Application-specific customisation of multi-threaded soft processors

R. Dimond, O. Mencer and W. Luk

Abstract: A multi-threaded microprocessor with a customisable instruction set, CUStomisable Threaded ARchitecture (CUSTARD), is proposed. CUSTARD features include design space exploration and a compiler for automatic selection of custom instructions. Custom instructions, optimised for a specific application, accelerate frequently performed computations by implementing them as dedicated hardware. Field programmable gate array implementations of CUSTARD are evaluated using media and cryptography benchmarks, and commercial MicroBlaze processor is compared. As low as 28% area overhead for four interleaved threads and up to 355% speedup over a processor without custom instructions are demonstrated.

1 Introduction

This paper introduces the Customisable Threaded ARchitecture: CUSTARD. It is a parameterisable processor that combines support for multiple hardware threads and automatic instruction set customisation. We propose the customisable threaded architecture as a soft processor – a processor implemented on the reconfigurable fabric of a field programmable gate array (FPGA) – for System-on-a-Chip 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.

Soft processors are frequently employed for control and data processing functions in a reconfigurable system. Soft processors provide three key advantages over a fully application-specific datapath/state machine: First, the capability to handle large applications. Second, a software design flow for rapid implementation and testing. Third, 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 [1]. XTensa [2] and ARC [3] are examples of commercial customisable processors targeted at performance critical System-on-Chip applications. XTensa and ARCtangent processors can be extended with custom instructions specified by the designer.

IEE Proceedings online no. 20050177

doi:10.1049/ip-cdt:20050177

Paper first received 1st November 2005 and in revised form 4th February 2006

The authors are with the Department of Computing, Imperial College, 180 Queens Gate, London SW7 2RH, UK

E-mail: rgd@doc.ic.ac.uk

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 bring two significant benefits. First, a context switch – changing the active thread – can be accomplished within a single cycle, enabling a uniprocessor to interleave execution of independent threads with little or no overhead. Second, 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 Ungerer *et al.* [4]. The major cost of supporting multiple threads stems from the additional register files required for each context. Fortunately, current FPGAs are rich in block static random access memory (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.

This paper presents five main achievements:

1. CUSTARD, a customisable multi-threaded processor with parameterisations including number of threads, threading type, datapath bitwidths and custom instructions.
2. An optimising C compiler, based on the CoSy framework [5], that targets CUSTARD and automatically generates custom instructions using our novel 'Similar Sub-Instructions' technique.
3. A cycle-accurate simulator, built using the SimpleScalar toolset, for evaluation of processor customisations.
4. A methodology to customise a multi-threaded processor for an application.
5. FPGA implementations of customised processors, running in hardware, with area and performance results for five media and cryptography benchmarks. We compare execution time with the commercial MicroBlaze soft-processor.

This paper is an extended version of one that appeared in Field Programmable Logic and Applications 2005 [6].

2 Related work

MicroBlaze [7] and Nios [8] 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 application specific integrated circuit (ASIC) world, for example, Tricore [9] and META [10]. The Java multi-threaded processor [11] is a research example that provides hardware support for the Java threads model. As such, CUSTARD is the first customisable multi-threaded processor for FPGAs. Current CUSTARD processors are automatically generated and not hand-optimised, allowing us to rapidly characterise the design space at the expense of some optimality. By comparing performance with the MicroBlaze processor, we demonstrate that the performance benefits of customisation can more than compensate for lack of manual optimisation.

The SPREE [12] system provides exploration of soft-processor design space. Processors are constructed from a library of register transfer level (RTL) components using a specialised structural description. In comparison with SPREE, CUSTARD provides a significantly larger design space, most importantly the potential for custom instructions, whereas SPREE provides efficient synthesis and fine-grained optimisation of the architecture once an instruction set has been fixed.

Atasu *et al.* [13], Brisk *et al.* [14] and Sun *et al.* [15] have 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.

Atasu *et al.* [13], formulate custom instruction selection as an integer linear programming problem that minimises the schedule length of a basic block subject to microarchitectural constraints. Identical instructions are identified and reused by a subgraph isomorphism test. This algorithm finds the optimum combination of custom instructions for a basic block, although reuse of instructions is considered only after their selection is complete and no access to memory can occur within a custom instruction. Sun *et al.* [15] propose a template-extraction technique that is driven by synthesis and simulation results to select a combination of instructions that achieve maximum speedup. This approach ensures high-quality results at the expense of a long compile time that is acceptable for an ASIC flow but would dominate the FPGA design cycle. Brisk *et al.* [14] demonstrate a heuristic clustering method that generates custom instructions to implement frequently occurring structures in the code.

Any of the above methods could be used to find instruction set extensions for the CUSTARD processor. However, we use a novel method that is specialised for a soft-processor implementation by supporting two enhancements over the previous work. First, to take advantage of the abundance of block random access memory (RAM) on modern FPGAs, we permit a limited form of memory access within custom instructions. Our compiler identifies read-only segments – often used as look-up-tables in software – that are placed in dedicated block RAM tightly coupled to the custom instruction. This is similar to the approach adopted in [16] except that we automate the process entirely with no need for manual verification. Secondly, our technique is able to reuse custom instructions across non-identical pieces of code. While conceptually similar to clustering techniques such as [14], our approach is based on methods from computer algebra that take into account commutative relationships to maximise reuse. For FPGA implementations, routing overheads exacerbate the clock rate penalties of adding each custom instruction, and so maximising reuse enables a large portion of the application to be accelerated while maintaining clock rate.

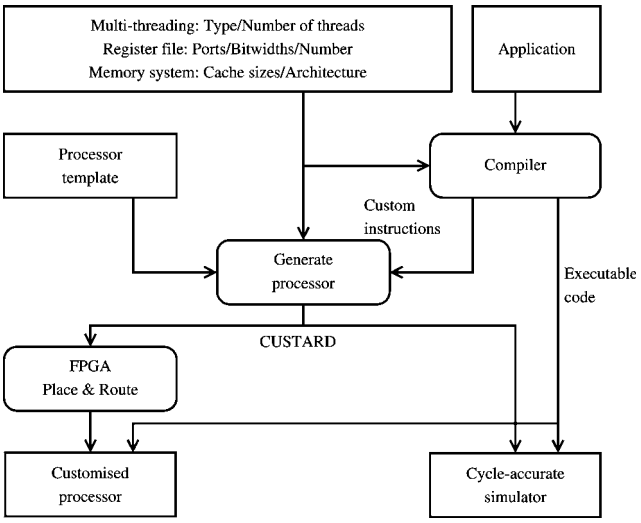

Fig. 1 *Toolflow for our processor customised for a particular application*

3 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. Fig. 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.

Our compiler performs standard optimisations and static analyses on the application C code. The compiler then generates a set of custom instructions to accelerate the application. We combine generated custom instructions with 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.

Our compiler identifies custom instructions automatically to accelerate frequently performed computations. We implement custom instructions as dedicated datapaths that can be single, multi-cycle or pipelined. 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.

4 Multi-threaded architecture

We generate instances of customisable multi-threaded processors using a parameterisable model. The parameterisable model (Fig. 2) instantiates a synthesisable hardware description and configures our cycle-accurate simulator.

The CUSTARD 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

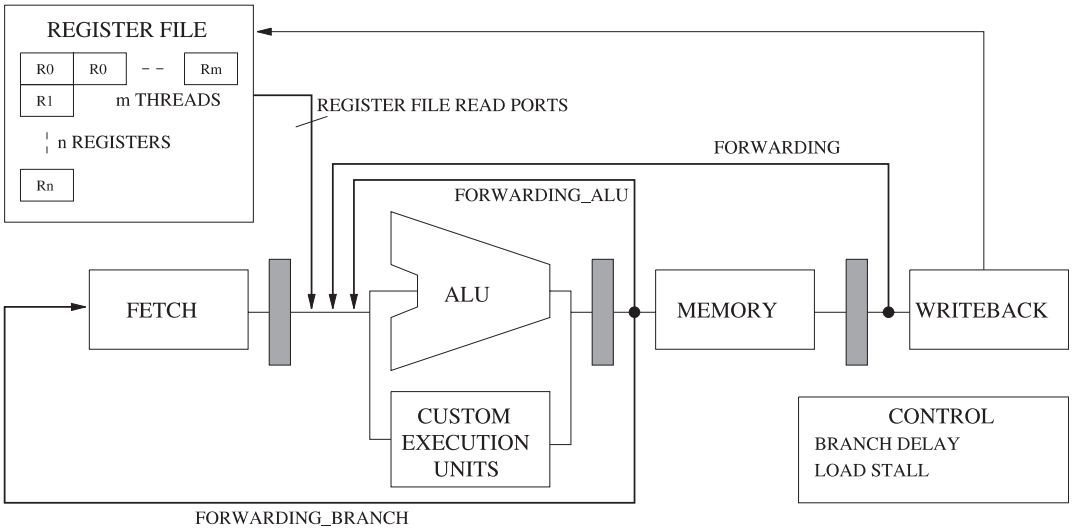

Fig. 2 *CUSTARD micro-architecture showing threading, register file and forwarding network parameterisations*

architecture supporting the full MIPS integer instruction set. CUSTARD supports augmentation of the pipeline with custom instructions using spare portions of the MIPS opcode-space.

The detailed parameters are:

1. Multi-threading support:
   - number of threads: a power of 2;
   - threading type: block (BMT)/interleaved (IMT).
2. Custom instructions: single/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 2;
   - number of register file ports: larger than or equal to 2;
   - Bitwidth: 8, 16, 32.

We support two types of multi-threading, block (BMT) and interleaved (IMT) multi-threading. Both types simultaneously maintain the context – the state of registers, program counter and so on – of multiple independent threads. The types of threading differ in the circumstances that context switches are triggered, illustrated for two threads in Figure 8.

BMT, as shown in Fig. 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 Fig. 3*a*.

IMT, as shown in Fig. 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 as, 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.

We do not discuss in detail the possible utilisation of the multi-threading features provided by CUSTARD. Our focus is on the hardware implementation and optimisation, compiler support and combination with custom instructions.

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 Fig. 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

Table 1: Summary of forwarding paths (as shown in Fig. 2) and interlocks that can be 'optimised away' for single-threaded, BMT and IMT parameterisations

Disable	Configuration	BMT	IMT	IMT
	No. of threads	≥ 1	≥ 2	≥ 1
FORWARDING_BRANCH			✓	✓
FORWARDING_ALU			✓	✓
FORWARDING_MEM				✓
BRANCH DELAY		✓*	✓	✓
LOAD INTERLOCK		✓*	✓	✓

*Optimising away this element in this configuration changes the compiler scheduler behaviour to prevent hazards

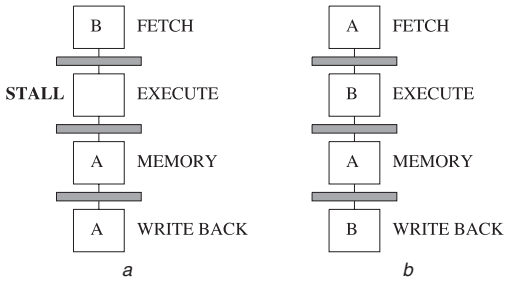

Fig. 3 *IMT and BMT multi-threading modes supported by CUSTARD*

These examples show interleaving of two hardware threads A and B

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 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 currently use the Handel-C [17] 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. Although high-level synthesis incurs some overheads, we compare to the optimised MicroBlaze processor and demonstrate that our designs are competitive.

5 Optimising compiler

Our compiler outputs MIPS integer instructions and custom instructions to optimise CUSTARD for the application. We generate custom instructions within the compiler using our novel Similar Sub-Instructions technique. Fig. 4 shows the flow through our compiler with the custom instruction finding stage highlighted. Prior to finding custom instructions, a pre-optimisation stage performs standard source-level optimisations together with loop unrolling to expose loop parallelism. After custom instructions have been selected, custom and base instructions are scheduled to minimise pipeline stalls. This scheduling stage is parameterisable to support the microarchitectural changes afforded by the CUSTARD multi-threading modes.

The result of compilation comprises hardware datapaths to implement custom instructions and software to execute on the customised processor. We add the custom instruction datapaths to the CUSTARD processor and then update the decoding logic to map the new instructions to unused portions of the opcode space. We assemble and link the software portion using modified versions of the GNU binary tools.

5.1 Similar Sub-Instructions

The object of Similar Sub-Instructions is to find instruction datapaths that can be reused across similar pieces of code. Essentially, we cluster operators from the directed acyclic graph (dag) of each basic block in the compiler intermediate representation. The clustering is guided by a figure-of-merit

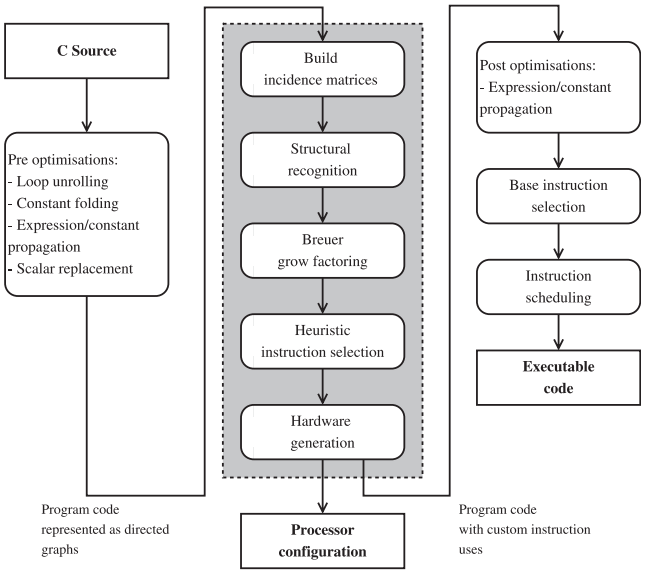

Fig. 4 *Complete compiler flow with the Similar Sub-Instructions (find reusable instructions) stage shaded*

Pre-optimisations run before custom instruction finding to expose regularity and simplify the code
Post-optimisations tidy up the output before code generation
Base instruction selection is a standard technique that selects base instructions for code not implemented within a custom instruction
The final scheduling pass orders base and custom instructions to maximise performance

(FoM) heuristic that attempts to maximise the proportion of code accelerated by custom instructions within an area constraint. The algorithm, inspired by techniques from computer algebra, allows commutativity to be taken into account to maximise reuse of instructions.

We do not give a full description here (interested readers should refer to Dimond *et al.* [18]) but hope to give an insight into the four main steps of the algorithm:

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). At this stage, read-only arrays in memory are also identified.
2. A heuristic is used to merge incidence matrix columns. Each column represents an input to the matrix: merging of columns occurs when the input can be computed using the same datapath.
3. A Breuer [19] 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.

$J = A \oplus (B >> 6)$
$K = C \oplus D \oplus ((E >> 5) + 1)$
$L = (F >> 6)$

a

$\oplus$	A	(B>>6)	C	D	((E>>5)+1)	(F>>6)
J	1	1	0	0	0	0
K	0	0	1	1	1	0
L	0	0	0	0	0	1

b

Fig. 5 *Incidence matrix with input chains example*
The $\oplus$ symbol denotes any binary commutative operator

Fig. 5 shows an example incidence matrix for a system of expressions. Each binary commutative operator is represented by a matrix, while unary expressions, constants and variable inputs form 'chains' at the inputs to the matrix. Building a set of incidence matrices from a directed acyclic graph of an expression is straightforward, where one matrix is required for each type of binary operator. We build a matrix set for each function, so that custom instructions can be reused across any expression within the same function.

We merge columns of incidence matrices in order to reuse custom instruction datapaths. Columns are merged subject to two constraints. First, overlapping of non-zero elements is avoided as this corresponds to reuse of hardware within the same custom instruction. Second, the chains (or inputs) to each column must be 'similar' so that they can be implemented using the same datapath resources. For our implementation, the similarity constraint is that the arithmetic operators must be of the same type, but constant inputs are permitted to be different. These constants correspond to immediate values in the final custom instruction.

We use a factoring process to select columns of the final matrix to implement as custom instructions. Adding an additional column moves more computation into hardware but can reduce the number of matrix rows that can be accelerated by the custom instruction. Our intention is to maximise the amount of computation performed in hardware while minimising the number of instructions required, so a small custom instruction that is reused many times has the same value as a large instruction used only once. We capture this behaviour using an FoM shown in (1). The factored rows parameter captures the reuse of the instruction, whereas weight gives an estimate for the amount of computation performed on each instruction input. We greedily maximise this heuristic, adding the best column at each stage (calculated by (2)).

$$\text{FoM(factor)} = \sum_{\text{column} \in \text{factor}} \text{FoM(column)} \quad (1)$$

$$\text{FoM(column } c) = \text{Weight(chain)} \times \text{factored rows} \quad (2)$$

Our compiler generates hardware datapaths for the factored columns in a matrix, and then inserts special nodes for custom instruction inputs and outputs into the intermediate representation. At the code-generation stage, these nodes are simply matched by the automatically designed custom instructions, whereas conventional instructions are selected for the rest of the graph.

5.2 Implementation

Fig. 6 provides an overview of our infrastructure, in particular, the source of results that we use to evaluate processor configurations.

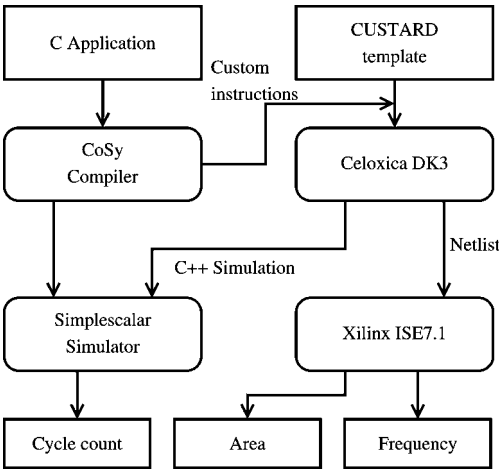

Fig. 6 *Implementation of our complete system, showing the source of each metric used to evaluate our processor designs*

We implement the compiler using the CoSy [5] system from ACE Associated Compiler Experts. The CoSy framework provides an extensive suite of standard compiler optimisations together with a robust framework for constructing compilers for novel architectures. We enable a full suite of optimisations to ensure that any speedup achieved from customisation could not also be achieved by standard optimisation. For example, a custom instruction that chains two instructions into a single instruction has no value if one or both instructions could be eliminated by conventional peephole optimisation.

CoSy provides an automated back-end generator that we use to generate the instruction selection, register allocation and scheduling components. We support custom instructions by inserting pure function calls – which have identical semantics to a custom instruction use – into the internal representation prior to selection of base MIPS instructions. We provide a separate instruction scheduler for each multi-threading configuration of the processor that re-orders instructions to maximise performance. Our scheduler accommodates custom instructions by mapping each one to a template instruction that has identical latency.

Instruction set customisation incurs a negligible increase in compile time, of <0.1 s (or $<20\%$ of compile time) for all benchmarks except DCT. DCT causes a 2.2 s increase in compile time.

6 Cycle-accurate simulator

Our cycle-accurate simulator is based upon the SimpleScalar [20] framework. The simulator is configured directly from the processor hardware description and simulates a parameterisable memory system.

SimpleScalar provides a default simulation model sim-outorder that is highly parameterisable to capture a wide spectrum of traditional processor designs. In particular, dynamically scheduled (superscalar) processors can be simulated with various microarchitectural and memory system configurations. SimpleScalar uses the C preprocessor to decouple the instruction set being simulated from the microarchitecture model, allowing portability across a number of targets such as Alpha and PISA (a superset of MIPS). Despite this flexibility, it is necessary to make significant changes to SimpleScalar in order to model the combination of custom instructions and multi-threading.

To support CUSTARD processors, we replace the simulation kernel of SimpleScalar with a CUSTARD specific

version. This kernel is generated automatically from the Handel-C description of the processor, including custom instructions and the multi-threading configuration. SimpleScalar tools provide functions such as memory system simulation, binary loading and the general infrastructure.

To generate the simulation kernel, we leverage the ability of the Handel-C compiler to output a high-level C++ model of the processor that can be compiled and linked to a custom SimpleScalar kernel. We provide the libraries to link the processor model with the software simulation at bus transaction level. Directly compiling the processor model from Handel-C allows us to generate simulator and hardware from the same description, ensuring 100% fidelity with the actual hardware. In addition, we are able to achieve an acceptable simulation speed (compared to RTL simulation) that is around 30 times slower than sim-outorder from SimpleScalar. Although higher performance could be achieved by writing a new version of sim-outorder specific to CUSTARD, our approach enables fast verification and development of the processor model.

7 Results

Our compilation and simulation framework is sufficiently complete to allow application-level benchmarks to be executed. To obtain indicative results from a compiler and processor very early in their development cycles, we select benchmarks from the MiBench [21] suite that are 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.

For multi-threaded processors, we do not demonstrate the performance advantage of zero overhead hardware context switching above a conventional single-threaded processor. This benefit is covered in existing literature [4] and is dependent upon the frequency that context switches are performed rather than the benchmark itself. Instead, we run as many instances of the benchmark as there are hardware contexts on each processor to demonstrate the effect of our architectural optimisations and the benefit of latency hiding.

Table 2 provides a summary of the exact custom instructions generated by the compiler for each benchmark and the cycle latency of their ASAP scheduled implementations. In addition, the number of uses of each instruction is shown to demonstrate the extent that instructions are reused. Each use corresponds to a distinct instruction in the assembly code, not the dynamic reuse caused by looping behaviour. Fig. 7 shows by example how the pseudocode description of the AES custom instruction corresponds to the hardware implementation.

Fig. 8 shows the execution cycle counts for the five benchmarks (Blowfish, Colourspace, AES, DCT and SUSAN) running on a CUSTARD processor implemented on a Xilinx XC2V2000. Fig. 9 shows the FPGA area utilisation in Xilinx slices and Fig. 10 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.

We examine the timing analysis reports to confirm that clock rate variations are not due to randomisation in the place and route tools. For BMT and single-threaded processors, the critical path includes the forwarding network: necessary for single-cycle latency of basic instructions (FORWARDING_ALU in Fig. 1). The critical path in the IMT processors is the ALU pipeline stage.

We make the following four observations from the results:

1. The IMT4 and BMT4 configurations add only 28 and 40% area, respectively, to the single-threaded processor but allow interleaved execution of four threads with no software overhead.
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%.

Table 2: Summary of the custom instructions automatically generated for each benchmark

Benchmark	Custom instruction(s) (input registers r_0-r_3, immediate value imm_0)	No. uses	Latency (cycles)	BRAM (bytes)
Blowfish	$LUT(r_0 \gg 24) + LUT(r_1 \gg 16)$	2	1	1024
	$LUT((r_0 \gg 8)\&255)$	2	1	
Colourspace	$((r_0 \gg 8)\&0xFF)\|(r_1 \,\&\, 0xFF00)\|$ $((r_2 \ll 8)\&0xFF0000)$	1	1	32
DCT	$LUT(r_1) + r_2 * (r_0 \ll 8)$	65	2	64
	$LUT(r_1) + r_2 * ((r_0\&255) - 128)$	65	2	
Edge Detect	$LUT(r_0 + 1 + imm_0)$	3	1	64
SUSAN	$LUT(r_0)$	31	1	516
AES	$LUT(r_0) \wedge LUT(r_1 \gg 8) \wedge$ $LUT(r_2 \gg 16) \wedge LUT(r_3 \gg 24)$	64	1	1024

Inputs r_0-r_3 are allocated to registers from the general purpose file. $LUT(a)$ = table lookup from dedicated block RAM address a. 'No. uses' demonstrates the extent of reuse by showing the number of times the instruction is used in the benchmark assembly code. 'Latency' is the number of execution cycles required before the output is available to the forwarding network or in the register file

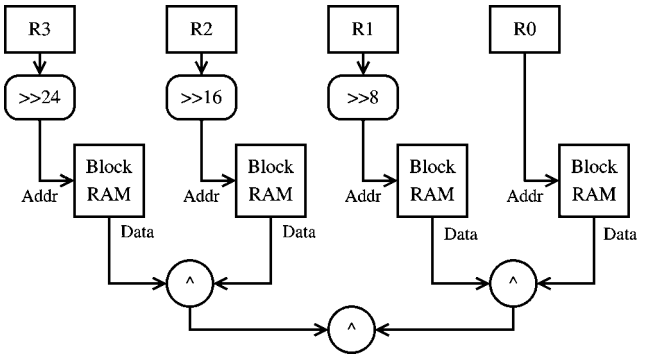

Fig. 7 *Implementation of the AES custom instruction from Table 2*

$LUT(r_0) \wedge LUT(r_1 \gg 8) \wedge LUT(r_2 \gg 16) \wedge LUT(r_3 \gg 24)$

3. The IMT processors without custom instructions provide a higher maximum clock rate than both BMT (41% higher) and single-threaded (5% higher) processors. The number of cycles is also reduced by an average of 10%.
4. The IMT processors hide pipeline latencies by tightly interleaving independent threads. We anticipate that the relatively low (10%) latency improvement is caused by the short latency of the custom instructions generated (Table 2), at most two cycles in every case. It was not possible to build longer latency instructions within the register file port constraints, so we expect that deeply pipelined processors or floating point custom instructions are needed to create latencies long enough for significant benefit in this area. However, the IMT processors do allow a higher maximum clock rate by removing the forwarding logic around the ALU.

Table 3 gives comparative 'wall clock' execution time results for single-threaded CUSTARD processors and the MicroBlaze soft-processor. The MicroBlaze results are for a processor running at 100 MHz, with (Div + Shifter) and without (Baseline) the optional hardware divider and barrel shifter options. The CUSTARD results are for a single-threaded processor, running at the maximum frequency reported by the tools, with and without custom

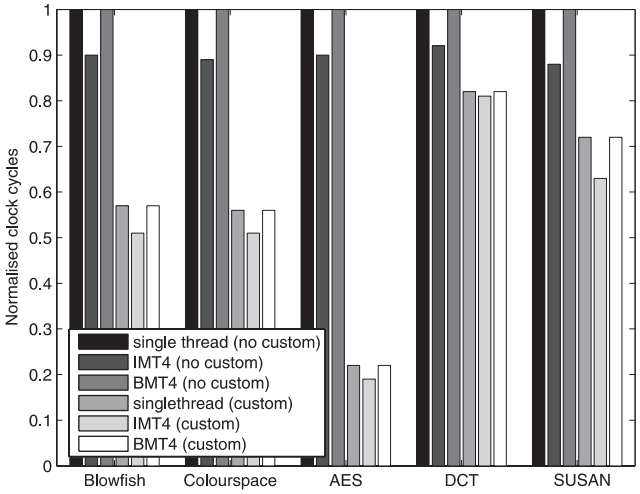

Fig. 8 *Normalised number of execution cycles required for six CUSTARD configurations running five benchmarks*

IMT and BMT processor results show overall throughput for four independent computations

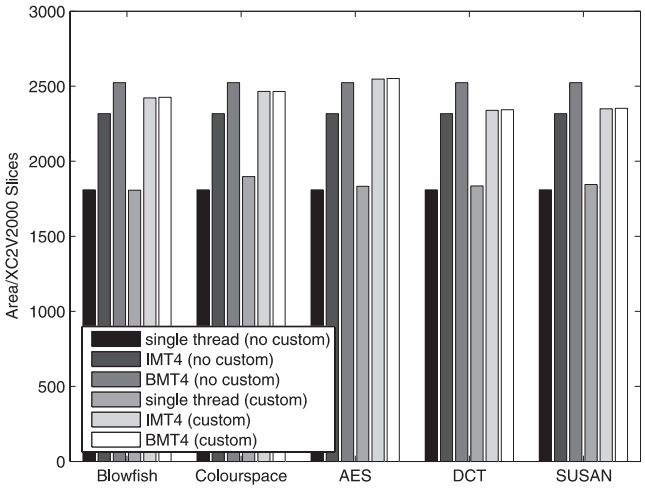

Fig. 9 *Required area in terms of XC2V2000 slices from a total of 21 504 for CUSTARD configurations*

instructions. The fairest comparison is with the second MicroBlaze processor that has the optional divide unit and barrel shifter enabled (Div + Shifter), because all CUSTARD processors presented include these features.

It is difficult to directly compare such different processors with different memory systems, although we do configure the same cache size for both processors (8k data and 2k instruction). Our intent is to show that CUSTARD processor performance is realistic and competitive with a highly optimised processor such as MicroBlaze.

Without custom instructions, the CUSTARD processor is slower than the MicroBlaze, with execution time increases of 61% (Colourspace) to 6% (AES). However, the single-threaded CUSTARD processors with custom instructions are significantly faster for the three benchmarks, with speedups as high as 3.93 (AES) with an average of 2.41 times across all benchmarks. Of course, MicroBlaze consumes only half the FPGA area of CUSTARD and provides a number of significant additional features for operating systems and I/O that we do not support. However, we assert that in terms of pure data processing CUSTARD processors are competitive across the benchmark set. This is a significant result for our automatically customised processor to beat the highly optimised MicroBlaze on performance, even at half of the clock rate.

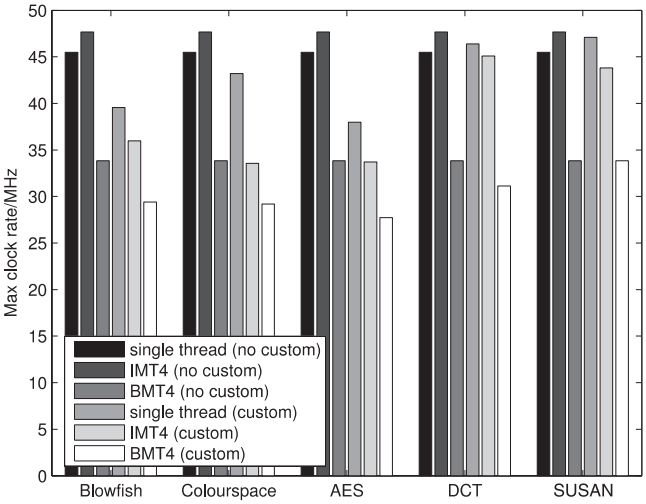

Fig. 10 *Maximum clock frequency for CUSTARD configurations as reported by Xilinx timing analyser*

Table 3: Comparison between CUSTARD and Xilinx MicroBlaze 4.00a wall clock execution time

Benchmark	MicroBlaze time, μs		CUSTARD time, μs		CUSTARD	Speedup CUSTARD/MB
	Baseline	Div + Shifter	No CI	With CI	MHz with CI	Div + Shifter
Blowfish	204 127	34 194	37 523	23 512	39.55	1.45
Colourspace	1828	448	720	424	43.21	1.06
AES	936	135	143	34	37.98	3.93
DCT	18 862	11 939	16 197	4952	46.39	2.41
SUSAN	671	642	694	201	47.10	3.19

CUSTARD processor running at maximum clock frequency on XC2V2000 (shown for processors with custom instructions) and MicroBlaze at 100 MHz. MicroBlaze results shown with and without the optional barrel shifter and divide unit. Both processors configured with 2k instruction and 8k data cache

8 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.

In our future work, we intend to investigate additional strategies for building custom instructions, in particular, instructions that allow pipelined execution of loops. In addition, we would like to explore multi-processor configurations of CUSTARD.

9 Acknowledgments

We gratefully acknowledge the support of ACE Associated Compiler Experts, Celoxica, the EPSRC and Xilinx. The comments of both FPL and IEE CDT reviewers were helpful in preparing the final manuscript.

10 References

1 Seng, S.P., Luk, W., and Cheung, P.Y.K.: 'Runtime adaptive flexible instruction processors'. Proc. Field-Programmable Logic and Applications, 2002
2 Xtensa extensible processor. http://www.tensilica.com
3 ARCtangent extensible processor. http://www.arccores.com
4 Ungerer, T., Robic, B., and Silc, J.: 'A survey of processors with explicit multithreading', *ACM Comput. Surv.*, 2003, **35**, (1), pp. 29–63
5 ACE Associated Computer Experts bv. CoSy compiler development system. http://www.ace.nl
6 Dimond, R., Mencer, O., and Luk, W.: 'CUSTARD–a customisable threaded FPGA soft processor and tools'. Proc. Field Programmable Logic and Applications (FPL), August 2005, pp. 1–6
7 Xilinx. MicroBlaze Hardware Reference Guide, March 2002. http://www.xilinx.com
8 Altera. 'Custom instructions for the Nios embedded processor', September 2002. Application Note 118
9 Norden, E.: 'A multithreading extension for low-power, low-cost applications (tricore processor)'. Embedded Processor Forum Presentation, 2003
10 META – RISC/DSP core with hardware multi-threading. http://www.metagence.com
11 Watcharawitch, P., and Moore, S.: 'JMA: the java-multithreading architecture for embedded processors'. Int. Conf. on Computer Design (ICCD), (The IEEE Computer Society), September 2002
12 Yiannacouras, P., Rose, J., and Steffan, J.G.: 'The microarchitecture of FPGA-based soft processors'. Proc. Compilers, Architecture and Synthesis for Embedded Systems (CASES), September 2005
13 Atasu, K., Dündar, G., and Özturan, C.: 'An integer linear programming approach for identifying instruction-set extensions'. Proc. Hardware/Software Codesign and System Synthesis (CODES + ISSS), September 2005, pp. 172–177
14 Brisk, P., Kaplan, A., Kastner, R., and Sarrafzadeh, M.: 'Instruction generation and regularity extraction for reconfigurable processors'. Proc. Compilers, Architecture and Synthesis for Embedded Systems (CASES), October 2002, pp. 262–269
15 Sun, F., Ravi, S., Raghunathan, A., and Jha, N.K.: 'Custom-instruction synthesis for extensible-processor platforms', *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, 2004, **23**, (2), pp. 216–228
16 Biswas, P., Choudhary, V., Atasu, K., Pozzi, L., Ienne, P., and Dutt, N.: 'Introduction of local memory elements in instruction set extensions'. Proc. DAC, June 2004, pp. 729–734
17 Handel-C language reference manual. http://www.celoxica.com
18 Dimond, R., Mencer, O., and Luk, W.: 'Automating processor customisation: Optimized memory access and resource sharing'. Proc. Design, Automation and Test in Europe (DATE), 2006
19 Breuer, M.A.: 'Generation of optimal code for expressions via factorisation', *Commun. ACM*, 1969, **12**, (6), pp. 333–340
20 Austin, T., Larson, E., and Ernst, D.: 'Simple Scaler: an infrastructure for computer system modeling', *Computer*, 2002, **35**, (2), pp. 59–67
21 Guthaus, M.R., *et al.*: 'MiBench: a free, commercially representative embedded benchmark suite'. Proc. IEEE 4th Annual Workshop on Workload Characterisation, Austin, TX, December 2001