# CUSTARD - A Customisable Threaded FPGA Soft Processor and Tools

Robert G. Dimond, Oskar Mencer, and Wayne Luk

Department of Computing, Imperial College, London, England

**Abstract.** We propose CUSTARD — CUStomisable Threaded ARchitecture — a soft processor design space that combines support for multiple hardware threads and automatically generated custom instructions. Multiple threads incur low additional hardware cost and allow fine-grained concurrency without multiple processor cores or software overhead. Custom instructions, generated for a specific application, accelerate frequently performed computations by implementing them as dedicated hardware. In this paper we present a flexible processor and compiler generation system, FPGA implementations of CUSTARD and performance/area results for media and cryptography benchmarks.

## 1 Introduction

This paper introduces the Customisable Multi-threaded Processor: CUSTARD. CUSTARD is a parameterisable processor that combines support for multiple hardware threads and automatic instruction set customisation. We propose the customisable threaded architecture as an FPGA soft processor for System-on-a-Programmable-Chip (SOPC) applications with high performance requirements. Processor implementations are supported by our optimising C compiler that automatically generates custom instructions from C applications. We generate custom instructions by finding frequently occurring segments of computation that can be evaluated using the same hardware datapath.

This paper presents four main achievements:

1. CUSTARD, a customisable multi-threaded soft processor with parameterisations including number of threads, threading type, datapath bitwidths and custom instructions.
2. A C compiler that targets CUSTARD and automatically generates custom instructions.
3. A methodology to customise a multi-threaded processor for an application.
4. FPGA implementations of customised processors with area and performance results for five media and cryptography benchmarks.

Soft processors — instruction processors implemented within the reconfigurable fabric of an FPGA — provide control and data processing functions for a reconfigurable system. Soft processors provide three key advantages over a fully application specific datapath/state machine: Firstly, the capability to handle large applications. Secondly, a software design flow for rapid implementation and testing. Thirdly, soft processors allow a designer to build complete systems on inexpensive FPGAs that do not provide a hard-core processor such as ARM or PowerPC.

Customisable processors are emerging as a technique for optimising performance in embedded applications. Customising the processor instruction set to directly implement frequently performed operations can provide a performance gain for a small additional area required to support these instructions [12]. XTensa [13] and ARC[1] are examples of commercial customisable processors targeted at performance critical System-on-Chip applications. XTensa and ARC-tangent processors can be extended with custom instructions specified by the designer.

Recent research [3, 8] has demonstrated strategies for automatically partitioning applications into segments implemented using basic instructions (add, subtract, shift etc.) and segments implemented directly in hardware as custom instructions. We use a novel approach that finds frequently occurring program segments that can be computed using the same hardware datapath. These datapaths are integrated into the processor pipeline as custom instructions.

Our multi-threaded processor supports multiple contexts within the same processor hardware. A context is the state of a thread of execution, specifically the state of the registers, stack and program counter. Supporting threads at the hardware level brings two significant benefits. Firstly, a context switch — changing the active thread — can be accomplished within a single cycle, enabling a uniprocessor to execute two concurrent threads with little or no overhead. Secondly, a context switch can be used to hide latency where a single thread would otherwise busy-wait. A comprehensive survey of multi-threaded processors, their various configurations and advantages is available in [14].

The major cost of supporting multiple threads stems from the additional register files required for each context. Fortunately, current FPGAs are rich in block SRAM that could be used to implement large register files. Additional logic complexity must also be added to the control of the processor and the current thread must be recorded at each pipeline stage. However, the bulk of the pipeline and the functional units are effectively shared between multiple threads, so we should expect a significant area saving over a multi-processor configuration.

MicroBlaze [16] and Nios [2] are examples of existing soft processors provided by FPGA vendors. Neither has any multi-threading ability although embedded multi-threaded processors are emerging in the ASIC world, for example Tricore [11] and META [10]. The Java multi-threaded processor [15] is a research example that provides hardware support for the Java threads model. As such, CUSTARD is the first customisable multi-threaded processor for FPGA's.

This paper is organised as follows: Section 2 of this paper describes our methodology and tool flow. Section 3 describes the architecture and parameterisations of CUSTARD. Section 4 describes the compiler and software tools to support customisation. Section 5 presents quantitative results for five embedded benchmarks running on instantiations of CUSTARD. Finally, Section 6 provides a conclusion and suggestions for future work.

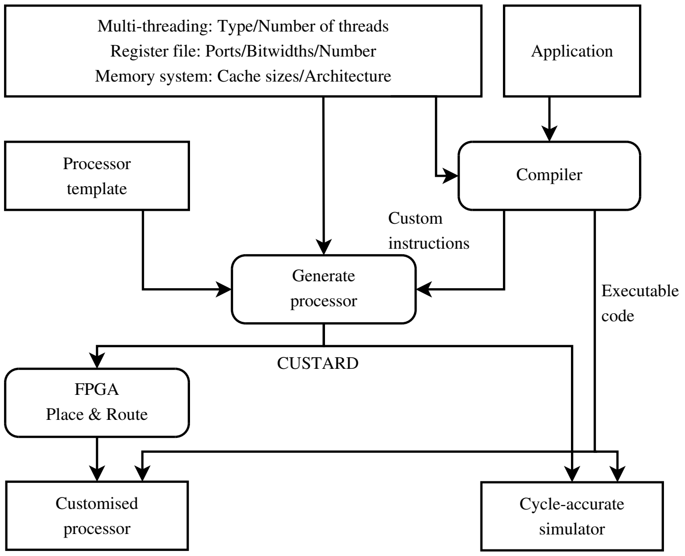


**Fig. 1.** Toolflow for a processor customised for a particular application.

## 2 Methodology and Tool Flow

Our methodology is to customise a multi-threaded processor to an application, using a combination of designer specified parameters and automatic design performed by a compiler. Figure 1 shows the overall tool flow from application to customised processor.

The inputs to the system are:

1. The application, specified in a high-level language such as C.
2. A parameterisable processor that serves as a template.
3. A set of user specified processor parameters.

The application code is analysed statically by the compiler. The compiler then generates a set of custom instructions to accelerate the application. Generated custom instructions are combined with the designer specified parameters to instantiate a synthesisable netlist for the processor.

The user parameters specify high level architectural features, most importantly the number of hardware threads supported by the architecture. Optimal values for these can be found by simulation or from the intrinsic requirements of the application. The results section of this paper sheds further light on the impact of parameterisation decisions.

The compiler identifies custom instructions via static analysis of the application code. Custom instructions implement frequently performed computations in dedicated datapaths. Replacing a sequence of instructions by a single custom instruction reduces the overhead of instruction fetch and the total number of cycles required for the computation.

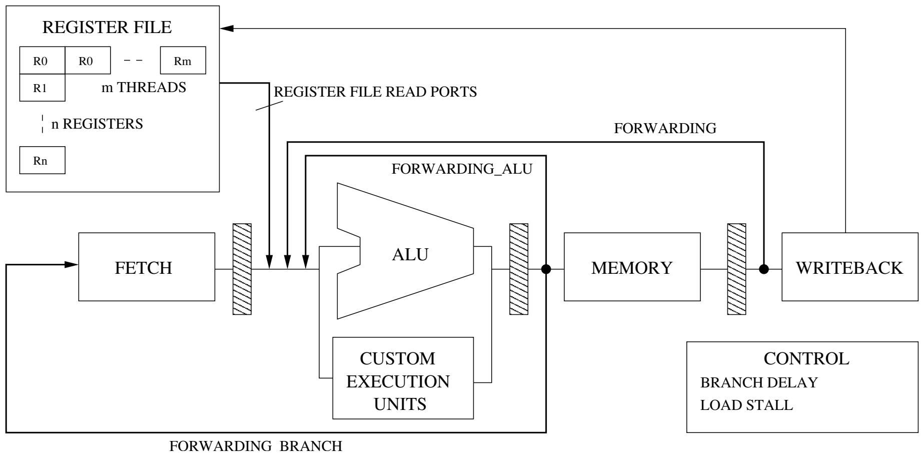


**Fig. 2.** CUSTARD micro-architecture showing threading, register file and forwarding network parameterisations.

## 3 Multi-threaded Architecture

We generate instances of Customisable Multi-threaded processors using a parameterisable model. The parameterisable model, Figure 2 both instantiates a synthesisable hardware description and configures our cycle-accurate simulator.

The base architecture is typical for a soft processor, with a fully bypassed and interlocked 4-stage pipeline. CUSTARD is in fact a load/store RISC architecture supporting the full MIPS integer instruction set. In addition, CUSTARD supports augmentation of the pipeline with custom instructions.

The detailed parameters are:

1. Multi-threading support
   - Number of threads: a power of 2
   - Threading type: Block (BMT) or Interleaved (IMT)
2. Custom instructions: Single-cycle, multi-cycle and pipelined
   - Custom datapaths at the execution stage of the pipeline
   - Custom memory blocks
3. Forwarding and interlock architecture
   - Branch delay slot: with or without
   - Load delay slot: with or without
   - Forwarding: enable/disable each forwarding path
4. Register file
   - Number of registers: a power of two
   - Number of register file ports: larger or equal to two
   - Bitwidth: 8,16,32

| Disable | Single/BMT | IMT 2 threads | IMT >= 4 threads |
|---|---|---|---|
| FORWARDING BRANCH |  | ✓ | ✓ |
| FORWARDING ALU |  | ✓ | ✓ |
| FORWARDING MEM |  |  | ✓ |
| BRANCH DELAY | ✓* | ✓ | ✓ |
| LOAD INTERLOCK | ✓* | ✓ | ✓ |

**Table 1.** Summary of forwarding paths (as shown in Figure 2) and interlocks that can be 'optimised away' for single threaded, block multi-threaded (BMT) and interleaved multi-threaded (IMT) parameterisations. * Optimising away this element in this configuration changes the compiler scheduler behaviour to prevent hazards.

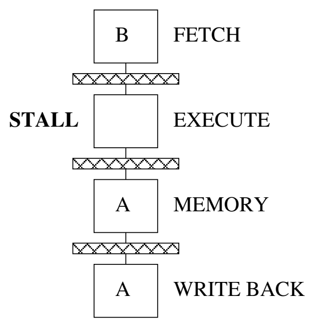
Two types of multi-threading are supported, block (BMT) and interleaved (IMT) multi-threading. Both types simultaneously maintain the context — the state of registers, program counter etc. — of multiple independent threads. The types of threading differ in the circumstances that context switches are triggered, illustrated for two threads in Figure 3.

Block multi-threading (BMT), as shown in Figure 3(a), triggers a context switch as a result of some run-time event in the currently active thread, for example a cache miss, an explicit 'yield' of control or the start of some long latency operation such as a custom instruction. When only a single thread is available, the BMT processor behaves exactly as a conventional single threaded processor. When multiple threads are available, any latency in the active thread is hidden by a context switch. The context switch is triggered at the execution stage of the pipeline, meaning that the last instruction fetched must be flushed and refilled from the new active thread. This results in the stall shown in Figure 3(a).

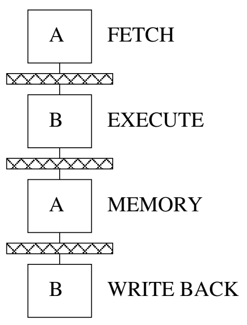
Interleaved multi-threading (IMT), as shown in Figure 3(b), performs a mandatory context switch every single cycle. This causes interleaved execution of the available threads. IMT permits simplification of the processor pipeline since, given sufficient threads, certain pipeline stages are guaranteed to contain independent instructions. IMT thus removes pipeline hazards and permits simplification of the forwarding and interlock network designed to mitigate these hazards. Our processor can exploit this by selectively removing forwarding paths to optimise the processor for a particular threading configuration.

Table 1 summarises customisation of the forwarding and interlock architecture for each multi-threading configuration. The forwarding paths, BRANCH, ALU and MEM are as illustrated in the pipeline diagram of Figure 2. The IMT columns show how elements of the forwarding and interlock network can be removed depending upon the number of available threads. For example, in the case of two threads, the ALU forwarding logic can be removed. When two IMT threads are available, any instruction entering the ALU stage of the pipeline is independent of the instruction leaving the ALU stage. Removing interlocks in certain situations (highlighted by *) constrains the ordering of the input instructions and so these parameters are made available to the compiler. Our compiler is able to adapt the scheduling of instructions based on these parameters.

Multiple contexts are supported by multiple register files which are implemented as dual-port RAM on the FPGA. Each register file access is indexed



(a) Block Multi-Threading (BMT). A context switch at the execution stage causes a stall.



(b) Interleaved Multi-Threading (IMT). A context switch occurs every cycle.

**Fig. 3.** Interleaved (IMT) and Block (BMT) Multi-threading modes supported by CUSTARD. These examples show interleaving of two hardware threads A and B.

by the register number and also the id of the thread that generated the access. Each register file is also parameterisable in terms of the number of ports and the number of registers per thread. Increasing the number of register file ports allows custom instructions to be selected by the compiler that take a greater number of operands.

We use the Handel-C [6] hardware description language to implement our parameterisable processor. Our Handel-C implementation of CUSTARD provides a framework for parameterisation of the processor together with a route to hardware. While the generated processor might be suboptimal compared to a hand optimised design, our interest is in rapid exploration of the design space and we expect that high-level architectural trends will be faithful to optimised designs.

## 4 Software Infrastructure for Customisation

Our processor is supported by both an optimising ANSI C compiler and a cycle-accurate simulator. The compiler automatically generates custom instructions for CUSTARD from a C application. The simulator provides rapid feedback of performance for an application running on an instance of the processor.

### 4.1 Optimising Compiler

Our compiler outputs MIPS integer instructions and custom instructions generated from the application. We generate custom instructions within the compiler using our novel Similar Sub-Instructions technique. Similar Sub-Instructions is outside the scope of this paper, we provide a brief introduction here but refer interested readers to a technical report [7] for detail.

The principle of Similar Sub-Instructions is to find instruction datapaths that can be re-used across similar pieces of code. We add these datapaths to our parameterisable processor and then update the decoding logic to map the new instructions to unused portions of the opcode space. The operation of Similar Sub-Instructions can be summarised in four steps:

1. Program statements are re-written as a set of *incidence matrices*. An incidence matrix is created for each binary commutative operator that appears at least once in the program (e.g. add, multiply, XOR).
2. A heuristic is used to merge incidence matrix columns. Each column represents an input to the matrix: merging columns occurs when the input can be computed using the same datapath.
3. A Breuer [5] factorisation process is used to select columns from each incidence matrix to maximise a heuristic 'figure of merit'. Custom instructions are generated to implement a 'sum' of the selected columns using the appropriate operator.
4. A final 'worthwhile check' is used to reject instructions that do not meet criteria for amount of computation performed within the instruction.

An incidence matrix is a representation for expressions of binary commutative operators. Each row of the matrix represents a 'sum' under a binary operator such as XOR or addition. The incidence matrix allows us to exploit the commutativity property when finding multiple opportunities to use an instruction. The merging (2) and factoring (3) steps actually select the regions of the program to be implemented as custom instructions. The 'worthwhile check' (4) stage prevents the compiler generating custom instructions that already exist in the processor basic instruction set.

### 4.2 Simulator

Our cycle-accurate simulator is based on the SimpleScalar[4] framework. The simulator is configured directly from the processor hardware description and simulates a parameterisable memory system. The simulator leverages the ability of the Handel-C compiler to output a high-level C++ model of the hardware. Conventionally, this model is compiled and executed on the host computer to provide behavioural simulation output. Our simulation flow links this software model to a processor simulation library that is, in turn linked to SimpleScalar. The simulation library intercepts bus requests for instruction fetches and memory operations and passes these to SimpleScalar. SimpleScalar handles memory system simulation and various housekeeping functions, such as binary loading.

## 5 Results

Our compilation and simulation framework is sufficiently complete to allow application level benchmarks to be executed. To obtain indicative results from a

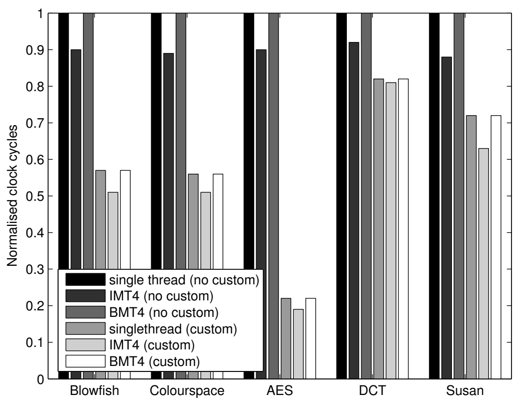


**Fig. 4.** Normalised number of execution cycles required for six CUSTARD configurations running five benchmarks.

compiler and processor very early in their development cycles, we select benchmarks from the MiBench [9] suite that were sufficiently self-contained. We use six benchmarks in total that cover two important application domains. From image/video processing: colourspace conversion, laplace edge detection, SUSAN edge detection and Discrete Cosine Transform (DCT). From cryptography: the Advanced Encryption Standard (AES) and Blowfish. All benchmarks are compiled 'out of the box', i.e. without hand optimisation or tailoring for the architecture or compiler.

Briefly, the custom instructions identified by the compiler are: For Blowfish, 3 instructions that do byte selection and a table lookup. DCT, one table lookup and multiply instruction. Colourspace, one table lookup instruction and a three input, single output logical instruction. AES, one instruction that XOR's four table lookups. SUSAN, one table lookup instruction.

Figure 4, show the execution cycle counts for the five benchmarks (Blowfish, Colourspace, AES, DCT and SUSAN) running on a CUSTARD processor implemented on a Xilinx XC2V2000. Figure 5 shows the FPGA area utilisation in Xilinx slices and Figure 6 shows the maximum clock rate, as reported by the timing analyser. Each bar in the graph corresponds to a particular processor customisation. For each benchmark we present results for three threading configurations: 1. Single threaded. 2. Block Multi-Threaded with four threads (BMT4). 3. Interleaved Multi-Threaded with four threads (IMT4). We show results with, and without automatically generated custom instructions for that benchmark.

1. The IMT4 and BMT4 configurations add only 28% and 40% area respectively to the single threaded processor but allow four threads to execute concurrently with no software overhead. In comparison, a multi-processor system would require four single processors plus arbitration logic.

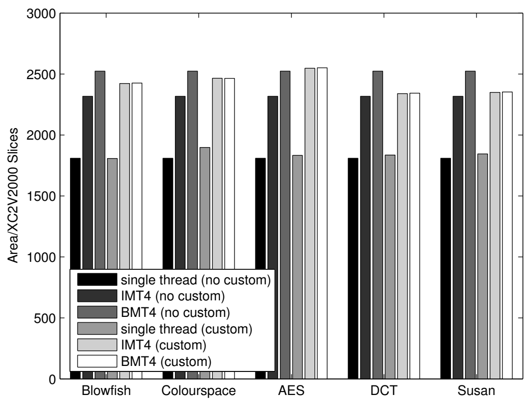


**Fig. 5.** Required area in terms of XC2V2000 slices from a total of 21,504 for CUSTARD configurations.

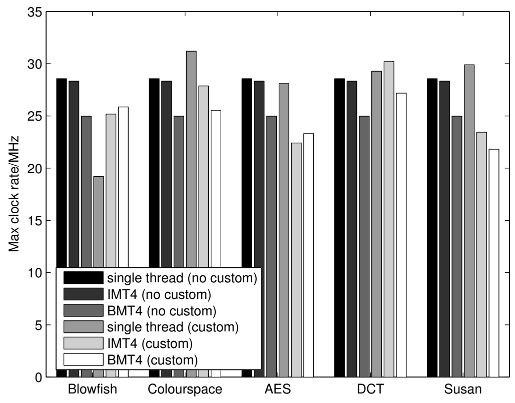


**Fig. 6.** Maximum clock frequency for CUSTARD configurations as reported by Xilinx timing analyser.

2. Custom instructions give a significant performance increase, an average of 72% with a small area overhead above the same configuration without custom instructions, an average of only 3%. CUSTARD accelerates AES by 355%.
3. The optimisations to the forwarding network in the IMT processor without custom instructions lead to a smaller area requirement and allow a higher maximum clock rate compared to the equivalent BMT processors. This trend is generally preserved across the custom instruction implementations, although there are some anomalies such as the AES and Blowfish IMT4 processors having a lower clock rate that the equivalent BMT4 processors.

## 6 Conclusion and Future work

We have presented CUSTARD, a customisable multi-threaded FPGA soft processor. We present a methodology for customising CUSTARD processors to an application and an implementation of the software infrastructure required to support our methodology. To evaluate CUSTARD, we present performance and area results for FPGA implementations of processors running important benchmarks from media and cryptographic domains.

Possible directions for future work are investigating multi-processor configurations of CUSTARD or combinations of customisable processors and hardware accelerators. In addition, we hope to extend the compiler to identify larger program segments as custom instructions to exploit greater parallelism at this level.

## References

1. ARCtangent extensible processor. http://www.arccores.com.
2. Altera. *Custom Instructions for the Nios Embedded Processor*, September 2002. Application Note 118.
3. Kubilay Atasu, Laura Pozzi, and Paolo Ienne. Automatic application-specific instruction-set extensions under microarchitectural constraints. In *Proc. DAC 2003*, June 2003.
4. Todd M. Austin. *A User's and Hacker's Guide to the SimpleScalar Architectural Research Tool Set*. http://www.simplescalar.com.
5. Melvin A. Breuer. Generation of optimal code for expressions via factorisation. *Communications of the ACM*, 12(6):333–340, June 1969.
6. Handel-C language reference manual. http://www.celoxica.com, 2001. Version 2.1.
7. Robert Dimond. Custard: a custom threaded architecture and tools. Master's thesis, Imperial College, 2004. http://www.doc.ic.ac.uk/ rgd00/reports/rdtr1.pdf.
8. F. Sun et al. Custom-instruction synthesis for extensible-processor platforms. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(2):216–228, February 2004.
9. Matthew R. Guthaus et al. MiBench: A free, comercially representative embedded benchmark suite. In *Proc. IEEE 4th Annual Workshop on Workload Characterisation, Austin, TX.*, December 2001.
10. META - RISC/DSP core with hardware multi-threading. http://www.metagence.com.
11. Erik Norden. A multithreading extension for low-power, low-cost applications. Embedded Processor Forum 2003, 2003.
12. S.P.Seng, W.Luk, and P.Y.K.Cheung. Runtime adaptive flexible instruction processors. In *Proc. Field-Programmable Logic and Applications*, 2002.
13. http://www.tensilica.com.
14. Theo Ungerer, Borut Robic, and Jurij Silc. A survey of processors with explicit multithreading. *ACM Computing Surveys*, 35(1):29–63, March 2003.
15. Panit Watcharawitch and Simon Moore. JMA: the java-multithreading architecture for embedded processors. In *International Conference on Computer Design (ICCD)*. the IEEE Computer Society, September 2002.
16. Xilinx. *MicroBlaze Hardware Reference Guide*, March 2002. http://www.xilinx.com.