sort algorithm has been written using both the direct hardware approach and the FIP approach for comparison. The direct hardware implementation takes 2.3ms to sort a list of 100 links, while the FIP-JVM takes 6.4ms and the ASIC JVM is estimated to take 1ms. The insertion of a link into the list takes 22 Java instructions.

By including a custom instruction to insert a link, we can reduce the execution time to 5ms, since the single custom instruction takes 12 cycles to complete. There is a saving of 10 cycles, and 10 fetch and decode cycles saved per in-

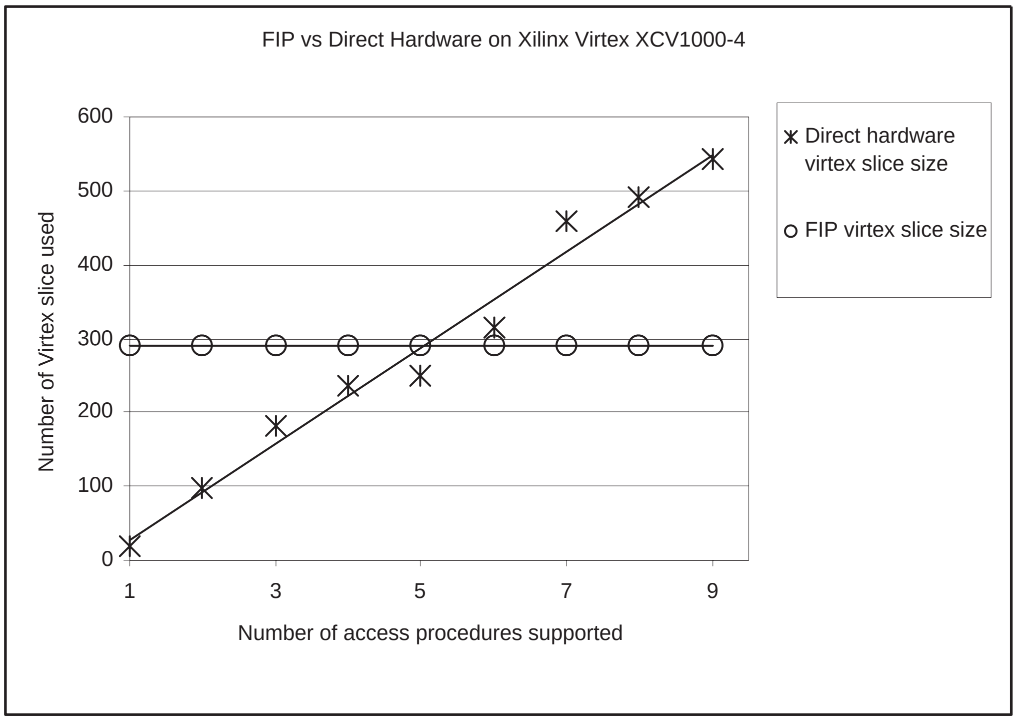


Figure 7: Device usage of MIPS style FIPs capable of running link list access procedures compared with direct hardware implementation of these procedures. This graph shows the comparison between the amount of FPGA resources used in a FIP and a direct hardware implementation without fetch and decode. This also shows the portability between a fully shared implementation (FIP) and a direct hardware implementation. The direct implementations can be clocked collectively at 39MHz while the FIP runs only at 30MHz. However, a link-list instruction may take tens of instructions to execute while only taking several cycles in direct implementation. Device independent results also show a similar trend, with one instruction requiring around 60 gates and registers, and 9 instructions taking 1706 gates and registers, compared to 901 gates and registers for the FIP implementation.

struction. Note that one can have a custom instruction that requires fewer cycles to execute, but the cycle time would be longer. If two custom instructions were added, the execution time is reduced to 3.1ms. However, the addition of custom instructions not only speeds up the application, but also increases the size of the FIP. The following section addresses this trade-off using another FIP.

6.3 Efficiency of MIPS style FIP

We have implemented a MIPS style processor which can be clocked at 30MHz. Two types of comparisons have been carried out. Device-independent comparisons look at the number of gates, registers and latches used. Device-dependent comparisons look at the number of Xilinx Virtex slices used. Figure 7 shows the trade-offs between a fully-shared FIP implementation and a direct hardware implementation. In general, direct hardware implementation executes in fewer cycles and can be clocked at a higher frequency than FIPs. An insert instruction, for instance, takes 12 cycles at 39MHz, compared to 22 cycles at 30MHz in FIP. The direct hardware implementation takes 2.3ms to sort a list of 100 links, while the FIP takes 7.1ms. However, the FIP uses 290 Virtex slices compared with the 460 slices used by the custom hardware. Figure 7 shows that the current FIP implementation is smaller than the direct hardware implementation for applications involving five or more access procedures. The cross-over point provides a means of estimating when it becomes unprofitable to include more custom instructions. As more custom instructions are added to the FIP, the cross-over point will shift upwards.

6.4 Evaluation summary

This study has shown that our initial assumptions are reasonable: the FIP structure is efficient and provides a good mechanism for resource sharing. The execution speed of the FIP can be improved by incorporating custom instructions, however this is at the expense of size. Furthermore, we can utilise device-independent results to estimate the number and type of custom instructions in a FIP. This provides a way to automate the optimisation of sharing. As sharing increases, the amount of routing congestion will also increase, since a larger number of instructions in a FIP may result in more congestion. Custom instructions reduce the number of instructions, hence increasing throughput and reducing congestion.

If the full instruction set of a particular architecture is required, it may be preferable to use an FPGA closely coupled to an embedded processor [2, 23, 27].

7. CONCLUDING REMARKS

We have described a framework for systematic customisation of instruction processors. The FIP approach enables rapid development of instruction processors by parametrising, composing and optimising processor templates. Either a standard compiler or a FIP-specific compiler can be used in the implementation process.

Current and future work includes extending our approach to cover other processor styles such as adaptive explicitly parallel instruction computing [17], refining existing tools such as those determining the degree of resource sharing, and integrating our tools with other related tools, for instance those for run-time reconfiguration [19]. We are also looking at ways to automatically select FIP styles and instantiate FIP parameters, and techniques to retarget and customise compiler and run-time systems for different applications.

8. ACKNOWLEDGMENTS

Many thanks to Matt Aubury, Steve Chappell, George Constantinides, Arran Derbyshire, Roger Gook, Steve McKeever, Henry Styles, Timothy Todman and Huy Vu for their comments and assistance. The support of Celoxica Limited, UK Engineering and Physical Sciences Research Council (Grant number GR/24366, GR/54356 and GR/59658), and Xilinx, Inc. is gratefully acknowledged.

9. REFERENCES

[1] Altera Corporation. *Megafunctions.* http://www.altera.com/html/mega/mega.html.

[2] Altera Corporation. *Excalibur Embedded Processor Solutions.* http://www.altera.com/html /products/excalibursplash.html.

[3] Celoxica. *Handel-C Production Information.* http://www.celoxica.com.

[4] Celoxica. *RC1000-PP from Celoxica.* http://www.celoxica.com.

[5] C. Cladingboel. Hardware compilation and the Java abstract machine. M.Sc. thesis, Oxford University Computing Laboratory, 1997.

[6] A. Donlin. Self modifying circuitry - a platform for tractable virtual circuitry. In *Field Programmable Logic and Applications*, LNCS 1482, pp. 199–208. Springer, 1998.

[7] J. A. Fisher. Customized instruction sets for embedded processors. In *Proc. 36th Design Automation Conference*, pp. 253–257, 1999.

[8] J. Gray. Building a RISC system in an FPGA. In *Circuit Cellar: The magazine for computer applications*, pp. 20–27. March 2000.

[9] M. Gschwind. Instruction set selection for ASIP design. In *7th International Workshop on Hardware Software Codesign*, pp. 7–11. ACM Press, 1999.

[10] J. He, G. Brown, W. Luk and J. O'Leary. Deriving two-phase modules for a multi-target hardware compiler. In *Proc. 3rd Workshop on Designing Correct Circuits.* Springer Electronic Workshop in Computing Series, 1996, http://www.ewic.org.uk /ewic/workshop/view.cfm/DCC-96.

[11] Helborn Electronics. *Java Processors.* http://www.helbon.co.uk.

[12] K. Küçükçakar. An ASIP design methodology for embedded systems. In *7th International Workshop on Hardware Software Codesign*, pp. 17–21. ACM Press, 1999.

[13] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification (2nd Ed.).* Addison-Wesley, 1999.

[14] W. Luk, J. Gray, D. Grant, S. Guo, S. McKeever, N. Shirazi, M. Dean, S. Seng and K. Teo. Reusing intellectual property with parameterised hardware libraries. In *Advances in Information Technologies: The Business Challenge*, pp. 788–795. IOS Press, 1997.

[15] C. J. G. North. Graph reduction in hardware. M.Sc. thesis, Oxford University Computing Laboratory, 1992.

[16] I. Page. Automatic design and implementation of microprocessors. In *Proc. WoTUG-17*, pp. 190–204. IOS Press, 1994.

[17] K. V. Palem, S. Talla and P. W. Devaney. Adaptive explicitly parallel instruction computing. In *Proc. 4th Australasian Computer Architecture Conf.* Springer Verlag, 1999.

[18] Pendragon Software Corporation. *CaffineMark 3.0 Java Benchmark.* http://www.pendragon-software.com/pendragon/cm3/index.html.

[19] N. Shirazi, W. Luk and P. Y. K. Cheung. Framework and tools for run-time reconfigurable designs. *IEE Proc.-Comput. Digit. Tech.*, 147(3), pp. 147–152, May 2000.

[20] H. Styles and W. Luk. Customising graphics applications: techniques and programming interface. In *Proc. IEEE Symp. on Field Programmable Custom Computing Machines.* IEEE Computer Society Press, 2000.

[21] Sun Microsystems. *PicoJava(TM) specification.* http://www.sun.com/microelectronics/picoJava.

[22] J. Teich and R. Weper. A joined architecture/compiler design environment for ASIPs. In *Proc. International Conference on Compilers, Architecture and Synthesis for Embedded Systems.* ACM, 2000.

[23] Triscend. *The Configurable System on a Chip.* http://www.triscend.com/products/index.html.

[24] R. Watts. A parametrised ARM processor. Technical report, Oxford University Computing Laboratory, 1993.

[25] M. Wirthlin and B. Hutchings. A dynamic instruction set computer. In *Proc. IEEE Symp. on Field Programmable Custom Computing Machines*, pp. 99–107. IEEE Computer Society Press, 1995.

[26] M. J. Wirthlin and K. L. Gilson. The nano processor: a low resource reconfigurable processor. In *Proc. IEEE Symp. on Field Programmable Custom Computing Machines*, pp. 23–30. IEEE Computer Society Press, 1994.

[27] Xilinx. *IBM and Xilinx team to create new generation of integrated circuits.* http://www.xilinx.com/prs_rls/ibmpartner.htm.

[28] Xilinx. *Relationally Placed Macros.* http://toolbox.xilinx.com/docsan /2_1i/data/common/lib/lib2_2.htm.