VThreads: A novel VLIW Chip Multiprocessor with hardware-assisted PThreads

V. A. Chouliaras*, D. Stevens and V.M. Dwyer

Wolfson School, Loughborough University, Loughborough, LE11 3TU, UK.

Abstract

We discuss VThreads, a novel VLIW CMP with hardware-assisted shared-memory Thread support. VThreads supports Instruction Level Parallelism via static multiple-issue and Thread Level Parallelism via hardware-assisted POSIX Threads along with extensive customization. It allows the instantiation of tightly-coupled streaming accelerators and supports up to 7-address Multiple-Input, Multiple-Output instruction extensions. VThreads is designed in technology-independent Register-Transfer-Level VHDL and prototyped on 40 nm and 28 nm Field-Programmable gate arrays. It was evaluated against a PThreads-based multiprocessor based on the Sparc-V8 ISA. On a 65 nm ASIC implementation VThreads achieves up to x7.2 performance increase on synthetic benchmarks, x5 on a parallel Mandelbrot implementation, 66% better on a threaded JPEG implementation, 79% better on an edge-detection benchmark and ~13% improvement on DES compared to the Leon3MP CMP. In the range of 2 to 8 cores VThreads demonstrates a post-route (statistical) power reduction between 65% to 57% at an area increase of 1.2%-10% for 1-8 cores, compared to a similarly-configured Leon3MP CMP. This combination of micro-architectural features, scalability, extensibility, hardware support for low-latency PThreads, power efficiency and area make the processor an attractive proposition for low-power, deeply-embedded applications requiring minimum OS support.

Keywords: RTL Implementation; Embedded Microprocessors; Hardware/software interface; Configurable VLIW architectures; Field-Programmable Gate Array Design; Standard-cell design

1 Introduction and motivation

State-of-the-art silicon technology nodes empowered VLSI designers to integrate complex systems on a single chip with such advanced Systems-on-Chip (SoC) incorporating multiple processing engines. These are, as a minimum, scalar 32-bit central processing units (CPUs), digital signal processors (DSPs) augmented either by data-level parallel (DLP) standalone coprocessors [1], instruction set architecture (ISA) extensions or combinations thereof [2], connected via numerous, high bandwidth, point-to-point buses and more recently, packet-switched networks [3]. These units are supplied by many local memory blocks under the control of Direct Memory Access (DMA) engines. This ever-increasing circuit complexity needs to be delivered in the face of very tight design deadlines, mandated by Time-to-Market (TTM) imperatives, resulting in the final SoC being almost always over-engineered (hence, sub-optimal). Over-engineering is the effect of an increasing productivity gap where the chip complexity that can be handled (and verified) by design teams falls well short of the potential offered by these advanced silicon process nodes. As a result, the investigation of appropriate architectures and micro-architectures and the exploration of the implementation space are minimal in most designs.

At the same time the industry is witnessing a revolution in the capability of Field-Programmable Silicon (FPGAs), particularly in the past few years. The leading vendors in the area such as Xilinx and Altera have

*Corresponding author email: v.a.chouliaras@lboro.ac.uk; Tel: +44(0)1509227013

consistently delivered high capacity silicon (Virtex 6/7/UltraScale/UltraScale+ and Stratix IV/V/10) incorporating hundreds of hard-wired blocks such as static (block) memories, DSP data-paths, clocking infrastructure (PLLs/DLLs), high-throughput interfaces (hard PCIe cores) and networking capability (Hard Ethernet MACs, Interlaken cores), supported by high speed differential I/O. The leading FPGA vendors supply a wealth of silicon intellectual property (IP) in the form of soft processors (Microblaze and Nios2 respectively), high-value hardened silicon IP (ARM A9 SMP subsystem in both the Zynq and Cyclone SoC device families), interconnect systems (IBM PLB, ARM AXI4 and Altera Avalon), and a wealth of other IP blocks covering practically every conceivable application.

What's even more noteworthy is that this rich ecosystem, along with proprietary Electronic Design Automation (EDA) tools is provided for (nearly) free to the FPGA silicon customers. On the tools side in particular, both vendors have embraced potentially disruptive technologies such as Electronic System Level (ESL) design, in its form as Behavioural synthesis (Xilinx AutoESL flow [4] and C2H from Altera) with a good overview of the current state-of-the-art given in [5]. This is a turning point in the design of such complex VLSI systems in Field-Programmable silicon; vendors have identified the complexity of composing such systems with established Register-Transfer-Level (RTL) methodologies as the design and verification bottleneck and thus are trying to embrace a more *software-centric* approach using such flows. Still, widespread adoption of such ESL methodologies is at an early stage thus raising the research question of the suitability of higher level descriptions for system design. Whilst there seems to be a concerted effort towards the adoption of the Single-Program, Multiple-Data (SPMD) model with OpenCL [6, 7] and CUDA [8] being prime examples, research by our group [9, 10] pioneered the use of the *Unified Modelling Language* (UML, Object Management Group UML 2 specification [11]) for the behavioural synthesis of embedded VLSI systems. The latter is widely used in large scale software engineering projects and initially lacked hardware-related modelling abstractions. Our research adopted the *Modelling and Analysis of Real-Time and Embedded systems* (MARTE) profile [12] of UML in the ENOSYS FP7 project[1]. Whilst not focused on the ENOSYS project in this paper, a short description is deemed necessary at this stage to set the scene for the need behind the proposed VThreads architecture, the next generation Very Long Instruction Word (VLIW) Chip Multi-processor (CMP).

1.1 The ENOSYS FP7 Project

The aim of ENOSYS was to reduce design and development cost on Field-programmable Silicon as well as shorten the time to market of electronic products [13] through the co-design of an optimal multi-core system architecture making use of a configurable, extensible VLIW architecture. Whilst there has been very many previous attempts at addressing embedded system co-design, the closest to ours and most notable such effort was the Pico project [14] from Hewlett Packard Labs[2] which proposed the close integration of a fully-predicated VLIW engine with hard-wired implementations of loop nests. ENOSYS took this much further through A) the use of a Multi-core configurable, extensible VLIW architecture B) Its close integration with HLS-designed objects of arbitrary complexity (not only loop nests) under the control of a simple RISC processor responsible for orchestrating data transfers across all processing elements in the system and C) The over-arching Design Space Explorer (DSE) environment. The ENOSYS flow is depicted in Fig 1 and includes two phases:

System Modelling phases (SMF1/2/3): Starting from a textual system description the system requirements are captured during phase SMF1. Subsequent refinement takes place in SMF2 transforming the initial specification to a MARTE-compliant description. The full system functionality is then encapsulated in that MARTE description through the use of an action language such as C++ or Java (SMF3). This is followed by source-to-source transformation which exposes parallelism both at the basic block level (through transforming control to data dependencies) as well as exposing data-level parallelism through automatic application of affine transformations to the iteration indices of tightly-coupled loops. The later form of parallelism is exploited both during behavioural synthesis using *FalconML*, the commercial successor to [15] (for hard-wired blocks) as well as

[1] http://www.enosys-project.eu

[2] Subsequently acquired and commercialized by Synopsys Inc.

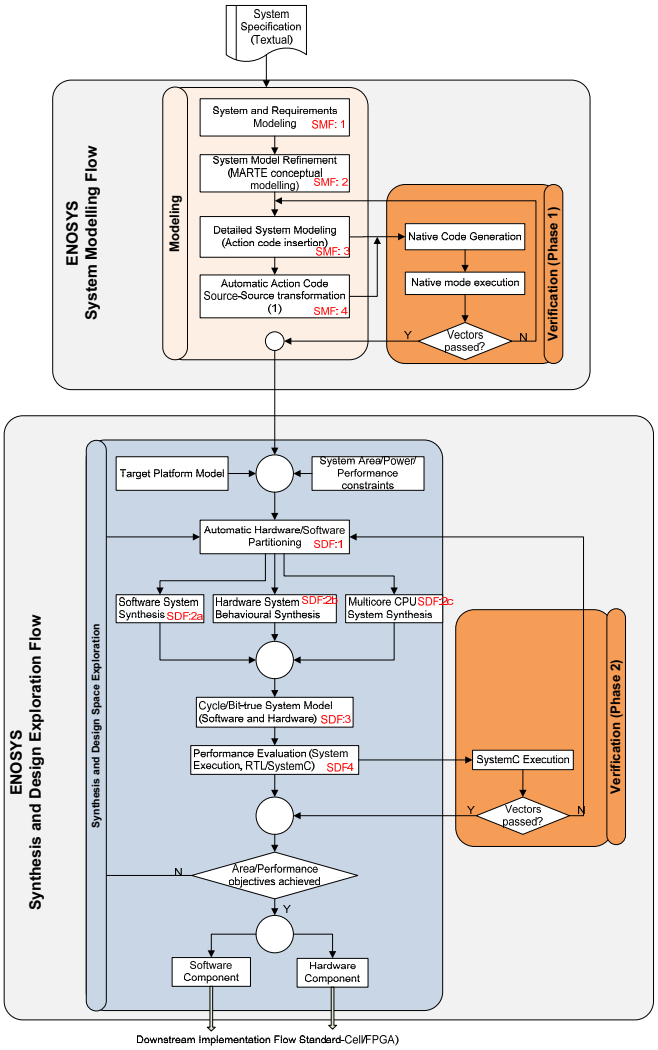


Figure 1: The ENOSYS project flow

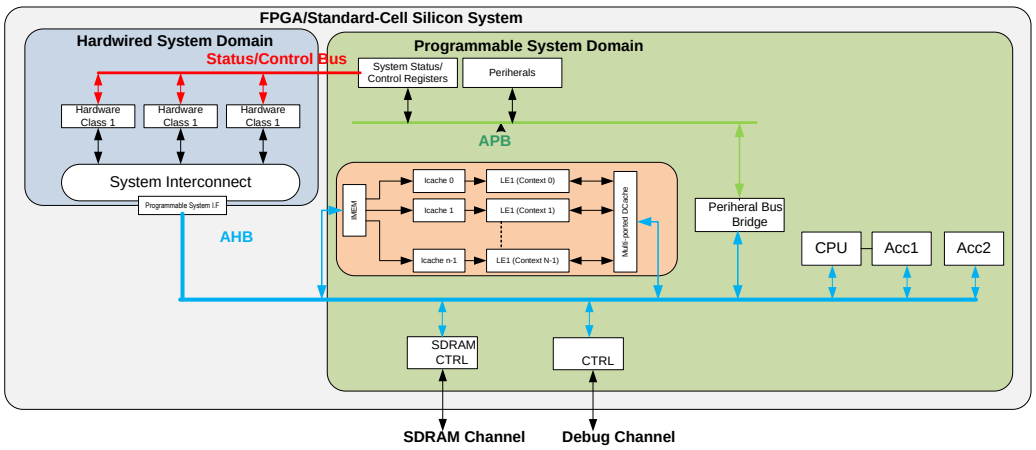
from the compiler of the multi-core CPU-subsystem. The captured system is validated and exported in textual form (XMI) as input to the second, Synthesis and Exploration Flow.

System Synthesis and Exploration phase: During this stage the validated design is partitioned automatically (under DSE control) with objects allocated to CPU cores or directly synthesized to gates via FalconML. The target embedded system architecture is shown in Fig. 2 and consists of a standard host processor (Leon3 for standard-cell technologies and the Xilinx Microblaze for FPGA targets) and a number of default peripherals. A key component of the flow is the multi-core VLIW engine [16, 17] (Figs. 2b and 2c) which forms the first-level accelerator. Objects that can't be efficiently executed on the combined Microblaze+LE1 CMP are offloaded to UML-designed accelerators *(Hard-wired System Domain)* in Fig. 2a. The allocation of software objects to either software classes executing on the LE1 CMP or hard-wired implementations as depicted is left to the DSE element of the flow. The whole platform uses a bespoke API (*bare-metal* on the Microblaze and *run-to-completion* on the LE1).

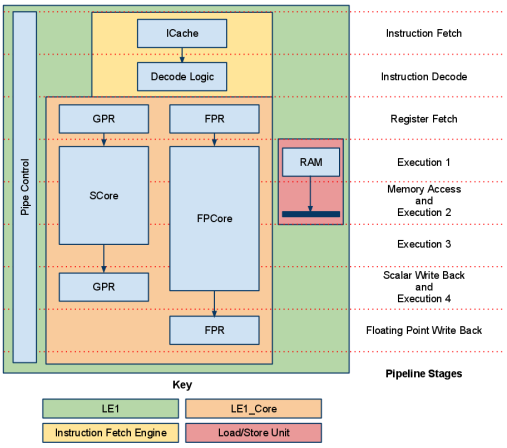
1.2 Motivation and Contributions

Lessons learned during the ENOSYS effort were in a number of areas namely low-level software (OS), hardware (tightly-coupled accelerators) and technology (FPGA vs standard-cell).

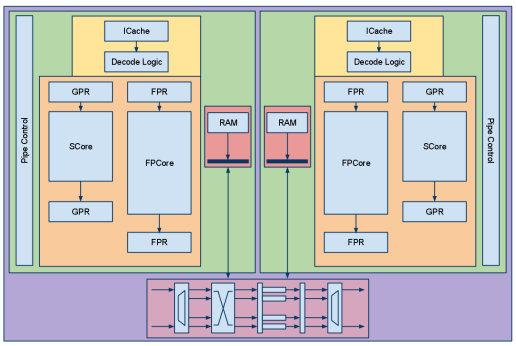
1. **OS:** The need for a low-level OS executive (a simplified RTOS) to be used by the LE1 VLIW accelerator arose towards the end of that research; the authors identified that the silicon overheads and the relatively low performance of a soft-CPU such as the Microblaze weren't always justified as the algorithm com-



(a) Hard-wired and Programmable Domains showing the LE1 VLIW CMP, the FalconML-synthesized Accelerators and the on-chip Host (CPU).



(b) The LE1 Processor Context pipeline



(c) A 2-context LE1 VLIW CMP system

Figure 2: ENOSYS HW Platform.

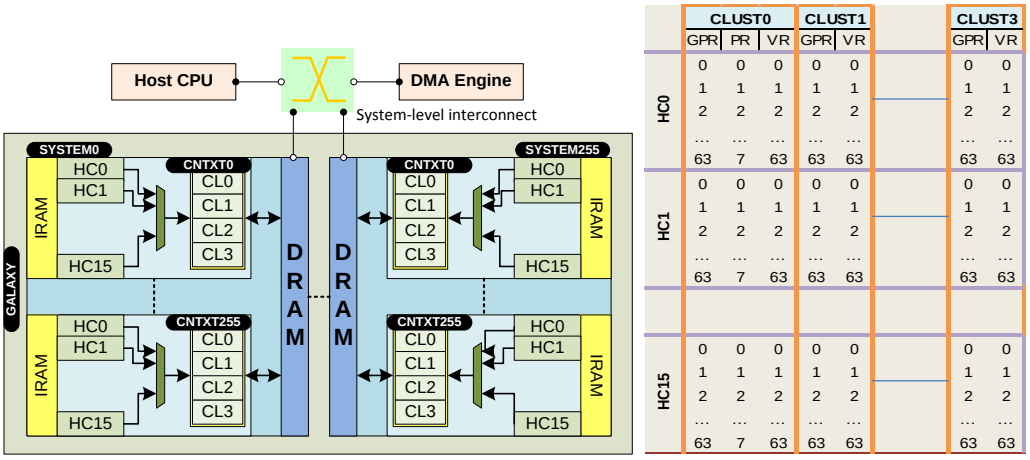
putations were performed by the LE1 CMP and the HW accelerators. It was decided that a new LE1 micro-architecture was necessary to make more efficient use of the 'bare-metal' CPU cores and their close interaction with the hard-wired UML accelerators, limiting the host processor to purely OS and I/O. This is achieved through native (hardware-assisted) PThreads support as code portability between PThreads and code executing on the VThreads CPU was deemed as important.

2. **Hardware:** The LE1 micro-architecture was also at the focus of performance optimization efforts as it was discovered that the LE1 memory subsystem (based on a configurable, multi-banked tightly-coupled RAMs) was more effective for the ENOSYS co-design tasks compared to the host system DDR2 solution. This called for a number of improvements in certain areas in the existing LE1 design namely the Instruction Fetch engine (IFE), tight integration of streaming accelerators, Multi-input/Multi-output (MIMO) Instruction Set Extensions (ISEs), branch prediction, extensive instrumentation and extended use of customization.
3. **Target technology:** Finally, it was decided that the LE1 optimizations should be such that the new version of the processor be fully technology-agnostic in order to support both FPGA and ASIC flows. This called for modifications in the embedded SRAM blocks and the system clocking.

The outcome of this evaluation is the next-generation LE1 system, know as VThreads which is the focus of this paper. The processor addresses (1) by providing minimalist low-level, low-latency, hardware-assisted PThreads support; (2) with more efficient hardware organization and 3) via a re-design in a technology-independent way. An overview of VThreads is given in the next section.

2 VThreads Overview

VThreads is a configurable CMP designed to be used in deeply-embedded co-design applications on FPGA and ASIC technologies. It implements a hybrid, explicit [18] parallelism model supporting both shared-memory semantics in hardware (hardware-accelerated PThreads) and a small set of Message-Passing primitives, facilitated by the host system. VThrads is designed to be attached to a larger system which includes a host CPU running the OS, high-level data scheduling and interfacing. A programmer's view of the VThreads system architecture (full deployment) is depicted in Figure 3a.



(a) VThreads Programmers View

(b) Architected State of Context

Figure 3: VThreads overview showing the hierarchical decomposition of the architecture (GALAXY) into shared memory systems (SYSTEM0..255), Contexts per System (CNTXT0..255), HyperContexts and data-path components per Context(processing clusters, CL0..3).

At the highest level (*Galaxy*), VThreads consists of a configurable number of shared-memory multiprocessors (*System(X)*) and the Galaxy-level interconnect[3]. The host can initiate remote memory access operations across these multiprocessors using it's own DMA facilities. Each such System hierarchy is a shared-memory multiprocessor and consists of up to 256 *Contexts* (processing containers) with each such context encapsulating up to 16 *HyperContexts* (HCs, architected states). HCs multiplex onto the available execution resources of their Context using vertical multi-threading (only one HC can issue per clock). The context execution resources are split over a maximum of 4 clusters (*CL0-CL3*) with each cluster including multiple integer ALUs (*IALUs*), multipliers (*IMULTs*), MIMO ISE data-paths and a multi-ported Load Store Unit (LSU). Clusters are instances of data-path templates (section 3.3) and their execution resources are time-multiplexed across the HCs. Figure 3b depicts in diagrammatic form the *full architected state* of the Context and demonstrates the compiler-available registers in the Context HCs and Clusters[4]

VThreads supports a single local data memory *per System* (DRAM on Fig. 3a), accessible from all the Contexts/HCs in that system, and a configurable, instruction RAM (IRAM). Both memories are parametric and multi-banked.VThreads relies on an ISA-agnostic pipelined micro-architecture with partial support for *fully-predicated* (EPIC) [19] and full support for *partially-predicated* (Multiflow-type) statically-scheduled, long instruction word architectures [20]. These are known as perspectives with the default perspective (*VT32PP*) being the 32-bit partially-predicated VLIW architecture. This is loosely based on the Multiflow architecture with architectural support for Single-Instruction, Multiple-Data (SIMD) processing, multiple register files, MIMO ISEs and control state and hardware support for POSIX Threads. There are also 512 control/configuration

[3]VThreads doesn't bus-master and the Galaxy-Level (Inter-System) Interconnect is implemented on the host system, typically using the AXI4 Interconnect IP on Xilinx 7 Series. Inter-System transfers are scheduled by the host DMA.

[4]The 256 Systems x 256 Contexts define the architectural extrema of VThreads. The 64K Contexts, each with a max. of 16 HCs are encoded in 20-bits which are returned by the VT32PP `CPUID` instruction. Whilst Figs. 3 and 4 depict a single AHB/AXI4 slave port *per system*, the authors have considered the use of an internal (Galaxy-level) NoC, instead of the external AXI4 system, to limit the number of such slave channels. This is not elaborated further as it's not part of the existing VThreads RTL database. All performance experiments have been conducted with a single-system Galaxy configuration as discussed in Section 5.

registers forming the `CTRL_SPACE`, accessible by the host system and the HCs via the DBG_IF (section 3.6) and `RDCTRL` and `WRCTRL` instructions respectively. Finally, there are 256 32-bit peripheral registers (`PERIPH_SPACE`) per context used to map the control registers of attached streaming peripherals. They are accessed via the DBG_IF and the HCs with the `RDPERIPH` and `WRPERIPH` instructions. VT32PP is a true Harvard architecture; instructions have their own 32-bit private address space which is only accessed by the host system with single or streaming IRAM read/write transactions. Long instruction words (LIWs) consist of a variable number (up to the architectural width of the processor) of RISC-type operations known as *syllables* or *RISCops* (used interchangeably in the text). Instruction accesses are byte-aligned (to allow for future implementations where variable instruction lengths are supported) and control transfer instructions target the first syllable of the target LIW. In its current form, VThreads supports blocked (vertical) hyper-threading with micro-architectural hooks in place to support Simultaneous (SMT) [21] and Cluster-Simultaneous (CSMT) [22] multi-threading.

Whilst there are numerous research and commercial machines in the VLIW domain such as CoreVA [23], PACDSP [24], Kalray [25] the VT32PP ISA was chosen for the VThreads organization due to it's similarity to the Multiflow and the LX [26] machines, the maturity of the compiler and the knowledge the designers had acquired during the development of the precursor LE1 CPU. A particularly important point and a major differentiator of our work was the architectural support for MIMO ISEs and embedded streaming accelerators which are not found in most VLIW implementations. The following section goes into the details of the VThreads Organization.

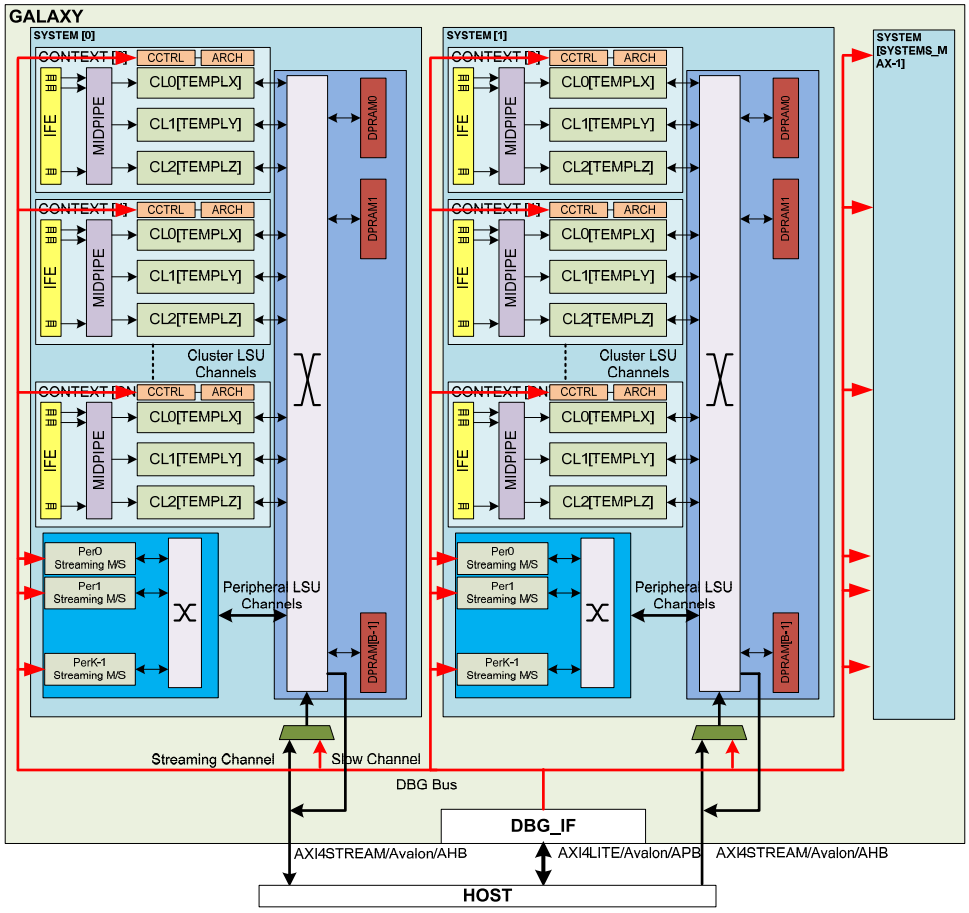


Figure 4: VThreads Micro-architecture

3 VThreads organization

From Fig. 4 the Galaxy is connected to a host system (Microblaze/Nios2/Leon3) which provides a number, equal to the number of instantiated Systems in the Galaxy, of memory-mapped ports (AXI4MM/AvalonMM/AHB)

and a single non-pipelined interface (AXI4LITE/APB) for debug purposes. Each System includes a configurable number of Contexts, a peripheral wrap component and the multi-ported, multi-banked Data Memory. That memory is host-mapped and accessed from via host DMA transactions. The DBG_IF logic is distributed within VThreads to all Systems/Contexts/periph_wrap instances and follows a simple Req/Ack protocol between the DBG_IF Finite State Machine (FSM), the per-context controller and associated processor and peripheral state.

The expert reader will notice that Inter-System communications and synchronization are supported only via host-intervention (host-side DMA and use of MUTEX-type IP) as VThreads is not in its present form designed to bus-master the host system. At the same time, multiple VThreads can be instantiated at a higher level of hierarchy (full system) however sufficient configurability and extensibility is built within the Galaxy to ensure there is no need to do that.

The following sections provide additional details into all the blocks of Fig. 4 with emphasis on the Context Instruction Fetch Engine (IFE), mid-pipe section (MIDPIPE), execution data-paths (Cluster-Architecture) and cluster-level Load-Store unit (CL_LSU)) followed by a discussion of the periph_wrap, the System-Level Memory (DRAM) and the DBG_IF which supports Hardware-accelerated PThreads.

3.1 Instruction Fetch Engine (IFE)

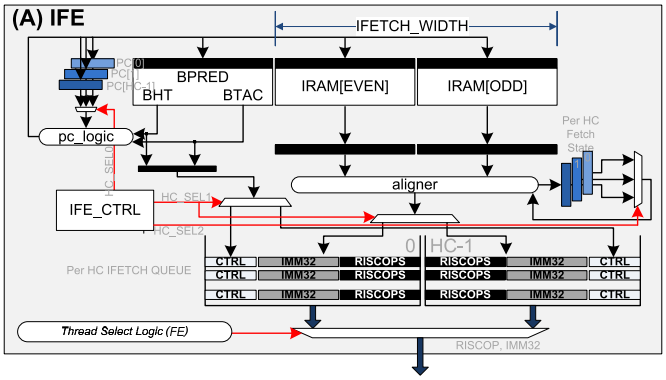


Figure 5: IFE organization

The IFE (Fig. 5) is responsible for accessing the local instruction memory and producing (buffering) as many VLIW bubdles as possible every clock. It includes the per-Context IRAM, the per-HC Program Counter (PC), a configurable 2-bit saturating counter branch predictor with per-HC history bits (BHT) and a single Branch Target Address array. The IRAM is implemented with single-port compiled RAMs (more suitable to standard-cell technologies) however, dual-port SRAMs are used on FPGA targets as they come at the same silicon cost while doubling the fetch bandwidth. The IRAM is split into even and odd banks to allow for single-cycle LIW bundle access in the case where single-port RAMs are used. The IFE includes a per-HC configurable instruction queue which buffers fetched LIW bundles, to be consumed by the downstream stages. A previous version of the IFE included a single-block, non-banked IRAM configuration which resulted in pipeline stalls when the LIW bundle wasn't naturally-aligned [27] and the new design removes this limitation. Micro-architectural parameters of IFE include the LIW issue width, IRAM size and block-size, the use of branch prediction and the detailed micro-architecture (legacy LE1 or VThreads).

3.2 Mid-Pipe

The Mid-Pipe (Fig. 6) includes the decode logic, register files, bypass logic and issue queues per HC. Following from the IFE, one of the non-blocked HCs is selected by the *Thread_Select_Logic* (bottom of Fig. 5) for access to the array of instruction decoders. These are combinatorial blocks producing a large array of control fields which schedule the downstream pipeline resources. The output from *Declogic* includes the register source/destination information which specifies the number of register sources (1-7), destinations (up to 2), type

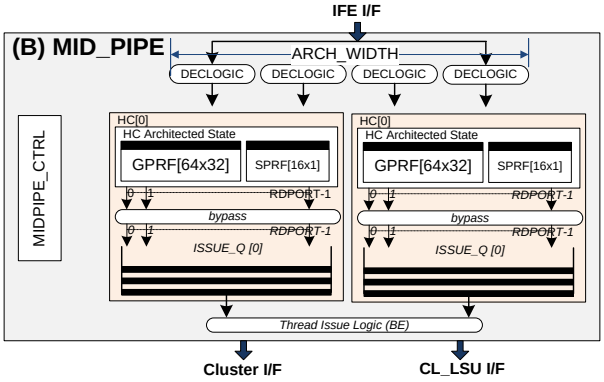


Figure 6: Mid-Pipe organization

(scalar static/rotating GRPs, static/rotating predicates, static vector, the Link register and finally, the PC). The relevant bits are extracted and passed through the *port_allocation* logic which *dynamically* allocates read ports from those available in the register file. This level of internal decoding adds a little complexity however it totally eliminates any LIW issue restrictions and permits the use of MIMO ISEs (Section 3.4). The register file is clocked on the next rising edge and the resultant data are made available late in that cycle. At the same time, the bypass logic settles and switches the necessary multiplexers which select, per HC, per register source, either the register-fetched value or a value produced in the downstream pipeline stages. These operands are finally clocked into the *ISSUE_Q*, waiting for issue to the execution data-paths. Fig. 6 shows the lightweight *Thread_Issue_Logic* which selects, amongst the ready-to-issue HCs, the next one to dispatch downstream and is based on a simple round-robin scheme. Micro-architectural parameters relating to the mid-pipe section are the number of Systems, Contexts per System, HCs per Context, register file organization, available Cluster Templates and Clusters per Context.

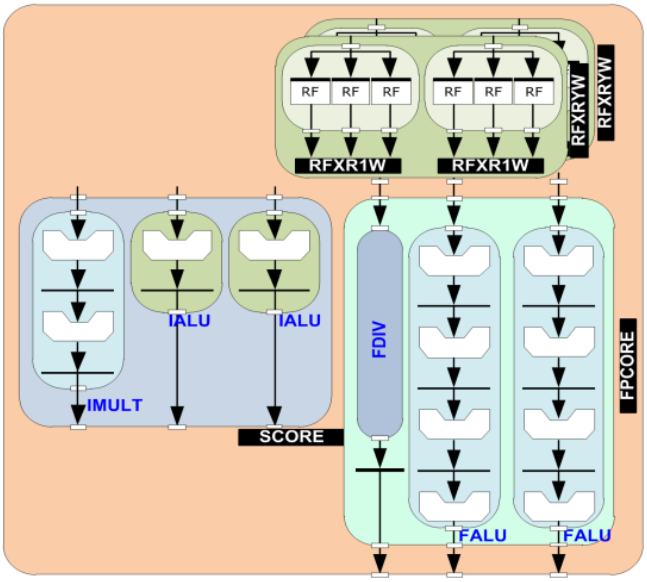
3.3 Cluster Architecture

Clusters are autonomous execution units including pipelined integer (SCORE), floating-point (FPCORE), MIMO ISE data-paths (CCORE, not shown) and associated processor state. A high level view of the cluster hierarchy is depicted in Figure 7a.

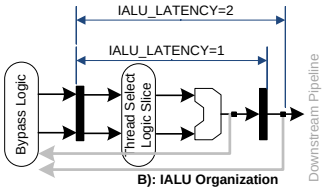
The SCORE encapsulates the data-path pipelines (IALU, IMULT) and the branch logic (BRU, not shown). These are the pipelined execution units and support arithmetic, logical, shift and multiplication operations. The IALU consists of primarily single-cycle data-paths and IMULT instantiates a configurable *DesignWare* component with an explicit latency annotation. Use of configurable latency for the IALU (Fig. 7b) and the IMULT (Fig. 7c) ensures that a high Fmax is achieved (via retiming) despite the series *Thread_Select_Logic* in Fig. 6. This series dependency can also be broken with the explicit specification (via an HDL constant) of a *Thread_Pick_Stage* (TPICK) however, at the expense of an extra clock in Branch/Jump resolution which penalizes the IPC of the machine. Finally, the BRU is responsible for all updates to the per-HC PC and validates the branch prediction, in the process re-steering the former to the correct address if a mis-predicted path was followed.

The FPCORE includes an iterative divider unit (FDIV) and two 4-stage pipelined generic floating point data-paths capable of performing single-precision addition/subtraction and multiplication. In addition, the generic FP data-path includes format conversion from signed 32-bit integers to single-precision FP. FPCORE also makes use of *DesignWare* IP.

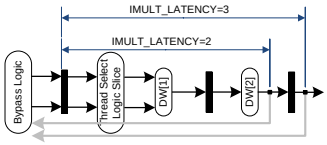
Finally, the cluster includes a multiplicity of register files, one per-HC, each with a statically-defined number of R/W ports and the Cluster Load/Store Unit (discussed in Section 3.5).



(a) Cluster organization showing major configurable data-path components such as SCORE, FPCORE and the Register file organization, per HC.



(b) IALU organization parameters (Latency)



(c) IMULT organization parameters (LATENCY).

Figure 7: Cluster organization VThreads specifies up to 16 cluster templates with a template specifying the SCORE, FPCORE, CCORE units; The designer selects, via HDL constants,the templates to be instantiated (number of clusters).

3.4 Custom Core (CCORE) and Peripheral Wrapper

VThreads includes architectural support for custom MIMO ISEs with syllables extended to 64 bits to allow for the encoding of a maximum of 7 register specifiers. There are 5 categories (`CASM4, CASM5, CASM6, CASM7`) each supporting 4, 5, 6 and 7 such specifiers. CCORE incorporates the data-paths that implement all these extensions and has direct access to the multi-ported RF.

To satisfy the deeply-embedded peripheral requirement of the Hardware aims of Section 1.2, VThreads was architected to tightly integrate such streaming peripherals, operating in parallel to the instantiated context. These peripherals are typically stream accelerators, designed with ESL flows [28] or directly in RTL, and access the common system memory via pipelined ports. Fig. 8 depicts the schematic of the block. VThreads peripherals include a programmer's interface consisting of both *mandatory* and *user-architected* registers which are read from/written to by individual HCs as well as from the host. Peripherals are instantiated at the System level as shown in Figure 4; relevant parameters identify the name of the instantiated peripheral, the number of LSU ports required and secondary channel arbitration. As it stands, the micro-architecture fully supports the accelerators designed with ROCCC 2.0 [29]. The close integration of pipelined accelerators to the processor core is a key characteristic of this design and a differentiator to the previous LE1 effort.

3.5 Cluster Load/Store Unit (LSU) and System-Level Data RAM

The Load/Store Unit (Fig. 9) is the interface to the system memory at the *Cluster level*. A Cluster can include an instance of the LSU and supports a number of direct channels (*ports*) to system shared DRAM. Active HCs

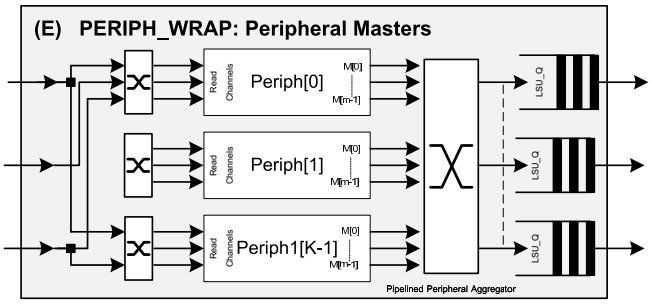


Figure 8: Periph_Wrap organization

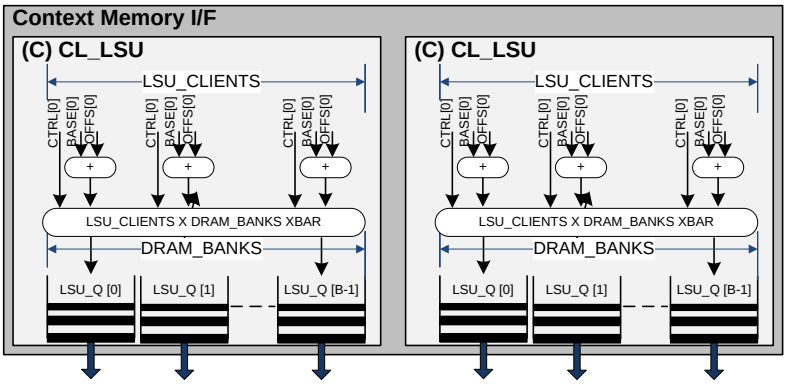


Figure 9: Cluster Load/Store Unit organization

arbitrate (per cycle) for the use of the available memory ports. The winner gets full access to all LSU ports and can issue up to that number Load/Store RISCops. At the System level (Fig. 10) there are multiple, single or dual-port compiled RAM blocks (*Banks*) supporting a maximum of *PORTS*BANKS* LD/ST RISCops per clock, distributed across all contexts in the current system. The clock can be x1 or x2 further expanding the number of ports on both ASIC and FPGA silicon. Supporting both single and dual port RAMs and clock multipliers at RTL makes the processor implementation fully technology agnostic with FPGA targets making use of the second port for free. In addition, configuration parameters specify the XBar architecture (single vs hierarchical) and in the former case, the pipelining depth. Fig. 10 also depicts the system-level periph_wrap block and the DBG_IF DMA Engine channels into the system memory, via that XBar. A further HDL parameter determines

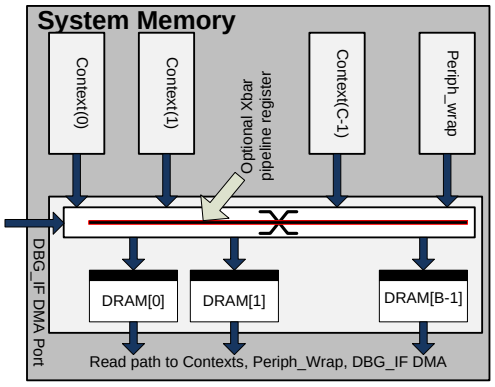


Figure 10: System Memory organization

the depth of the buffering for each issuing context; a value of 0 means that the output from the Effective Address calculation is made available to the DRAM XBar at the end of the cycle. This choice allows for the best single-thread performance (minimal Load-Use latency) however, at the expense of maximum operating frequency.

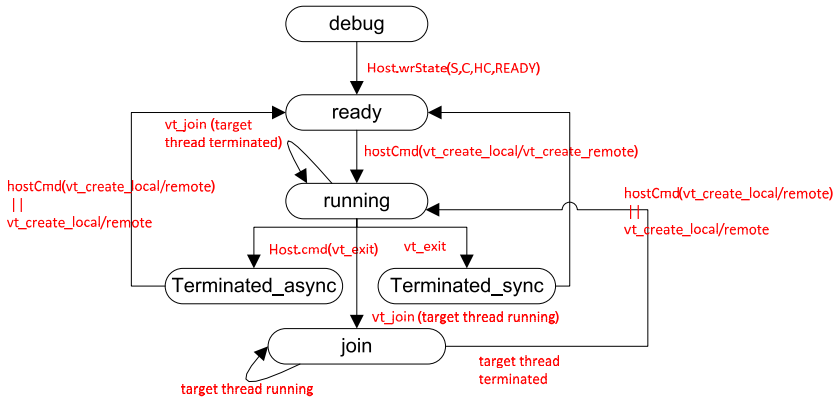


Figure 11: HC State transitions

3.6 The DBGIF and Hardware-accelerated PThreads

This section details the hardware-accelerated POSIX Threads primitives. PThreads was used as the primary threading mechanism due to its low-level nature which maps well to the accelerator philosophy behind the VThreads processor. The HCs can be in a number of states based on run-time events; these events include host interventions (`hostWrState`, `hostCmd`) or execution of VT32PP native instructions such as `vthread_create, vthread_join, vthread_exit`. All HCs maintain private state (transitions depicted in Fig. 11) with the FSM being the synchronization point across all agents (host and HCs) requesting actions.

As shown, all HCs start in the *debug* state; a Host command (`Host.wrState`) advances a particular HC to the *ready* state where it can participate in host-issued thread operations. When such a host command is issued, the HC moves to the *Running* state where it stays there until A) the host issues an asynchronous terminate (`Host.cmd(vthreads_exit)`); B) the thread exit normally (`vthreads_exit`); C) a join operation is issued (`vthreads_join`). In the A) and B) cases, the HC state changes to *ready*, in preparation for re-allocation; in the case of C), the HC enters the *Join* State where it remains until the parent thread joins; it then returns to *ready* for re-allocation. Note that *Terminated_async* and *Terminated_sync* are transient states, currently used as place-holders for future enhancements. The state is maintained via the *threadStateTable* hardware structure as shown in Fig. 12.

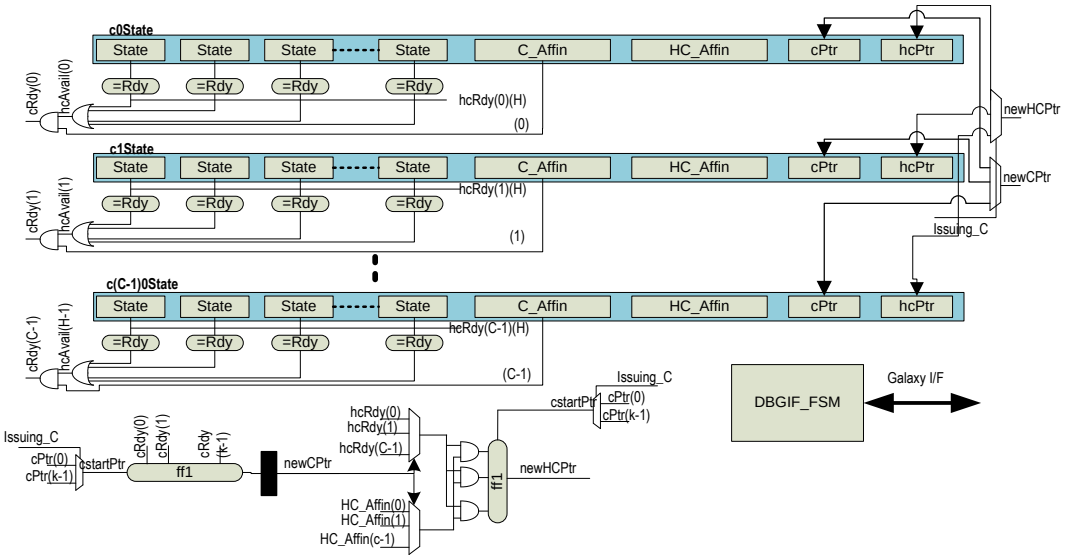


Figure 12: Thread State Table demonstrating the deep VThreads state used in accelerating PThreads operations.

The macro thread-affinity observed when executing software on VThreads is based on the simple Context affinity (C-Affinity) and HC-Affinity hardware algorithms. These rely on two hidden state vectors (System-wide registers, `C_Affin, HC_Affin`) which maintain a bit-mask associating Contexts and HCs, per Context, used

in `vthread_create` operations, *for that context*. From Fig. 12, the Context state includes the state of all HCs, `C_Affin` and `HC_Affin` vectors and two pointers, the `cPtr` and `hcPtr`, used below. Assuming Context0 issues a `vthreads_create()` operation, the state table of all HCs across all Contexts are searched in parallel to identify Contexts with available HCs (vector `hcAvail[]`). This vector is anded with the `C_Affin[]` vector of the requesting Context to identify all Contexts able to provide a new HC for execution (vector `cRdy[]`). The `cPtr` associated with the latter Context is used to drive a *find_first_one* (with a biased starting point) combinatorial block which resolves one out of all the available Contexts. This is clocked in the `newCptr` register, updates the `cPtr` of the issuing Context and is further used to select the first available HC (using the same biased ff1 block). The final output are the vectors `newCPTR, newHCPtr` which are read by the FSM which subsequently loads the PC, LR and SP of the chosen Context, HC and drives the Galaxy-level pipeline signals for the latter to commence execution in MIMD mode. Though not studied in this research, we are investigating other algorithms in which closely-related threads (SPMD paradigm) are grouped on the same Context to maintain close tracking. This is not elaborated further as its not implemented in the current RTL.

4 Processor Customization and Tool-chain

VThreads is a highly configurable, extensible processor architecture implemented in RTL VHDL and includes a low-level C-based API with the whole Sw/Hw flow orchestrated by the LE1 Tool-chain. This section provides more details into the customization capabilities of VThreads, discusses the developed tool-chain and demonstrates user-related software aspects when programming the system.

4.1 Processor Customization

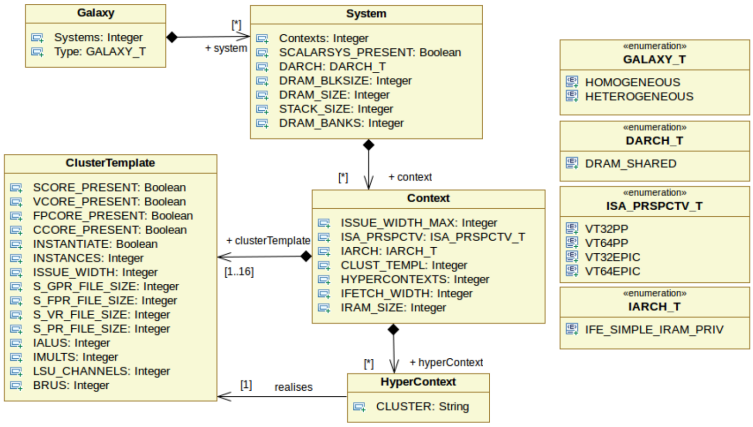


Figure 13: VThreads Customization using an XML template.

Galaxy-level HDL customization is achieved via an XML file as shown in Fig. 13 which is parsed to extract a number of parameters. The example diagram RHS specifies three enumerated types (`GALAXY_T, DARCH_T, ISA_PRSPCTV_T, IARCH_T`) which are used to configure the *Galaxy* (Systems and type); *Systems* (Contexts, availability of scalar processor at system-level, Data memory architecture/block size/size/banks); *Contexts* (VLIW width, ISA perspective, IFE architecture and Fetch width, Cluster Templates, HCs, IRAM banks/-size/block size). The data-path components are described in `ClusterTemplate`. The extracted XML parameters are used to populate a top-level RTL VHDL configuration file (`mastercfg_pkg.vhd`), precisely specifying the hardware architecture, and are communicated to the processor hierarchy via VHDL generics.

4.2 VThreads Tool-chain

The tool-chain developed for VThreads consists of the HP Labs VEX research compiler[5] [26] along with scripting infrastructure to process the generated assembly. The final output of the flow is a set of header files, incorporated on the host application, with the initialized IRAM and DRAM images. The low-level API is used to communicate between a master authority (VHDL simulator/Host processor/x86 workstation) to a slave authority (Insizzle/FPGA silicon) in a seamless way. The host driver and VThreads application is cross-compiled for the host with gcc (Leon3/Microblaze/ARM target) resulting in the `app.elf` file. This is loaded via the Xilinx *xmd* utility to the final FPGA silicon for real-time execution. The flow is depicted in Fig. 14 and the individual tools are further elaborated below:

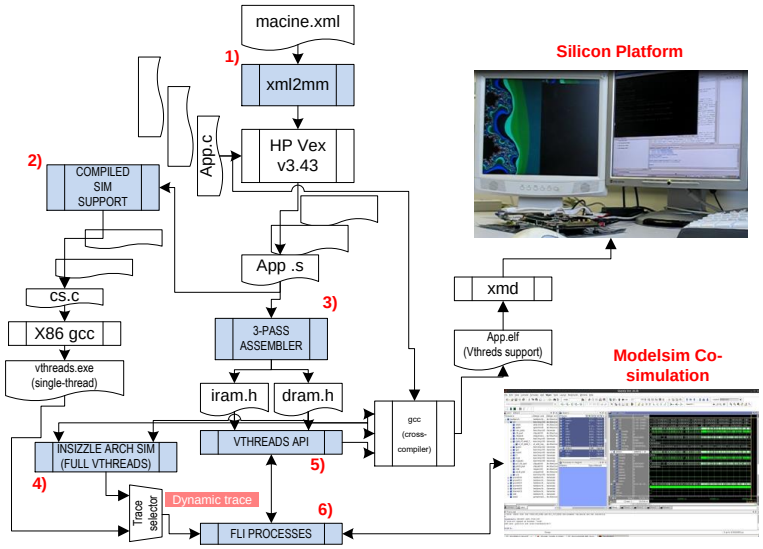


Figure 14: VThreads tool-chain. Tools developed for the processor are shown shaded. The tool-chain permits seamless integration of the application and the host to one of three targets (Insizzle cycle-accurate simulator, Modelsim RTL simulator and FPGA silicon)

1. **xml2mm:** The xml2mm Perl utility parses the `system.xml` file (Fig. 13), recovers all configuration parameters and auto-generates a single `mastercfg_pkg.vhd` file which customizes the RTL database. It generate also the machine description file (MM) which is input to VEX compiler along with the application sources `(App.c)`. The latter produces a number of assembly files `(App.s)`.
2. **Compiled Simulator Support:** This is the first level of execution of the user application `(App.c)`. It consists of a collection of Perl utilities which use headers from VEX and auto-generate a number of C files (`App.cs.c`). When all these files are combined together the resulting x86 executable faithfully represents the behaviour of the user application, as it would when running on silicon. This mode of operation supports a single Context/HC however, it is particularly useful for providing a *Dynamic Trace* (the delta of the processor state affected by every dynamically-executing instruction) which is used, via the Foreign Language Interface (FLI) mechanism below, to communicate with the VThreads RTL in Modelsim Co-simulation of the application and the processor RTL.
3. **3-pass Assembler:** The Assembler input is the scheduled assembly (`App.s`) produced by VEX which goes through a number of transformation stages. These adjust the assembly to the target VT32PP perspective, map the PThreads calls to custom assembly opcodes and allow use of the standard C library on both the physical implementation and Insizzle, the VThreads cycle accurate simulator. Finally, the assembler produces the `iram.bin/iram.h` and `dram.bin/dram.h` images/header files with the latter automatically included with the API in the host application.
4. **Insizzle:** The VThreads Cycle-Accurate Simulator (Insizzle) is an extensible, interpreted simulator which executes instructions as the VThreads hardware would[6]. Unlike the compiled simulator discussed previ-

[5] http://www.hpl.hp.com/downloads/vex/
[6] The deterministic DRAM memory system is modelled accurately on Insizzle resulting in very high RTL/Silicon cycle correlation.

ously, Insizzle includes the full architected state (including HCs, custom ISA support etc.). It is used to confirm correct execution as well as producing various output heuristics which can be used to inspect/profile the code. Insizzle requires the instruction and data binary files (`iram.bin/dram.bin`) produced by the Compilation and Assembly stages along with the XML machine description. As shown in 14, Insizzle is the second source of architectural Traces, used for RTL co-simulation.

5. **VThreads API:** The VTAPI is a C library statically incorporated in the host-resident application code. It is used to identify, initialize and test features of the attached VThreads slave. In addition, it provides a simplified interface to load the IRAM and DRAM of the latter, begin execution and poll the VThreads state for completion. It provides a simplified DMA interface to the host. The API is designed to be used in architectural, cycle-accurate (single and multi-threaded) and RTL simulation modes and on the FPGA prototypes while it allows an external host (Ethernet proxy) to be used for stdio *at runtime*. A subset of the VTAPI calls and relevant categories are listed in Fig. 15.

Category	API Call	Description
Instrumentation	vtApiSetUpInstrumentation	Attach an event to an instrumentation counter
	vtApiRdInstrumentation	Return the contents (long long) of an instrumentation counter
	vtApiextractStats	Extracts statistics from the addressed target
	vtApiPrintStats	Prints statistics from the addressed target
State Extraction	vtApi<Rd/Wr>OneSGpr	Read/Write one static GP register in <S.C.HC>
	vtApi<Rd/Wr>OneSPr	Read/Write one Predicate register in <S.C.HC>
	vtApi<Rd/Wr>PC	Read/Write PC of <S.C.HC>
	vtApiDumpOneArchState	Dump all arch state for context <S.C>
	vtApiDumpFullArchState	Dump all arch state for Galaxy <S>
Peripherals	vtApiRdPeriph	Read one register in the addressed peripheral
	vtApiWrPeriph	Write one register in the addressed peripheral
Ethernet proxy access	vtApiLoadIramFromFile	Load the IRAM of a given context from the file
	vtApiLoadDramFromFile	Load the DRAM of a given system from the file
Standard access	vtApiLoadIram	Load the IRAM of a given context with the array iramBin
	vtApiLoadDram	Load the DRAM of a given SYSTEM with the array dramBin

(a) VTAPI set 1

Category	API Call	Description
Tests	vtApiDmaTest	Full Test of the DMA capabilities of DBG_IF
	vtApiSysTest	Full System Test
	vtApiRfTest	Architected state test
	vtApiPrivMemTest	Private memory tests
	vtApiPeriphTest	Peripheral identification tests
	vtApiDbgIfDiags	Full DBG_IF diagnostics
Execution Control	vtApiCtrlWait	Busy-waits on the VT_CTRL_REG of the addressed <S.C.HC> until DEBUG=1
	vtApiWaitOneHc	Waits for the addressed <S.C.HC> to return to the DEBUG state (VTPRM)
	vtApiStartOneHc	Starts the addressed <S.C.HC> (STATE=RUNNING) at given PC with given SP
	vtApiForkMultiCHC0	Start the <S.C.0> in multiple contexts (0..contexts)
	vtApiJoinMultiCHC0	Busy wait on <S.C.0> in multiple contexts (0..contexts). Return when all HCs are in DEBUG mode
	vtApiLoadAndStartOneHC	Start the addressed S.C.HC
	vtApiIssueDbgCmdAndWaitForCompletion	Wait for execution of Debug command on addressed <S.C.HC>
DBG_IF DMA	vtApiDma	Initiate polling-based DMA

(b) VTAPI Set 2

Figure 15: VTAPI Subset functions. These are available to the embedded host with the Ethernet proxy being used to re-direct host stdio to an x86 host.

6. **FLI Processes:** The VThreads tool-chain is architected to provide a dynamic (through Unix Queues) execution trace to the RTL simulator. This is achieved with the Foreign Language Interface (FLI) of Modelsim which is our simulator of choice. Typical usage includes the use of a dynamic trace, produced *either* by the compiled simulator *or* Insizzle, to cross-check the VThreads pipeline in a VHDL test-bench. The RTL includes a large number of test-points exposed at the top-level interface which, whilst non-synthesizable (guarded with `translate-off/on` pragmas), are available in the HDL simulator allowing full checking of a number of micro-architectural fields, per RISCop, per clock, per Context/System.

4.3 Application Environment

The VThreads tool-chain allows the programmer to seamlessly develop, compile and execute the target application in a Unix environment. Fig. 16 depicts an extract from the DES benchmark (inner loop creating multiple threads, Section 5.3.4) which was run on the benchmarked processors (Leon3+FSU, Leon3MP+Custom PThreads and VThreads, Section 5.3) with the only modification being the initialization of the FSU library (`pthread_init`) for Leon3. The automation afforded by our tool-chain takes care of all low-level details resulting in a final executable which can be run on Insizzle or FPGA silicon.

5 Performance Evaluation

This section discusses the performance of VThreads compared to a number of other CPU cores for A) a number of micro-benchmarks (section 5.2) and B) by executing four larger threaded applications (section 5.3). Four

```
#if defined LEON3
  pthread_init();
#endif

  zeroOut(Keys);
  initDES(K, Keys);

  {
    int id;
    /* thread it */
    for(id=1;id<THREAD;++id) {
      args[id] = id;
      pthread_create(&ids[id], NULL, thr_DES, &args[id]);
    }

    /* do work */
    args[0] = 0;
    thr_DES(&args[0]);

    /* join it */
    for(id=1;id<THREAD;++id) {
      pthread_join(ids[id], NULL);
    }
  }
#endif
```

Figure 16: Code snippet showing the inner create/join operation for the DES benchmark.

processors are used in the study. The MicroBlaze and Leon3 are two scalar (single-issue) soft CPUs designed for FPGAs (both) and standard-cell implementation (Leon3). Leon3 was included in this study as it was originally considered for the ENOSYS CMP architecture (CPU on Fig. 2) and dismissed due to it's scalar issue and lack of sufficient RF bandwidth for MIMO ISEs. The MicroBlaze is included in the Xilinx tools (PlanAhead and Vivado for the latest FPGA families). It is AXI4-based, highly configurable and can be optimized for either area (3-stage pipe) or performance (5-stage pipe). The 5-stage configuration was used in this study. This processor has *Interleaved Multi-threading* (IMT) PThreads support through the Xilinx Kernel (XilKernel). The Leon3 is based on the SPARC-V8 RISC architecture from Gaisler Research. It is AHB-based, easily portable to both standard-cell and FPGA silicon and comes in a multiprocessor configuration (Leon3MP). PThreads support was implemented on Leon3 with the library developed at Florida State University[7]. Both implementations above support IMT and use context switching to time-multiplex multiple software threads on a single CPU. For the multi-core Leon3MP PThreads support was provided via a custom Sw/Hw solution developed as part of this research (section 5.1). Finally, the last processor is VThreads configured as a single-System Galaxy with between 1-8 single-HC Contexts and making use of the PThreads hardware-support (section 3.6) to provide a minimalistic environment which allows for the execution of all benchmarks in this section. All processors were benchmarked on silicon (Xilinx ML605, Virtex6 LX240T-FG1156) and in the case of the Leon3MP and VThreads, they were also synthesized on the UMC 65 nm 10M standard-cell technology from Faraday Tech. On FPGA silicon Microblaze operated at 75 and 150 MHz; the Leon3MP at 120 MHz (1, 2 cores), 86 MHz (4 cores) and 75 MHz (8 cores); VThreads run at 120 MHz (1 Context), 100 MHz (2 Contexts), 75 MHz (4 Contexts) and 62.5 MHz (8 Contexts). The experimental set-up is summarized in Table 1:

Table 1: Processor architectures and PThreads support used in this study

Processor	Number of cores	Threading Library	Threading Type	Fmax
MicroBlaze	1	PThreads	Interleaved (IMT)	75:150
Leon3	1	PThreads	Interleaved (IMT)	120
Leon3MP	up to 16	custom PThreads	Chip-Multiprocessing (CMP)	120:120:86:75
VThreads CMP	up to 8	hardware-assisted PThreads	Chip-Multiprocessing (CMP	120:100:75:62.5

[7]http://moss.csc.ncsu.edu/~mueller/pthreads/

5.1 Leon3MP and the Custom PThreads Library

To provide a realistic benchmarking target for VThreads, a custom PThreads implementation was devised for the Leon3MP processor. The latter allows up to 16 AHB masters (a total of 12 CPUs) to be instantiated within a single-tier 32-bit AHB system; Our custom PThreads library relies on CPU0 being the master processor with all others activated on demand through the use of an interrupt register. A custom boot-loader was developed to offset the initial CPU stack pointers (defined in `crt0.s`) for each Leon3 core and override memory resetting when any CPU other than CPU0 is started. Additionally methods for performing thread creation/termination/synchronization were devised using global arrays and data structures to make the state of each CPU globally available. The subset of PThreads operations supported are `pthread_create`, `pthread_exit` and `pthread_join` and closely track both the POSIX standard as well as the hardware PThreads support of VThreads. To illustrate, consider only CPU0 being active at the start of execution. It reads system configuration registers to ascertain the number of available processors in the Leon3MP system. Then the global state table is initialized; this includes the state of each core in the multiprocessor and function and argument pointers to the target routine argument for `pthread_create`. The other CPUs are then started however, they do not run any initialization code (due to the modified `crt0.s`) and instead are directed to a function where they enter a very tight polling loop, monitoring the state table entry relating to their ID. Execution of `pthread_create` on a core causes the global state to be scanned to locate a currently inactive (unallocated) core; once identified, its global state is updated to *busy* and the function and argument pointers are initialized to the values passed to the `pthread_create` operation *on the issuing core*. The selected core then begins execution of that function (thread). Upon completion it updates its global state table to *available*, to allow subsequent threads (issued on other cores) to be allocated on it, finally returning to the tight polling loop. Note that if no cores are available, `pthread_create` is stalled until another core is made available using the same mechanism. Though not ideal from an energy-efficiency point of view, this mechanism does provide a very fast PThreads implementation on Leon3MP; a second generation of this library is being considered to identify and take into account low-power modes of the Leon3 processor thus dispensing with the use of tight polling.

5.2 Micro-benchmarking

This section evaluates the performance of basic PThreads operations across the processors and PThreads API implementations of Table 1. The following test cases are identified:

- **Test 1: Create:** Measures the time elapsed between the master thread issuing a `pthread_create` and the slave thread beginning execution. A timer is started prior to `pthread_create` and stopped within the slave thread. In the case of VThreads, internal instrumentation is used to obtain a precise figure of the number of elapsed clocks.
- **Test 2: Join:** Measures the elapsed time from the slave thread completing to the master thread being aware of the termination of the slave thread through a `pthread_join` operation. A timer is started and the slave thread exits; the master thread performs a `pthread_join` and once it is made aware of the termination of the slave thread the timer is stopped from the master thread.
- **Test 3: Create & Join:** The slave thread in this test executes an empty function and the results show the accumulated time taken for both previous tests. A timer is started, the master thread performs a `pthread_create` directly followed by a `pthread_join`; once that happens the timer is stopped. Internal hardware timer/instrumentation mechanisms are used to retrieve either the number of clock cycles or the wall time with all data obtained from FPGA silicon. For the MicroBlaze data are generated using the XPS Timer IP whereas the Leon3 and Leon3MP figures were obtained from the `newlib clock()` function. The latter returns the number of microseconds since the start of execution and that value includes an error margin with the higher the frequency the greater that error. In this study frequencies of 75 MHz (8-core) and 120 MHz (2-core) were used, resulting in possible errors of ± 37.5 and ± 60 cycles, respectively which are negligible. Finally, the VThreads results are generated also from FPGA silicon making use of the internal instrumentation peripheral. This monitors a number of internal events across all Contexts and records them on 16 33-bit counters. These

cumulative values are extracted by the host via the DBG_IF. The VThreads configuration was a 2-Context System with both Contexts consisting of asingle HC. A single Cluster template with 2 IALUs, 2 IMULTs and 1 LSU channel was used per Context. A single-bank system DRAM was used. The tests were executed 1000 times to account for the non-deterministic nature of the IMT implementations and the mean time was used. Table 2 depicts the results across all architectures with derived figures (for the standard-cell implementations) displayed in italic. These derived (extrapolated) figures convert FPGA-to-ASIC real-time use Frequency Scaling and measure the efficiency of the processors on ASIC technologies[8]. Results are discussed in section 5.4.

Table 2: Simple Benchmarks Results (Clocks+real time (usec))

Silicon Target	Processor	Fmax	Create		Join		Create & Join	
			Cycles	Time	Cycles	Time	Cycles	Time
Virtex6 LX240T	MicroBlaze	IMT @ 75MHz	2185	29.13	1418	18.91	3542	47.23
	MicroBlaze	IMT @ 150 MHz	3254	13.02	1937	7.75	5129	20.52
	Leon3	IMT @ 75MHz	15600	208.00	359100	4788.00	374025	4987.00
	Leon3	IMT @ 120 MHz	18360	153.00	581280	4844.00	598680	4989.00
	Leon3MP (2 CPUs)	CMP @ 120 MHz	240	2.00	0	0.00	240	2.00
	LE1 (2-contexts)	CMP @ 100 MHz	32	0.32	32	0.32	64	0.64
UMC65 nm 1.2V, 25C	*Leon3MP (2 CPUs)*	*CMP @ 500 MHz (extr.)*	*240*	*0.48*	*0*	*0.00*	*240*	*0.48*
	LE1 (2-contexts)	CMP @ 481.7 MHz	32	0.07	32	0.07	64	0.13

5.3 C benchmarks

This section presents more substantial application benchmarks threaded at *fine* and *coarse* levels. In coarse-level threading, a single `pthread_create` operation is executed per available core at the beginning (initiated by core 0). Each thread then performs the relevant computation and finally `pthread_join` is used to synchronize all active threads. In this case the number of `pthread_create` and `pthread_join` operations used is equal to N-1 where N=number of CPUs. Fine-grain threaded benchmarks make deliberately as much use of the PThreads support as possible. Also each benchmark takes into account the physical cores available with the coarse-grained benchmarks always creating enough threads to fully utilize all cores and each thread computing an equal, if possible, fraction of the workload. Fine-grained benchmarks differ in that at certain points within the code multiple threads are created to saturate the architecture. These threads compute their fraction of work and are then synchronized to the main thread. Results were extracted from FPGA silicon for both the IMT (Leon3, MicroBlaze) and CMP (Leon3MP, VThreads) architectures.

- **IMT Architectures:** An issue experienced in some benchmarks in the coming sections related to the on-board timers for the IMT Leon3 and MB. When clocked at 75MHz the 32-bit MB timer overflowed after 57 seconds which was simply not long enough for the successful execution of all instances of the benchmark. A separate process, running as a thread, was investigated to track this overflow and increment a second counter however introducing this housekeeping thread added overheads in thread interleaving. Similarly for the IMT Leon3 the timers returned odd numbers executing on silicon. It was discovered that the

[8]The expert reader will notice that the Leon3MP results are based on the Xilinx Coregen DDR2 controller which can't be benchmarked on our trial ASIC implementation. The VThreads results however scale perfectly as the processor uses an internal single-port-compiled-SRAM based memory. Still, the Leon3MP results are valid under the assumption that an ASIC implementation of the aforementioned Coregen component exhibits the same latencies at the higher Frequencies achieved in the ASIC implementations in Section 6.2

timer only incremented for CPU0 while it was executing (disabled during context switch) thus skewing the results. As a result of these issues IMT systems were not studied further in the following sections.

- **CMP Architectures:** The multiprocessor architectures (Leon3MP and VThreads) were evaluated with a varying number of active cores (1, 2, 4 and 8) and for the case of Leon3MP, the extrapolated results (identified with Italics in Tables 3 through 6) scale the real-time obtained from FPGA execution to derive the ASIC real-time. The benchmarks were compiled for Leon3MP with *sparc-elf-gcc*, -O3 optimization level and emulated FP support[9]. For VThreads the Tool Collection of Section 4 was used with -O4 optimization switches on VEX. The maximal CMP configurations are: A) Leon3MP: 8 CPUs, 4-way 32KB instruction cache, 4-way 16KB data cache per CPU at a system clock of 75MHz; B) VThreads: 8 homogeneous Contexts, each comprised of a 2-issue VLIW with 2 IALUs, 2 IMULT units and a single LSU channel per Context. A 256KB/4-bank system DRAM is used with the Galaxy clocked at 62.5 MHz. To ensure fairness in our standard-cell comparison, the same VThreads memory system (256KB/4-banks) was migrated to the Leon3MP replacing the DDR2 controller). When implemented on a 65 nm 10M process Leon3MP achieved 412.9 MHz and the VThreads CMP 398.2 MHz respectively at 8 cores. Standard-cell results are discussed in section 6.2.

5.3.1 Mandelbrot Set (Multi-threaded)

The Mandelbrot Set [30] is a mathematical set of points whose boundary is a distinctive 2D fractal shape named after the mathematician Benot B. Mandelbrot. This benchmark was selected as it is a highly parallel and truly data-independent. Each point can be calculated in parallel based on a predefined magnification setting and origin coordinates. A second set of co-ordinates is then passed to calculate the pixel value at a specific position. In this study a 160 by 480 Mandelbrot fractal is computed using a 10-colour palette and a maximum iteration value of 100. Originally the benchmark was split into contiguous sections (slices) of the image which resulted in large computational load imbalance. Subsequently, computations were split on a row-basis resulting in an interleaved output with the load more evenly balanced across threads. In coarse-grained threading a master thread generates a new thread on each available core tasked to compute a section of the output image. The fine-grained threading uses multiple PThreads operations. This are ((Size / N umber of C ores) $*$ (N umber of C ores $-$ 1) with Size equal to the number of output (76,800 based on a 160x480 image). The master thread iterates through all output pixels and creates new threads for each. It then computes a pixel value itself and synchronizes with all other threads. This is performed in a loop until all output pixels have been generated. This results in 0 PThreads operations pairs on single core and up to 67,200 PThreads operations pairs on 8 cores. Table 3 shows the real time taken for the execution of the benchmarks on the Virtex6LX240T device and the extrapolated Leon3MP ASIC results.

Table 3: Mandelbrot consolidated results. Data shows the real time (usec) for all architectures executing on FPGA (Virtex6 LX240T-FG1156) and on 65 nm standard-cell silicon (UMC65 nm from Faraday Technologies, Typ. conditions: 1.08V, 25C). The Leon3MP UMC65 results (Italic) are extrapolated from the FPGA execution.

Threading	**Platform**	**1-core**	**2-core**	**4-core**	**8-core**
Coarse	Leon3MP (40 nm FPGA)	12234.96	6149.95	4337.86	2521.29
	Leon3MP (UMC65)	*2936.39*	*1475.99*	*875.94*	*463.49*
	VThreads (40 nm FPGA)	2224.97	1314.52	906.09	681.44
	VThreads (UMC65)	533.99	272.89	166.83	85.55
Fine	Leon3MP (40 nm FPGA)	12250.82	6322.52	4605.69	2797.64
	Leon3MP (UMC65)	*2940.20*	*1517.41*	*930.02*	*514.29*
	VThreads (40 nm FPGA)	2408.72	1357.75	973.60	771.24
	VThreads (UMC65)	578.09	281.87	179.26	96.83

[9] http://www.jhauser.us/arithmetic/SoftFloat.html

5.3.2 JPEG Decode (Multi-threaded/Multi-programmed)

This JPEG decoder is based on a small C implementation called NanoJPEG[10]. modified to remove dynamic memory allocation and file I/O in order to run on embedded VLSI processors. A 64x64 JPEG image (Lena) was used as the input data set. Due to the block-based nature of JPEG there are data-dependencies between macro blocks (MBs) with pixels computed possibly relying on other pixels in the same or adjacent MBs. As a result, the coarse-threaded version uses eight instances of the JPEG decoder with all instances decoding a separate image. For fine-grained threading it was noted that at the end of MB decoding an inverse discrete cosine transform (`IDCT`) is performed on each column of pixels within that MB (`colIDCT`), called eight times, once per column. This loop was modified to split the work over all available cores and then use a unique identifier along with the total number of parallel threads to specify whether or not to perform the computation. Fine-grained threading is performed on a 64x64 image resulting in 96 MBs and up to 672 PThreads operation pairs in the 8-core configuration. As the amount of computation performed within that function is small it serves as a good example to demonstrate the low-latency threading support in Leon3MP and VThreads. Real-time results from executing on JPEG benchmark are shown in Table 4:

Table 4: JPEG Decode consolidated results. Data shows the real time (usec) for all architectures executing on FPGA (Virtex6 LX240T-FG1156) and on 65 nm standard-cell silicon (UMC65 nm from Faraday Technologies, Typ. conditions: 1.08V, 25C). The Leon3MP UMC65 results (Italic) are extrapolated from the FPGA execution.

Threading	**Platform**	**1-core**	**2-cores**	**4-cores**	**8-cores**
Coarse	Leon3MP (40 nm FPGA)	227.18	139.65	144.10	148.05
	Leon3MP (UMC65)	*54.52*	*33.52*	*29.10*	*27.22*
	VThreads (40 nm FPGA)	145.38	103.55	94.91	123.85
	Vthreads (UMC65)	34.89	21.50	17.48	21.30
Fine	Leon3MP (40 nm FPGA)	26.09	24.62	33.48	39.25
	Leon3MP (UMC65)	*6.26*	*5.91*	*6.76*	*7.22*
	VThreads (40 nm FPGA)	16.60	20.60	28.91	48.40
	Vthreads (UMC65)	3.98	4.28	5.32	6.08

5.3.3 Sobel Filter (Multi-threaded)

The Sobel Filter algorithm is typically used as first stage processing in feature-detection in which a source image results in an output grey-scale image displaying ”edges”. Each pixel in the source image is used in calculations with its surrounding pixels and two masks are used to find horizontal and vertical transitions in order to detect edges and corners. The algorithm is computationally-intensive however, each output pixel can be calculated independently. Inputs to the algorithm are a 640x480 image and two 3x3 mask arrays. Coarse-grained threading results in between 0 and 7 PThreads operation pairs. This example is split in an interleaved fashion similar to the Mandelbrot Set where each thread (core) processes a full row of the input image and then moves down a set number of rows. The real-time results recorded from FPGA silicon are shown in Table 5:

Table 5: Sobel Filter consolidated results. Data shows the real time (usec) for all architectures executing on FPGA (Virtex6 LX240T-FG1156) and on 65 nm standard-cell silicon (UMC6 5nm from Faraday Technologies, Typ. conditions: 1.08V, 25C). The Leon3MP UMC65 results (Italic) are extrapolated from the FPGA execution.)

Threading	**Platform**	**1-core**	**2-core**	**4-core**	**8-core**
Coarse	Leon3MP (40 nm FPGA)	10131.32	5097.15	3619.16	2260.82
	Leon3MP (UMC65)	*2431.52*	*1223.31*	*730.81*	*415.61*
	VThreads (40 nm FPGA)	4947.30	3071.59	2219.78	2053.55
	Vthreads (UMC65)	1187.35	682.39	408.72	257.82
Fine	Leon3MP (40 nm FPGA)	10286.05	5575.03	4398.11	3119.44
	Leon3MP (UMC65)	*2468.65*	*1338.01*	*888.10*	*573.45*
	VThreads (40 nm FPGA)	4952.16	3287.04	2507.72	2569.92
	Vthreads (UMC65)	1188.52	682.39	461.73	322.65

[10]http://keyj.emphy.de/nanojpeg

5.3.4 Data Encryption Standard (Multi-threaded)

The Data Encryption Standard (DES) was developed in the early 1970s and published as an official standard in 1977. It has since been surpassed by other such standards as it is considered insecure due to the key size being small enough to be susceptible to brute force attacks. The algorithm uses a 64-bit key to generate a set of 16 48-bit sub-keys used to encrypt 64-bit blocks of plain-text, through a 16 stage Feistel Network, into 64-bit blocks of cipher-text. This benchmark can be threaded with multiple threads processing 64-bit plain-text and cipher-text pairs as they are data-independent. Similarly to previous benchmarks DES is parallelized both at coarse and fine-grained levels. 64KB of data are processed resulting in 8192 blocks of 64-bits to encrypt. In coarse-grained threading a thread is instantiated within each available core and then works across the input data using its knowledge of the total number of threads and the size of the data to be encrypted whereas in the fine-grained threading the maximum number of PThreads library operations pairs executed is 7,168, based on 8,192 blocks. The real-time extracted from FPGA silicon and the scaled ASIC results are shown in Table 6:

Table 6: DES consolidated results. Data shows the real time (usec) for all architectures executing on FPGA (Virtex6 LX240T-FG1156) and on 65 nm standard-cell silicon (UMC65 nm from Faraday Technologies, Typ. conditions: 1.08V, 25C).). The Leon3MP UMC65 results (Italic) are extrapolated from the FPGA execution.

Threading	**Platform**	**1-core**	**2-core**	**4-core**	**8-core**
Coarse Threading	Leon3MP (40 nm FPGA)	1400.53	704.93	498.13	310.61
	Leon3MP (UMC65)	*336.13*	*169.18*	*100.59*	*57.10*
	VThreads (40 nm FPGA)	907.89	564.97	444.15	401.52
	Vthreads (UMC65)	217.89	112.99	81.78	50.41
Fine Threading	Leon3MP (40 nm FPGA)	1392.65	711.13	541.89	358.27
	Leon3MP (UMC65)	*334.23*	*170.67*	*109.42*	*65.86*
	VThreads (40 nm FPGA)	909.20	530.65	391.42	400.56
	Vthreads (UMC65)	218.21	110.16	72.07	50.29

5.4 Discussion of results

Fig. 17 depicts the speed-up of VThreads over all the measured CPUs for the micro-benchmarks of Section 5.2. As expected, the 2-Context, 100 MHz configuration demonstrates a speed-up of between x40-x3.91 over the Microblaze/IMT (XilKernel) and x478->x15000 over Leon3/IMT (FSU PThreads). For the CMP implementations, VThreads demonstrates a speed-up of between x3.13 to x6.25 compared to the dual-core 120 MHz Leon3MP on the VIRTEX6LX240T device. It is clear that the IMT implementations can't match the two multi-core architectures with the basic PThreads support; A full implementation such as FSU-PThreads in particular imposes substantial performance overheads which can't be justified when minimalist OS-like capability is required for deeply-embedded applications such as the ENOSYS platform of Fig. 2. The capability is very well supported by both Leon3MP and VThreads. Targeting UMC65 technology the 481.7 MHz VThreads demonstrates a speed-up of between x3.6 -x7.23, for create and join, compared to the very tight custom PThreads implementation of the dual-core 500 MHz Leon3. As the Leon3MP data are extrapolated it is expected that results will improve somewhat for Leon3MP when using the very same memory subsystem as the VThreads. Overall, micro-benchmarking quantifies the benefit of very fine-grained, low-latency PThreads support compared to state-of-the-art commercial and research processors and API implementations.

Fig. 18 depicts the VThreads speed-up across all silicon configurations, threading granularity and core count for the more substantial workloads against the Leon3MP+Custom PThreads. The dashed black line identifies the limit with VThreads being faster than Leon3MP (result >1.0). Focusing on the 1-core results (no threading), conclusions can be drawn on the relative efficiency of the micro-architectures; VThreads demonstrates a substantial speed-up on the Mandelbrot workload which can be attributed to the use of banked local memories (both IRAM and system-wide DRAM) compared to the caching system of Leon3MP. Also, the 2-issue LIW efficiency is evident in the remaining benchmarks compared to the Leon3MP scalar issue with the Sobel filter showing slightly >2.0 speed-up - this can be attributed to the differences of a trace-scheduling compiler such as VEX against the gcc 3.3 used for Leon3MP.

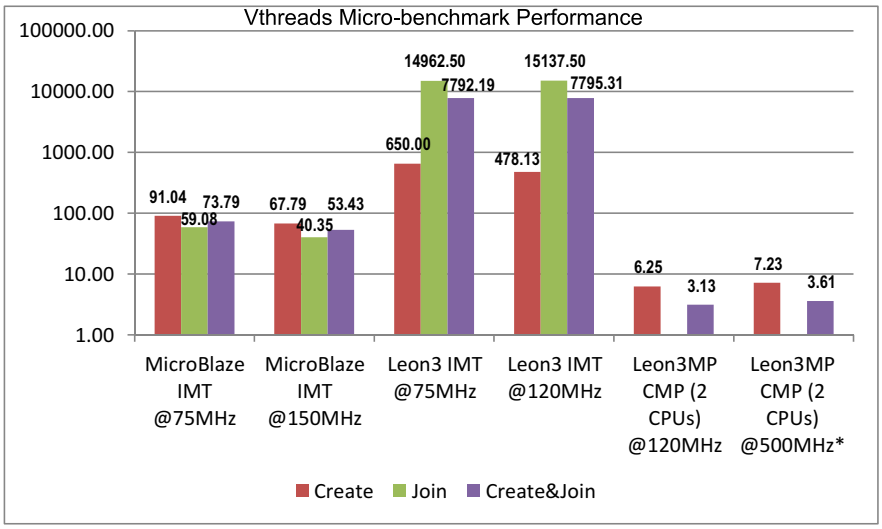


Figure 17: VThreads speed-up over all processors, all technologies for the synthetic benchmarks (Micro-benchmarks)

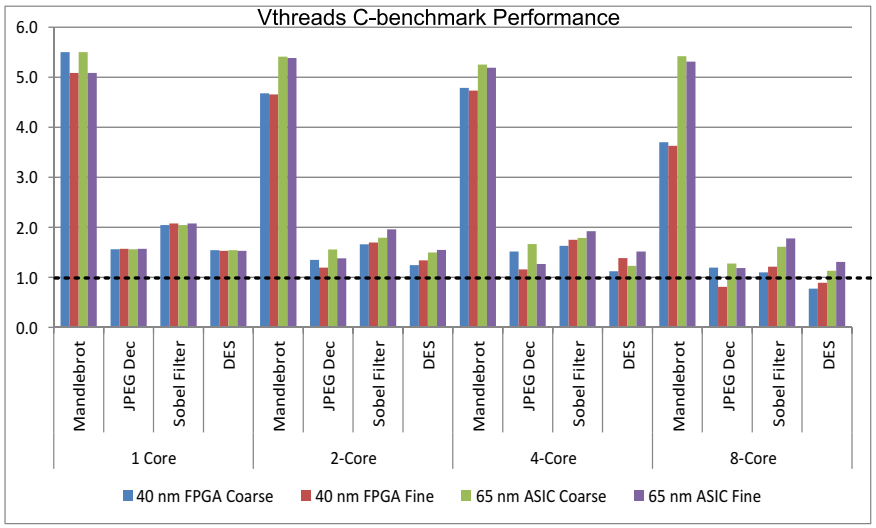


Figure 18: Consolidated results: VThreads speed-up over all processors, all technologies for the C-based benchmarks (All technologies, threading mechanisms)

For the threaded versions of the workloads we note that VThreads maintains a >x5 performance advantage on Mandelbrot against the Leon3MP on standard-cell whereas the FPGA speed-up varies from x4.73-x3.63. For the JPEG decoder, VThreads is 52% faster (4-cores) and 20% faster (8-cores) at coarse-threading (FPGA) with ASIC results being slightly better (66% at 4 cores, 28% at 8 cores). The situation is slightly different for the fine-grained JPEG in which the VThreads is 20% and 16% faster for 2 and 4-cores (FPGA) and slower by 19% at 8-cores. Standard-cell results still demonstrate that VThreads is faster on this benchmark (38% at 4-cores and 19% at 8-cores). VThreads maintains a lead on the highly-parallel coarse-threaded Sobel filter benchmark being 66%-10% faster (FPGA), 79%-61% faster (ASIC). At the fine-threaded level it is faster by 70%-21% (FPGA) up to 96%-78% (ASIC). Finally, for the coarse-grained DES workload VThreads is faster at 2 and 4-cores (by 25%-12%) while being 23% slower at 8-cores (FPGA) with the ASIC results showing that is faster by between 50%-13% (2-8 cores respectively). On the fine-threaded DES workload VThreads is 34% faster (2-cores) and 11% slower at 8-cores (FPGA) with the ASIC results showing it's faster by 51%-31% (2 and 8-cores respectively),

Overall, VThreads with its hardware-assisted PThreads support demonstrates better performance to the Leon3MP system particularly for highly-parallel workloads such as Mandelbrot and Sobel filter; the pattern on more complex benchmarks such as JPEG decode and DES is slightly different with the Leon3MP system

demonstrating better performance at higher core counts (8). This can be attributed to the very fast custom-PThreads implementation in which all cores are active in a very tight polling loop in a shared-memory system whereas in VThreads, the DBG_IF PThreads mechanism in Section 3.6 is a natural synchronization point which can be further optimized if implemented in a pipelined fashion such as the MPI coprocessors of [31].

6 Silicon Implementation Results

An important mandate for this work was the re-architecture of the legacy LE1 CPU to a technology-independent form. This is discussed in this section with three implementations carried out. These include mature 40 nm FPGA, 28 nm SoC-FPGA and 65 nm ASIC technologies. In the latter case, an 8-way Leon3 CMP system was implemented with identical number of CPU cores with VThreads Contexts and utilizing the same memory subsystem (256KB, 4 banks). The use of the VThreads DRAM subsystem was done to level the field and provide a more accurate comparison across the ASIC implementations of the CMPs.

6.1 FPGA implementation

Data were collected for a 40 nm device (Xilinx V6LX240T-FG1156) and a 28 nm SoC FPGA(z7045 device on the Xilinx ZC706 development board). The 40 nm target was synthesized with the older (deprecated) xst-based flow whereas the z7045 target used of the advanced Vivado environment (2014.2). It should be noted that the LX240T-FG1156 results didn't include the re-timing option which further increases the performance of the wide-multiplexer-heavy VThreads design. In addition, the z7045 results don't include data for the Leon3 CPU as the host processor in this case is a high-speed (800 MHz) dual ARM A9 Cortex CPU. Tables 7 and 8 summarize the FPGA implementation results:

Table 7: FPGA Implementation Results. Xilinx V6LX240T (40 nm) - Not-retimed

Processor configuration		FLOPS	LUTs	SLICES	RAMB36E1	RAMB18E1	DSP48E1	Fmax
CPU=1	Leon3	14225	18650	8284	8	28	4	120
Contexts=1	DBANKS2	15147	20501	9319	45	256	7	100
	DBANKS4	15640	20912	9147	37	264	7	100
	DBANKS8	15150	22276	9163	45	256	7	75
CPU=2	Leon3	18,524	26,541	10,900	14	44	8	120
Contexts=2	DBANKS2	18260	29103	11999	69	256	11	75
	DBANKS4	18586	30064	12488	69	256	11	75
	DBANKS8	18079	31920	13093	69	256	11	50
CPU=4	Leon3	27,088	42,215	15,662	26	76	16	86
Contexts=4	DBANKS2	26432	45490	17732	85	288	19	75
	DBANKS4	23635	47346	18441	117	256	19	50
	DBANKS8	24194	49266	20551	85	288	19	50
CPU=8	Leon3	44187	73437	25581	50	140	32	75.9
Contexts=8	DBANKS2	39360	78655	28218	149	320	35	50
	DBANKS4	37617	80534	29033	149	320	35	50
	DBANKS8	34283	91007	30261	213	256	35	25

From Table 7 (40 nm device), VThreads experiences significant Fmax degradation with increasing number of cores and number of data banks, (100 MHz at 1 context, 2 banks down to 50 MHz (8 contexts, 4 banks) unlike the Leon3MP system which is over 50% faster (75.9 MHz) at 8 cores. This is to be expected as Leon3MP is a scalar processor with a blocking, single-port data cache whereas VThreads uses a multi-banked memory system. It should be noted that VThreads is a more advanced architecture and, due to the configurability and extensibility aspects, is by nature more complex. Our experience is that the VIRTEX6LX240T silicon is less forgiving on mux-heavy designs compared to the equivalent Stratix IV 230 device which shows substantially better performance. On the 28 nm part (Table 8) VThreads performs much better with the 8-core, 4-bank configuration being x2.8 faster compared to the equivalent 40 nm configuration. This can be attributed to both

Table 8: FPGA Implementation Results. Xilinx z7045 SoC (28 nm)

Processor configuration		Slices	Slice Regs	Slice LUTs	RAMB36E1	RAMB18E1	Fmax
Contexts=1	DBANKS1	2753	2584	9070	32	2	159.2
	DBANKS2	2583	2573	8646	36	3	153.1
	DBANKS4	2656	2592	8778	36	3	145.9
	DBANKS8	3969	2682	13777	36	3	138.3
Contexts=2	DBANKS1	5541	4956	18029	32	4	123.8
	DBANKS2	5015	4932	16490	40	6	137.4
	DBANKS4	5072	4950	16583	40	6	131.0
	DBANKS8	7817	5116	26658	24	6	123.8
Contexts=4	DBANKS1	10311	9687	34160	32	8	120
	DBANKS2	9751	9642	32445	48	12	121.1
	DBANKS4	9979	9664	33646	48	12	119.5
	DBANKS8	10772	9716	35040	48	12	102.8
Contexts=8	DBANKS1						
	DBANKS2	18926	19069	64934	64	24	119.0
	DBANKS4	19834	19076	66674	64	24	113.9
	DBANKS8	20550	19126	69054	64	24	92.7

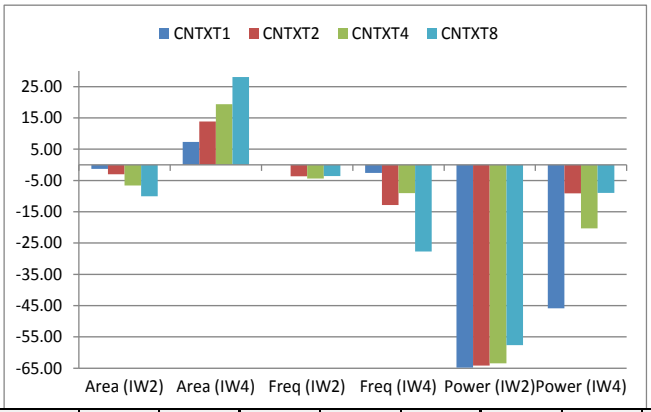
the new synthesis environment (Vivado 2014.2), and the more advanced FPGA fabric.

With the ever increasing silicon mask costs making ASIC engineering a most expensive endeavour for academic departments and industry it is our view that the 28 nm part is a realistic silicon target for combining a mature, stable software/hardware ecosystem (ARM) with the low-latency PThreads support of VThreads. It is noted that at the implementation limit, the z7045 device accommodates 3 homogeneous Systems each with 8 Contexts for a total execution throughput of 24x2 RISCOps at 83 MHz (48 IPC max).

6.2 Standard Cell Implementation

A final comparison can be made between the Leon3MP and the VThreads VLIW CMP. The study was parametric and included automated synthesis and prototype place-and-route flows for various configurations of both CMPs. These included 1-8 core Leon3MP CMP with the VThreads memory subsystem replacing the DDR2 controller, 1-8 Contexts VThreads CMP for both 2-wide (IW2) and 4-wide (IW4) configurations. The ASIC implementation was on the Faraday/UMC 65 nm Library (10M) targeting the LL-RVT (Low-K) UMC process. Typical-case conditions (VCC =1.08V, 25C) were selected for front-end synthesis with Cadence RTL Compiler (RC) V10.10. All Leon3MP and VThreads configurations were constrained for 500 MHz operation (2ns) and clock uncertainty of 100 ps. The *retiming* and *auto-ungrouping* features of RC were used, as well as low-power implementation techniques such as *operand isolation* and *clock-gating*. Front-end synthesis was *RTL-activity-driven* with the activity produced by running the default diagnostics for the Leon3MP system and executing a small C-benchmark on VThreads. These were used to drive front-end dynamic and leakage power optimizations. Finally, to enable more back-end freedom, certain switches were enabled to allow for *Total-Negative-Slack Optimization*. Both designs were automatically floor-planned (prototype flow); Power was measured post-place-and-route using full Signal-Integrity (SI) effects (full extraction within Encounter) and assumed 0.2, 0.5 and 0.5 toggle rates for primary inputs, design flip-flops and clock-gate cells respectively. Back-end synthesis results are summarized in Fig. 19. Fig. 20 presents the floor-plans and VLSI layouts of the CMP designs.

From Fig. 19, the 2-wide VThreads configurations show an area reduction of between 1.26% to up to 10% (8-Context) compared to Leon3MP. The situation is different for the 4-wide configurations which exhibit a silicon area increase from 7.3% to 28%. This is to be expected as the 4-wide IFE LIW packing mechanism is more complex compared to the 2-wide. In terms of max. operating frequency, the IW2 configuration was slower by 4.6% (4-Contexts) with the IW4 configuration being slower by up to 27.7% (8-Contexts). Finally, both 2 and 4-wide configurations exhibit better average power ranging from 64.8%-57.6% (IW2) and 45.8%-9% (IW4). This is to be expected as VThreads overwhelmingly uses clock-enabled registers whereas the Leon3MP uses synchronous resets with (most of the times) re-circulating multiplexers. Finally, the authors note that the final



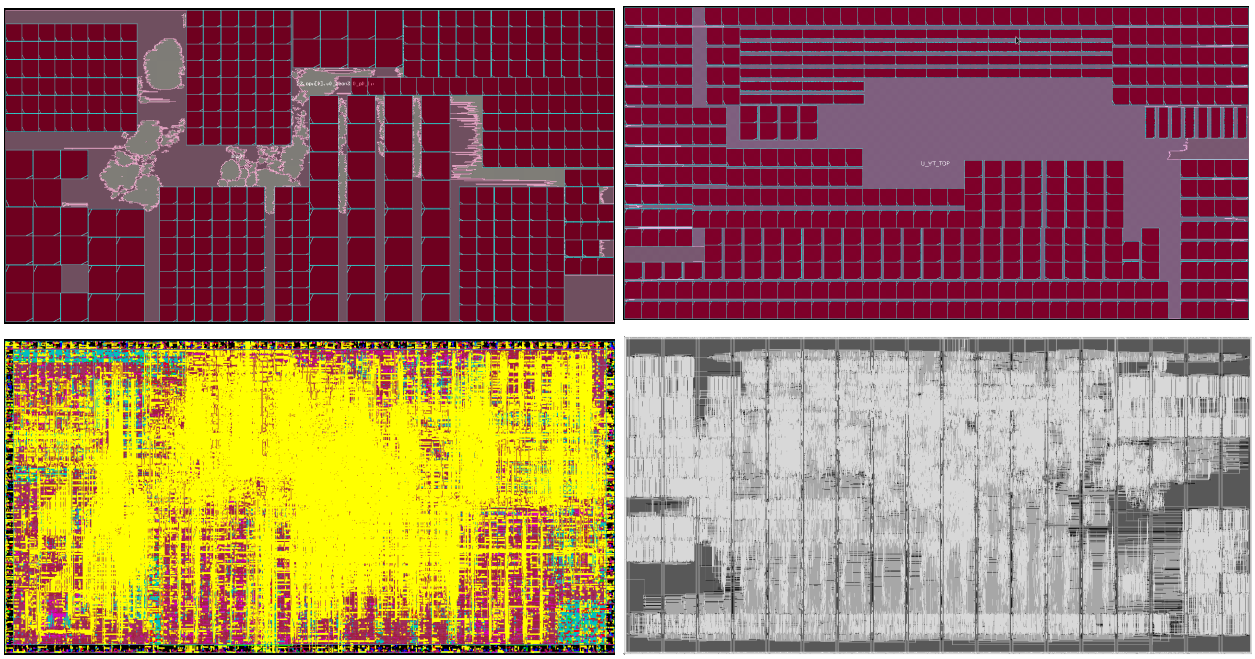
NCPUs	Area (micron sq.)	Fmax(MHz, CDS Enc)	Static Power, mW	Area (micron sq.)	Fmax(MHz, CDS Enc)	Static Power, mW	Area (micron sq.)	Fmax(MHz, CDS Enc)	Static Power, mW
	Leon3MP			Vthreads IW=2			Vthreads IW=4		
1	3698787	500.0	354.3	3652021	500.0	124.6	3968778	487.3	191.91
2	4173178	500.0	407.1	4047688	481.7	146.0	4750432	435.9	369.95
4	5171130	425.9	676.5	4831983	407.3	247.3	6174203	387.5	539.04
6	6140060	424.1	983.5	5621954	402.3	371.2	7644319	364.3	699.62
8	7114809	412.9	1141.5	6401685	398.2	483.5	9111572	298.4	1038.96

Figure 19: Percentual comparison of VThreads (Issue Width 2,4) vs the Leon3MP CMP for equivalent number of CPU cores. The VThreads 256 KB 4-bank memory system was used on Leon3MP

performance can improve further if a bottom-up flow is used. This is currently inhibited by the extensive use of HDL generics both in Leon3MP and VThreads which makes each instance of major blocks such as Context *unique* thus requiring manual intervention. It is expected that removal of the generics would make the back-end process substantially simpler and result in a higher Fmax for VThreads.

7 Related work

The authors in [32] discuss Hthreads, a mechanism to abstract the CPU/FPGA interface and seamlessly execute threaded applications on either using the PThreads programming model. They propose an abstract hardware/thread interface (HWTI) and support the compilation of data-parallel sections of code onto hard-wired accelerators. In addition, the authors in [33] discuss distributed, hardware-based Micro-kernels, based on the PThreads programming model as a framework for heterogeneous, FPGA-based MPSoCs parallel software development. They propose a system where each CPU core includes a Hardware Abstraction Layer (HAL) library and all make use of modified Hthreads hardware micro-kernel cores. They performed their investigation on a heterogeneous MP-SoC system (Xilinx Virtex5 series) consisting of a hard-PPC and multiple soft MicroBlaze processors. Whilst sharing the idea of using the PThreads model to express application parallelism, we note that VThreads is a highly-customizable and extensible, technology-agnostic architecture and as shown, can be targeted to both field-programmable silicon (section 6.1) and standard-cell technologies (section 6.2). At the same time, VThreads provides a full software-based environment and it's performance can be further enhanced via custom MIMO ISEs and pipelined accelerators to closely match the processing requirements of the application. Similar to this work is [34] which makes use of the task flow graph of an application with Kahn Process Networks (KPNs) to efficiently design MPSoCs with hardware accelerators (Hardware Threads, HWTs) for partially-reconfigurable systems and streaming applications. REconos [35] is an extension to the eCos RTOS, making integrated use of software threads and hardware cores (hardware threads) to provide POSIX-compliant services on FPGAs. We note that the authors make use of a single synchronization point (where hardware interact with software threads), similarly to the VThreads DBG_IF FSM being the synchronization point for all pthread operations. The authors propsed the use of distinct calls (e.g., `pthread_create()` and `rthread_create()` for Sw and Hw threads respectively). The latter is not an issue with VThreads as only Sw threads can be created on the unallocated HCs. Along the same lines, the Berkeley Operating system for Re-programmable Hardware (BORPH) [36] provides unified Unix-like interface based on inter-process



(a) 8-core Leon3MP system with a 256KB TCRAM (b) Single-system 8-context/single HC VThreads CMP

Figure 20: VLSI-Macro-LayoutsFloor-plans and VLSI Layouts on UMC65 nm process, typical conditions (1.08V, 25C).

Communication (IPC) mechanisms to FPGA-resident "hardware-processes" for the easy migration of software components onto such devices. The ARPA-MT processor [37] proposes a technology-agnostic, scalar/SMT implementation of the MIPS32 architecture with hardware-support for OS primitives (real-time-clock RTC, task management and semaphore-based synchronization) in the form of system coprocessor 2 (Cop2). The coprocessor implements a register interface, supports a multitude of task manipulation/synchronization instructions and is under the control of the host MIPS CPU. It provides very fine scheduling granularity (5-13 clocks) making it suitable for a number of hard real-time applications. Whilst VThreads wasn't conceived as a real-time processor, it also supports very low latency PThreads primitives and is suited more to high-throughput applications, making use of all available parallelism (ILP, DLP and TLP) combined with hard-wired accelerators. Ziavras et al [31] discuss the use of a pipelined coprocessor for implementing Message-Passing-Interface (MPI) primitives directly in hardware. This work complements the findings of the researchers by investigating the use of a complex FSM-based design to accelerate PThreads operations. Finally, we note [38] as a methodology similar to the ENOSYS project; the authors leverage Object-Oriented (OOP) and aspect-driven (AOP) approaches to specify and shield the system behaviour from the system implementation as a collection of Sw/Hw components. The ENOSYS approach allowed for the tagging of objects as Sw/Hw and, under the control of the DSE environment, the use of Action Language to describe system functionality and the intelligent allocation of such objects into A) Hardware accelerators (behavioural synthesis with FalconML); B) onto software threads executing on the bare-metal LE1 VLIW CMP.

8 Conclusions

This paper discussed VThreads, a novel VLIW CMP designed to provide lightweight OS-like services into deeply-embedded applications requiring very fast thread management capabilities, typically not provided by software implementations. VThreads, building on from it's predecessor, demonstrated better performance and power efficiency compared to the Leon3MP CMP in a variety of workloads whilst providing much improved application optimization opportunities via it's unique configurability. In the course of this work a number of micro-architectural issues were identified and these are noted here as suggestions for future research, to improve the performance of the processor. In particular VThreads relies heavily on single-port memories as the primary

target was standard-cell technologies; modern 28 nm+ FPGAs provide high speed (250 MHz operation on the Kintex7 fabric) dual-port RAM blocks which can be time-multiplexed to provide 4 independent R/W ports. A third-generation micro-architecture will address this and allow for such multi-ported configurations. This would permit fetching from multiple HCs per clock thus improving the performance of the IFE and permitting for true SMT implementations instead of the current VMT approach. Further performance degradation was noted due to the use of a shallow (1-stage deep) pipeline between the clients addressing the banked DRAM. This proved to be a performance bottleneck (particularly on FPGA targets) and will also be addressed. It is noted that the IFE FSM, responsible for the packing of LIWs (and the separation of embedded 32-bit constants) is perhaps overly complicated and limits the frequency of the VThreads design, particularly on 4-wide configurations (Fig 19). On the software side, we plan to provide more support for PThreads primitives and integrate the Trimaran environment to to allow the use of the predicated ISA. A video of a dual-context, dual-issue VThreads system executing the Mandelbrot benchmark in the context of the ENOSYS FP7 project can be seen in https://www.youtube.com/watch?feature=player_embedded&v=Ltp4xWcEqr0.

9 Acknowledgements

This research was partially supported by the EU FP7 ENOSYS project. David Stevens was supported by the School of Electronic, Electrical and Systems Engineering and partly by the IFest EU FP7 project. The authors gratefully acknowledge that support

References

[1] Seyed A. Rooholamin and Sotirios G. Ziavras. Modular vector processor architecture targeting at data-level parallelism. *Microprocessors and Microsystems*, 39(4?5):237 – 249, 2015.

[2] Vassilios A. Chouliaras, Konstantia Koutsomyti, Simon Parr, David Mulvaney, and Mark Milward. Architecture, performance modeling and {VLSI} implementation methodologies for {ASIC} vector processors: A case study in telephony workloads. *Microprocessors and Microsystems*, 37(8, Part D):1122 – 1143, 2013.

[3] G. De Micheli. An outlook on design technologies for future integrated systems. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 28(6):777–790, June 2009.

[4] J. Cong, Bin Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Zhiru Zhang. High-level synthesis for fpgas: From prototyping to deployment. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 30(4):473–491, April 2011.

[5] S. Windh, X. Ma, R.J. Halstead, P. Budhkar, Z. Luna, O. Hussaini, and W.A. Najjar. High-level language tools for reconfigurable computing. *Proceedings of the IEEE*, 103(3):390–408, March 2015.

[6] M. Owaida, N. Bellas, K. Daloukas, and C.D. Antonopoulos. Synthesis of platform architectures from opencl programs. In *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*, pages 186–193, May 2011.

[7] Mingjie Lin, I. Lebedev, and J. Wawrzynek. Openrcl: Low-power high-performance computing with reconfigurable devices. In *Field Programmable Logic and Applications (FPL), 2010 International Conference on*, pages 458–463, Aug 2010.

[8] A. Papakonstantinou, K. Gururaj, J.A. Stratton, Deming Chen, J. Cong, and W.-M.W. Hwu. Fcuda: Enabling efficient compilation of cuda kernels onto fpgas. In *Application Specific Processors, 2009. SASP '09. IEEE 7th Symposium on*, pages 35–42, July 2009.

[9] V. A. Chouliaras S. Moyers, R. Thomson and D. Mulvaney. Uml-based design of a jpeg-ls ip via axilica falconml. In *Proceedings of the Intellectual Property-based Electronic System Conference and Exhibition (IP08)*, Grenoble, France., 2008.

[10] David Mulvaney Robert Thomson, Scott Moyers and V. Chouliaras. The uml-based design of a hardware h.264/mpeg 4 avc video decompression core. In *5th International UML-SoC Workshop (in conjunction with 45th DAC)*, Anaheim Convention Centre, USA, June 2008.

[11] Unified modeling language specification, April 2015.

[12] Uml profile for marte: Modeling and analysis of real-time embedded systems. Online, April 2015.

[13] M. Milward, D. Stevens, and V. Chouliaras. Embedded uml design flow to the configurable le1 multicore vliw processor. In *Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2012 7th International Workshop on*, pages 1–8, July 2012.

[14] V. Kathail, S. Aditya, R. Schreiber, B.R. Rau, D.C. Cronquist, and M. Sivaraman. Pico: automatically designing custom computers. *Computer*, 35(9):39–47, Sep 2002.

[15] R. Thomson, V. Chouliaras, and D. Mulvaney. The hardware synthesis of a java subset. In *Norchip Conference, 2006. 24th*, pages 217–220, Nov 2006.

[16] D. Stevens and V. Chouliaras. Le1: A parameterizable vliw chip-multiprocessor with hardware pthreads support. In *VLSI (ISVLSI), 2010 IEEE Computer Society Annual Symposium on*, pages 122–126, July 2010.

[17] David Stevens, Nick Glynn, Panagiotis Galiatsatos, Vassilios A. Chouliaras, and Dionysios I. Reisis. Evaluating the performance of a configurable, extensible vliw processor in fft execution. In *ICECS*, pages 771–774, 2009.

[18] Arvind, D. August, K. Pingali, D. Chiou, R. Sendag, and J.J. Yi. Programming multicores: Do applications programmers need to write explicitly parallel programs? *Micro, IEEE*, 30(3):19–33, May 2010.

[19] M.S. Schlansker and B.R. Rau. Epic: Explicitly parallel instruction computing. *Computer*, 33(2):37–45, Feb 2000.

[20] R.P. Colwell, R.P. Nix, J.J. O'Donnell, D.B. Papworth, and P.K. Rodman. A vliw architecture for a trace scheduling compiler. *Computers, IEEE Transactions on*, 37(8):967–979, Aug 1988.

[21] D.M. Tullsen, S.J. Eggers, and H.M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Computer Architecture, 1995. Proceedings., 22nd Annual International Symposium on*, pages 392–403, June 1995.

[22] M. Gupta, F. Sanchez, and J. Llosa. Csmt: Simultaneous multithreading for clustered vliw processors. *Computers, IEEE Transactions on*, 59(3):385–399, March 2010.

[23] B. Hubener, G. Sievers, T. Jungeblut, M. Porrmann, and U. Ruckert. Coreva: A configurable resource-efficient vliw processor architecture. In *Embedded and Ubiquitous Computing (EUC), 2014 12th IEEE International Conference on*, pages 9–16, Aug 2014.

[24] Tay-Jyi Lin, Chun-Nan Liu, Shau-Yin Tseng, Yuan-Hua Chu, and An-Yeu Wu. Overview of itri pac project - from vliw dsp processor to multicore computing platform. In *VLSI Design, Automation and Test, 2008. VLSI-DAT 2008. IEEE International Symposium on*, pages 188–191, April 2008.

[25] B.D. de Dinechin, R. Ayrignac, P.-E. Beaucamps, P. Couvert, B. Ganne, P.G. de Massas, F. Jacquet, S. Jones, N.M. Chaisemartin, F. Riss, and T. Strudel. A clustered manycore processor architecture for embedded and accelerated applications. In *High Performance Extreme Computing Conference (HPEC), 2013 IEEE*, pages 1–6, Sept 2013.

[26] P. Faraboschi, G. Brown, J. Fisher, G. Desoll, and F. Homewood. Lx: a technology platform for customizable vliw embedded processing. In *Computer Architecture, 2000. Proceedings of the 27th International Symposium on*, pages 203–213, June 2000.

[27] Chouliaras V. Azorin-Peris V. Zheng J. Echiadis A. Stevens, D. and S. Hu. Biothreads: a novel vliw-based chip multiprocessor for accelerating biomedical image processing applications. *IEEE Trans Biomed Circuits Syst*, 6(3):257–268, Jun 2012.

[28] V.A. Chouliaras, K. Koutsomyti, T. Jacobs, S. Parr, D. Mulvaney, and R. Thomson. Systemc-defined simd instructions for a cmp/smt asic platform. In *Norchip Conference, 2006. 24th*, pages 285–288, Nov 2006.

[29] J. Villarreal, A. Park, W. Najjar, and R. Halstead. Designing modular hardware accelerators in c with roccc 2.0. In *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on*, pages 127–134, May 2010.

[30] Benoit B. Mandelbrot. Fractal aspects of the iteration of z ? ?z(1 ? z) for complex ? and z. *Annals of the New York Academy of Sciences*, 357(1):249–259, 1980.

[31] Sotirios G. Ziavras, Alexandros V. Gerbessiotis, and Rohan Bafna. Coprocessor design to support {MPI} primitives in configurable multiprocessors. *Integration, the VLSI Journal*, 40(3):235 – 252, 2007.

[32] D. Andrews, R. Sass, E. Anderson, J. Agron, W. Peck, J. Stevens, F. Baijot, and E. Komp. Achieving programming model abstractions for reconfigurable computing. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 16(1):34–44, Jan 2008.

[33] J. Agron and D. Andrews. Distributed hardware-based microkernels: Making heterogeneous os functionality a system primitive. In *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on*, pages 39–46, May 2010.

[34] David Watson, Ali Ahmadinia, Gordon Morison, and Tom Buggy. Hardware threading techniques for multi-threaded mpsocs. In *Proceedings of International Workshop on Manycore Embedded Systems*, MES '14, pages 56:56–56:59, New York, NY, USA, 2014. ACM.

[35] Enno Lübbers and Marco Platzner. Reconos: Multithreaded programming for reconfigurable computers. *ACM Trans. Embed. Comput. Syst.*, 9(1):8:1–8:33, October 2009.

[36] R. Brodersen, A. Tkachenko, and H. Kwok-Hay So. A unified hardware/software runtime environment for fpga-based reconfigurable computers using borph. In *Hardware/Software Codesign and System Synthesis, 2006. CODES+ISSS '06. Proceedings of the 4th International Conference*, pages 259–264, Oct 2006.

[37] A.S.R. Oliveira, L. Almeida, and A. de Brito Ferrari. The arpa-mt embedded smt processor and its rtos hardware accelerator. *Industrial Electronics, IEEE Transactions on*, 58(3):890–904, March 2011.

[38] T.R. Muck and A.A. Frohlich. Toward unified design of hardware and software components using **c** !+ ! +. *Computers, IEEE Transactions on*, 63(11):2880–2893, Nov 2014.

Authors

Vassilios A. Chouliaras was born in Athens, Greece in 1969. He received a B.Sc. in Physics and Laser Science from Heriot-Watt University, Edinburgh in 1993 and a M.Sc. in VLSI Systems Engineering from UMIST in 1995. He has been active in the design and implementation of scalar, super-scalar and vector embedded microprocessors for over 16 years. He is a Senior Lecturer

at Loughborough University and conducts research in CPU architecture and micro-architecture, and ESL methodologies amongst others. He was previously a Senior R&D Engineer/CPU architect for ARC International, working on various hardware projects in the embedded CPU domain. Prior to ARC he worked as an ASIC design engineer for a telecommunications organization. He is the Loughborough University Investigator and technical director in the ENOSYS FP7 project in which he contributed the tool-chain, automation and the FPGA System-on-Programmable-Chip (SoPC). He is the architect and designer of the LE1 and VThreads VLIW machines and a founder of Axilica Ltd, a Loughborough University spin-out company commercializing disruptive R&D into high-level (UML) behavioural synthesis.

David Stevens David Stevens is from Leicester. He attended Loughborough University for both his undergraduate and postgraduate studies, receiving a B.Eng. in Computer System Engineering in 2007 and a Ph.D. in 2013. The latter investigated methods of application acceleration and UML-based co-design targeting a VLIW CMP. He is currently employed by the University of Leicester on a knowledge transfer (KTP) scheme and works for a mobile market research company focusing on web technologies.

Vince Dwyer is originally from Manchester. He attended Cambridge University as an undergraduate in Mathematics and York University where he received his doctorate in Theoretical Physics. After periods at Trinity College, Dublin and Warwick University he joined the Department of Electronic and Electrical Engineering, where his is now a Reader in Electronic Devices. He has published over 70 academic papers (over 60 of which are in academic journals) on a variety of topics but which lately have included the reliability and physics of failure of VLSI circuits. His roles currently include Associate Dean (T) and Director of Research Programmes for the Systems Division. He was a Visiting Scientist at the Nanyang Technological University in February/March 2011.