TCEMC: A Co-Design Flow for Application-Specific Multicores

Pekka O. Jääskeläinen*, Erno O. Salminen*, Carlos S. de La Lama†, Jarmo H. Takala*, and Jose Ignacio Martinez†
*Tampere University of Technology, Department of Computer Systems, Tampere, Finland
Email: {pekka.jaaskelainen,erno.salminen,jarmo.takala}@tut.fi
†Universidad Rey Juan Carlos, Department of Computer Architecture, Móstoles, Madrid, Spain
Email: {carlos.delalama,joseignacio.martinez}@urjc.es

***Abstract*—Contemporary embedded systems are often designed as *Multiprocessor System-on-Chips (MPSoC)* which include multiple processors and other peripherals on a single chip. In contrast to general purpose multiprocessors, the design of an embedded MPSoC is usually customized to the requirements of the application domain. The need for fast time to market of new embedded MPSoC designs calls for a rapid design flow of the included customized processors.**

This paper proposes a *Multicore Application-Specific Instruction Set Processor (MCASIP)* co-design flow that exploits parallel programming languages as the application description format. The designer can capture the parallelism of the algorithm and exploit specialized instructions using a single high-level programming language. Parallelism of the designed MCASIP architectures can be scaled both at instruction and task levels, enabling easy exploration of the MCASIP design space. This paper describes the design flow and its key technical challenges, and demonstrates its scalability potential. The presented preliminary results show promise for an efficient multiprocessor design methodology.

I. Introduction

Contemporary embedded systems are often designed as *Multiprocessor System-on-Chips (MPSoC)* that are composed of multiple processors, fixed function hardware accelerators and other peripherals on a single chip. MPSoCs differ from general purpose multiprocessors in the level of customization applied. The design of a new MPSoC is tailored to the requirements of the applications at hand while general purpose multiprocessors aim at providing good overall performance. [1]

General purpose multiprocessors are commonly implemented as *homogeneous* shared memory computers to achieve easier programmability and scalability for certain types of workloads. In contrast, MPSoCs are often *heterogeneous* systems including processors of varying *Instruction Set Architectures (ISA)* and memory hierarchies [1], [2].

The multiprocessors of embedded systems can be customized more freely according to the application domain (e.g. video processing) which is possible in part thanks to the lack of the legacy ISA support burden. In addition to special instructions, the customized multiprocessors can provide varying degrees of instruction, data and task level parallelism in order to meet the performance goals placed by the targeted set of algorithms while staying in a limited area and energy consumption budget. At the same time, programming such devices should be as simple as possible in order to reduce software porting and development costs.

This paper proposes a design flow (later referred to as TCEMC) supporting rapid generation of *Multicore Application-Specific Instruction Set Processors (MCASIP)*. The designed MCASIPs can be implemented, e.g., as standalone processors on FPGAs or as part of larger heterogeneous MPSoCs. The main goal is to enable scalable parallel architecture co-design for programs described in established parallel programming paradigms such as OpenCL or OpenMP. This goal is reached with a compiler supported parallel processor template that combines the ease of programmability enabled by a common shared memory with the ability to optimize shared memory usage with independent per core local memories.

The main contributions of this paper are:

1) Description of a MCASIP design flow based on the use of parallel programming paradigms for application description,
2) a dual-address space processor template that can be easily scaled both at the instruction and task levels, and
3) a design of a threading library suitable for the proposed multiple address space distributed memory template.

The rest of the paper is organized as follows. Section II takes a look at the related work. Section III discusses the processor template from which the designed MCASIPs are instantiated. Section IV describes the programming model of the designed MCASIPs. Section V presents the main steps in the design flow. Section VI evaluates the proposed design flow and the FPGA implementability of the template, and Section VII concludes the paper.

II. Related Work

The related work is threefold. It includes contributions to multicore ASIP design flows, parallel processor architectures and scalable threading runtime library implementations.

To the best of our knowledge, the ASIP customization flows have only until recently concentrated on single core optimization aspects without paying much attention to exploring the homogeneous multicore ASIP space. A good survey of the current processor customization flows can be read from [3]. These customization flows can be used to produce multiple customized cores to form a heterogeneous multicore. The proposed design flow enables the design of customized homogenous-ISA multicores with local and shared random access memories programmed using parallel multi-address space programming languages. The benefits of the homogeneous approach include easier programming and dynamic workload scaling. It should be noted that the cores produced with the proposed design flow can be used as building blocks in heterogeneous systems.

The proposed parallel processor architecture template and its memory hierarchy with private local memories is similar to the *Synergistic Processor Unit (SPU)* of the heterogeneous IBM Cell architecture [4]. The main difference in the SPUs in comparison to our cores is that SPU concentrates on data level parallelism for vectorizable code with their SIMD datapath which includes only limited support for *Instruction-Level Parallelism (ILP)*. In TCEMC, the emphasis is on the more widely exploitable ILP. While ILP is easier to exploit as it does not require vectorizable code for parallelizing operations, it has the drawback of potentially wider instruction word size in comparison to vector or SIMD instructions.

TCEMC shares similar ideas with the NVIDIA GPU architecture and its programming model CUDA [5]. An NVIDIA GPU can include multiple *Streaming Multiprocessors (SMs)*. Each SM consists of several *Scalar Processors (SP)* than can execute one scalar instruction at a time. However, these scalar cores do not execute fully independent instruction streams as their operations are defined by instructions from a single data parallel program – a model NVIDIA calls Single-Instruction Multiple Threads (SIMT). The NVIDIA GPU approach is an efficient solution for massively data parallel applications as it minimizes the instruction stream bottleneck. Our aim, on the other hand, is wider applicability and easier programmability. The TCEMC cores are fully independent, fed with independent instruction streams. On the memory model side, the set of scalar processor access a fast local memory that is shared among them. This resembles our case where multiple function units on a single core share the local memory. Similarly to TCEMC, NVIDIA GPUs require the use of a parallel multiple address-space programming language in order to efficiently exploit the targeted parallel platform.

Finally, plenty of earlier work has been published in the scalable runtime thread scheduler implementation and processor load balancing in the past. The work balancing technique used in the proposed distributed memory threading runtime (Dthreads) is a combined work sharing and work stealing algorithm. Work stealing has been discussed for example, in [6]. The additional work sharing improves the balancing of load for uniform work loads. Our implementation considers the ready queue lock contention problem apparent in higher core counts by distributing the lock load to multiple work queues and using the lock contention to direct the work stealing function. In addition, the emphasis in the Dthreads scheduler is on simplicity due to the major requirement of small instruction memory footprint.

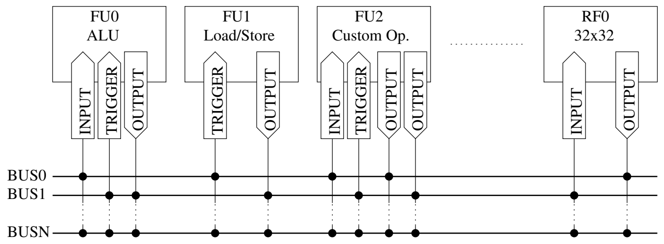


Fig. 1: Example of a TTA processor datapath. The datapath consists of one or more function units, at least one register file and a customizable interconnection network. Data transports are explicitly programmed; a write to a special *trigger port* of a function unit starts the operation execution.

III. Processor Template

This section describes the processor template from which new multicore ASIPs are instantiated in the proposed design flow. The first part describes the main customization points and the processor template used to customize datapath resources inside each core. The second part presents the supported memory models.

A. Customizable Multicore Processor Template

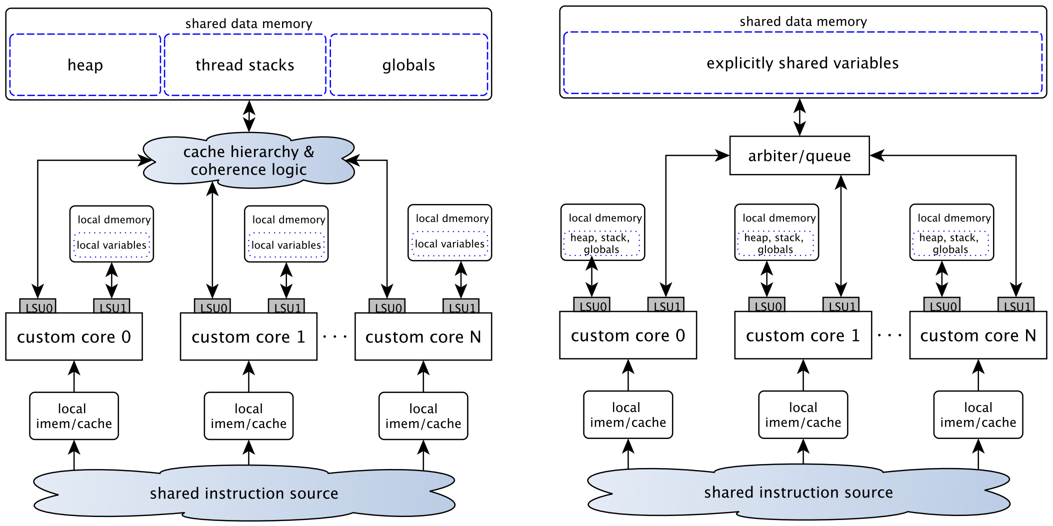
The input programs to the design flow are assumed to express parallel execution, thus the main responsibility for the proposed processor template is to provide adequate level of parallelism at the processor architecture side. Therefore, the support for scalable parallelism was a top priority for the processor template. The second goal was simplicity; as the compiled input program is assumed to express the parallelism explicitly, as little additional hardware as possible should be dedicated to the runtime extraction of it.

TCEMC uses the *Transport Triggered Architecture (TTA)* [7], [8] (see Fig. 1 for an example) as the template for customizing the single cores in the multicore. It has the same properties as the single-core template augmented with extensions to support multiple address space access from higher level languages and minimal homogeneous multicore customization. The single core customization points in the *Architecture Description Format (ADF)* of the single-core customization flow are described in our earlier work [9], [10].

The compile-time exploitable ILP of the TTA is more scalable than with the traditional VLIW architectures [7]. TTA clearly fulfills also the simplicity requirement being one of the most "bare bone" processor design paradigms available [8]. No control or scheduling logic at all is required after the simple decode stage of the instructions.

B. Memory Model

In the traditional terms, TCEMC processor template adheres to the *Harvard architecture* where data and instructions are stored in separate memories. This is beneficial especially in FPGA implementations where the instruction memory can be often implemented with cheaper ROM blocks as the reprogrammability can be implemented with FPGA reprogramming. The only constraint the processor template places to the instruction memory implementation is that each core must be



(a) Shared Default Data Memory (SDDM) configuration.

(b) Local Default Data Memory (LDDM) configuration.

Fig. 2: Two different configurations of the memory model depending on the placement of the default data address space.

able to read one instruction per cycle. Otherwise, the whole core will be locked, leading to reduced performance.

The address spaces were already a customizable feature in the original single core ADF. The described architectures could include one or more load-store units which could each access independent address spaces. TCEMC adds two new customization points to the ADF address space properties:

1) An integer identification number for each address space. Address space #0 denotes the default address space and higher ids denote additional explicitly accessed address spaces. The address space ids are referred to using the explicit address space attributes supported by the used compiler front-end, thus enabling the use of multiple address spaces from higher level languages.
2) Shared/local. In case an address space is marked "shared", it is seen by all cores in the multicore. Otherwise, the address space maps a core local memory.

These new address space customization points, along with a simple integer parameter for the number of cores were enough to add support for homogeneous multicore application-specific processor customization in the ADF.

The data memory model of the TCEMC processor template assumes two or more disjoint address spaces accessed using independent *Load-Store Units (LSU)*. The dual-LSU setup allows accessing both memories in parallel. The choice in which address space the global variables of the program reside is made by the programmer using language-specific *shared qualifiers*.

One of the address spaces is mapped to a global memory that is shared across all the cores in the multicore. The *shared memory* can be used for communication and memory mapped I/O. In addition to the shared memory there is at least one private address space that maps a fast per-core *local memory*.

The choice of mapping the "default address space" of C language (the one which stores the stacks and the heap of the threads) to the shared memory or the local memory leads to two different memory configurations.

a) Shared Default Data Memory (SDDM): Figure 2a represents the generic TCEMC template in a configuration where the default address space is mapped to the shared memory. In this case, the stacks of all threads and the shared heap reside in the global memory leading to increased, and often unnecessary, shared memory traffic. This creates higher demand for a cache hierarchy with potentially complex coherence logic [11] and unpredictable latencies. The main benefit of this configuration is the easy programmability for engineers familiar with shared memory threading libraries and not comfortable declaring variables shared only when needed.

b) Local Default Data Memory (LDDM): The second case is shown in Fig. 2b. When the private local memory of the core is set as the default address space the cores use their local memories for thread stacks and heaps. The main benefit of this is reduced unnecessary (accidental) shared memory traffic. As all shared memory accesses are explicitly defined by the programmer, and data is local by default, higher shared memory access latencies can be tolerated than with SDDM. This means that a simple shared memory queue/arbiter without data caching might be sufficient for controlling the shared memory accesses. Of course, this distinction is not strict since both configurations can use a complex cache hierarchy, or a just simple queue for the shared data memory access. LDDM merely forces the programmer to pay better attention to the shared memory accesses.

LDDM is the recommended memory configuration of TCEMC in case the chosen programming language supports it. The common shared memory threading programming models cannot be used because the default address space is not shared. For example, passing pointers to stack or heap objects to other threads breaks when the receiver thread is executing on a different core. It should be emphasized that shared memory is available for communication also in LDDM but must be explicitly accessed by marking the shared variables with the shared qualifiers in the program.

IV. Programming Model

The MCASIPs designed with TCEMC are not typically assembly programmed. Instead, the applications are described in one of the supported high level programming languages that provide explicit access to multiple address spaces and can launch threads. This section describes the software stack of the proposed design flow and how it is used to implement threading support for the different memory configurations.

A. Software Stack

The simplicity is a major point in the utilized software stack because providing the instruction streams to potentially high number of independent cores is challenging. Low memory footprint software stack allows some designs to use private local instruction memories where the whole software is replicated.

Fig. 3 shows the software stack in the generated MCASIPs. The application is described using one of the supported parallel programming models. Currently the design flow has partial support for Pthreads [12], OpenCL [13], and the embedded C [14] style named address spaces. OpenMP support is planned. In addition, the proposed *Dthreads* library implemented for this design flow (described in the next subsection) can be used directly for programming. In the picture Dthreads is drawn below OpenCL as it can be used to implement the execution of *OpenCL work groups* in multiple (distributed) threads.

The retargetable LLVM-based [15] compiler is responsible for mapping the whole input program along with its threading library to the customized instruction set in the single core customization flow. The code generation supports optional instruction compression to further reduce the required instruction memory size.

B. Dthreads: a Threading Library for LDDM

Traditional C-based threading libraries such as *Pthreads* (POSIX threads [12]) assume all threads to share a single common address space where also the stacks of the threads and the heap reside. This assumption can be seen for example in thread creation; the thread argument data is passed as a pointer to some arbitrary data without size information. The launched thread can then read the data on demand through the passed pointer which can even point to the stack of the launcher thread.

Pthreads work well in the SDDM configuration where the default address space is visible to all the threads in the system. In case of LDDM, each thread is created in a local context and cannot make assumptions in which core it ends up executing.

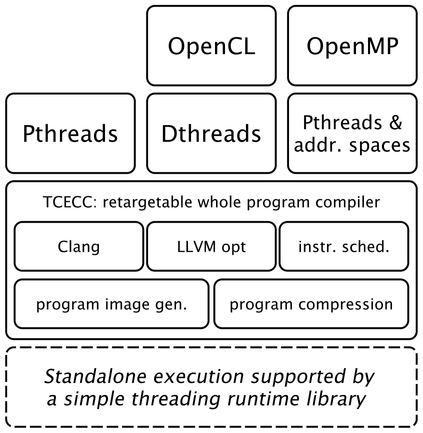


Fig. 3: The software stack of TCEMC. At the highest level, the application is described in one of the supported parallel languages. The LLVM-based whole program compiler compiles the application and the required threading libraries to LLVM bit code which is then converted to executable bit images with the retargetable TCE code generation tools. The compiled program is executed without operating system.

Dthreads was implemented as a threading runtime library optimized for the LDDM configuration of TCEMC. In the programmer's point of view, the API of Dthreads resembles the customary Pthreads API. In fact, the main API difference is the aforementioned thread argument passing example; the thread argument data must be copied to the thread table as there is no guarantee the thread will execute in the same core (thus access the same local memory) it was created in. Therefore, in thread creation, the programmer must pass the size of the argument data along with the argument data pointer.

Fig. 4 shows an example of a Dthreads program that computes a dot product with multiple threads and a parallelizable for loop. It performs the dot product for 800 of 2048-wide *float* vectors using 800 threads. A reader familiar to Pthreads should notice the API similarity. In addition to the aforementioned difference in thread creation (argument data is passed with an additional API call), another notable detail is that the input and output buffers are explicitly marked to reside in the shared memory using the *__shared__* keyword. Other data such as the automatic variables reside in the core local memories.

In addition to the similarity with Pthreads, other design goals for the Dthreads are presented in the following.

1) Low instruction memory footprint: This goal was reached by including only the most important thread programming interfaces in the library and avoiding the use of complex standard library functions such as *malloc()*. The supported functionality include thread creation, joining, the mutex synchronization primitive, thread argument passing and a thread scheduler with threads running always to completion (no preemption). The aggressive full program dead code elimination of the LLVM compiler infrastructure [15] used in the TCEMC compiler removes threading functions not used by the program from the final program image.

```
#include <dthread.h>

#define V_N 800
#define V_WIDTH 2048
#define THREAD_N V_N

volatile __shared__ float a[V_N][V_WIDTH];
volatile __shared__ float b[V_N][V_WIDTH];
volatile __shared__ float products[V_N];


void* dot_prod(void* args) {
    int i;
    int thread_id = *(int*)args;
    products[thread_id] = 0.0f;
    for (i = 0; i < V_WIDTH; ++i) {
        products[thread_id] +=
            a[thread_id][i] * b[thread_id][i];
    }
    return NULL;
}

int main() {

    int tid;
    dthread_t threads[THREAD_N];
    dthread_attr_t attr;
    dthread_attr_init(&attr);
    for (tid = 0; tid < THREAD_N; ++tid) {
        dthread_attr_setdetachstate(
            &attr, DTHREAD_CREATE_DETACHED);
        dthread_attr_setargs(&attr, &tid, sizeof(tid));
        dthread_create(&threads[tid], &attr, dot_prod);
    }
    return 0;
}
```

Fig. 4: Dot product using the proposed Dthreads API.

The measured ROM footprint for a minimal threaded program is in the order of $6 - 16\ kB$, depending on the width of the instructions and the instruction compression scheme of the designed core. The thread scheduling and initialization functionality takes approximately half of this space. Data memory consumption is highly dependent on the max thread function argument size and the thread stack size

2) Autonomous execution: All cores in the multicore are assumed to be identical without any particular control/scheduler processor orchestrating the thread execution. Hence, each core can create and fetch new threads freely. This was implemented with a start-up function that makes one of the cores to execute the first thread that runs the *main()*. All cores initiate an idle thread which is executed when there are no active threads in the system. Thus, other cores wait in their idle threads until the main function creates new threads for them to execute.

3) Minimal shared memory access: Especially in the LDDM configuration, the shared memory accesses are assumed to be very expensive and the likely bottleneck for the throughput of the multicore. Therefore, the threading library must minimize accesses to its book keeping data structures in the shared memory and rely on local memories as often as possible.

Dthreads splits the thread book keeping to two: the *Shared Thread Table (STT)* and the *Local Thread Table (LTT)*. The former resides in the shared memory and contains only unstarted or dead threads. As soon as a core requests for threads to execute, a thread is obtained from the STT to the LTT where also the thread stack is initialized. The thread accesses the STT next time only at its exit or when joining another thread.

In order to support synchronization of the shared memory accesses, the MCASIPs implement the atomic *Compare-And-Swap (CAS)* operation. It allows one core to compare a shared memory location and to update its value conditionally without other cores intervening. No other atomic operations are required to be present in the instruction set of the MCASIP.

4) Scalability: The threading runtime library must adapt to the number of cores available without software modifications. This enables fast experimentation with different number of cores as recompilation of the code for each variation is not necessary. In addition, the performance of the threading runtime routines must not degrade when number of cores or threads are increased.

In TCEMC, the number of cores in the designed processor is a single integer parameter in the architecture description format. Dthreads implementation is unaware of this parameter and does not need any special core identification instruction to be present in the multicore's instruction set. The core identification and scaling is implemented in Dthreads with a global counter in the shared memory. Each core gets their core id from this global counter and copies it to their local memory for later use. The sizes of threading book keeping data structures are not dependent on the total core count, thus making the recompilation unnecessary.

Performance degradation when increasing the core or thread count is avoided by using constant time thread creation and scheduling routines, and by minimizing the critical sections to reduce the time multiple cores wait to access the STT.

In our first attempt in implementing the Dthreads scheduling runtime library, we encountered a severe *lock contention* problem due to a naive STT *Ready Queue (RQ)* implementation. In this version, there was a single RQ from which all cores fetched new threads, thus competed for a single lock to this shared structure. This caused the main thread that tried to create new threads to starve due to the lock being granted also to the idle threads which tried to poll for new work unsuccessfully.

A solution for the lock contention problem was found by combining the *work sharing* and *work stealing* techniques [6] in a scalable shared RQ implementation. In this version, each core has its own RQ in the shared memory, guarded with a separate lock. When new threads are created, they are distributed to different RQs in a round robin manner (work sharing). In the case a core runs out of tasks to execute, it tries to steal work from another core's RQ (work stealing). What is important is that neither thread creation nor work stealing *waits* for locks, but only *tries* to acquire one. In case a core wanting to add or steal threads cannot acquire a lock to a core's RQ at the first attempt, it simply proceeds to the next core's RQ until it manages to lock an RQ. This distributes the lock contention problem to the number of cores in the system, thus improves the scalability of the Dthread runtime scheduler when there are tens or even hundreds of cores competing for

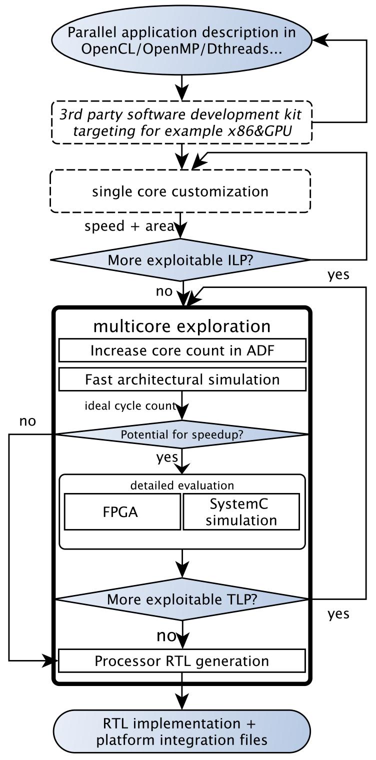

Parallel application description in OpenCL/OpenMP/Dthreads...
3rd party software development kit targeting for example x86&GPU
single core customization
speed + area
More exploitable ILP?
yes
no
multicore exploration
Increase core count in ADF
Fast architectural simulation
ideal cycle count
no
Potential for speedup?
yes
detailed evaluation
FPGA
SystemC simulation
More exploitable TLP?
yes
no
Processor RTL generation
RTL implementation + platform integration files

Fig. 5: The toolset supported design flow of multicore ASIPs with TCEMC. The design flow adds an additional multicore exploration phase after the standard single core customization flow of TCE.

work.

V. Design Flow

The high level design flow is based on an extended version of the TTA-Based Co-design Environment (TCE [10], [16], [17]). TCE was extended to TCEMC, as in "TCE Multicore", by adding features supporting homogeneous multicore simulation and the extensions to its architecture description file format as described in Section III-A. While using the same single core customization flow as the original TCE, TCEMC adds a task level parallelism exploration step. Figure 5 illustrates the MCASIP design flow.

A. Application development and single-core exploration

The first phase is the software development phase. It is often faster to develop the basis for the software using native compilation and execution using a 3rd party *Software Development Kit*. This allows rapid implementation of the software along with its verification data without using an instruction set simulator which is always slower than native execution. In practice, the parallel program can be implemented and tested, for example, on a desktop PC using one of the supported input languages while paying attention to code portability. The output of the software development phase is a verified portable program description, which is then fed into the TCE single core customization design flow.

The single core customization flow outputs a TTA design with datapath resources scaled to the exploitable ILP in the target program. In principle, this part consists of adding datapath resources to the designed architecture using feedback from the retargetable compiler and the architecture simulator. At this stage, also the operation set of the single core can be customized to include special operations that further accelerate the execution. More details on the single core customization can be read in our earlier publications [16], [18].

The customized single core architecture is entered to the multicore exploration phase.

B. Multicore exploration

Multicore exploration consists of increasing the *core count* parameter of the architecture description file and measuring its effects on the processor performance. When entering this phase, the designer also makes a choice between LDDM and SDDM by setting either the local or shared memory as the default data memory. LDDM should be chosen whenever possible as it minimizes unnecessary shared memory traffic. However, if the designer uses certain programming models such as the traditional Pthreads API, the only choice is SDDM.

The designer can test the potential for scalability using the fast architecture simulator that does not include a detailed model of the shared memory microarchitecture but assumes ideal memory access latencies without contention. This allows the designer to measure the potential benefits of adding more cores without going through the rest of the slower more detailed evaluation steps. In case the architecture simulation shows significant enough speedup potential, the designer continues to the detailed evaluation phase. Otherwise, the design flow exits to processor RTL generation.

In the detailed evaluation phase the designer has two options: FPGA evaluation and SystemC simulation. At this point the shared memory hierarchy must be described in more detail to evaluate its effects to the multiprocessor performance. In the case of FPGA evaluation, the processor RTL is automatically generated along with its shared memory hierarchy implementation. The detailed cycle counts are then obtained by running the design on an FPGA. Another alternative for more detailed performance evaluation is a system level simulation where also the memory hierarchy is modeled. This can be done by using SystemC-based simulation. TCEMC provides SystemC hooks for connecting the TTA core architecture simulation model to SystemC simulations, thus enabling incremental simulation detail level addition while still using faster higher-level simulation model for the core.

After the evaluation phase, the effects of the shared memory are known which can lead the designer to modify the memory hierarchy or to optimize the program to exploit local memories more efficiently. In case the FPGA evaluation was chosen, also the FPGA resource consumption and maximum execution frequency is now known which also adds a limit to the number of cores that can be still added. Depending if these results show potential for more task-level parallelism to be exploited, the designer either exits the design flow to RTL implementation generation or goes back to adding more cores to the multicore.

The final step in the design flow is the processor RTL implementation generation step. RTL generation is implemented with an hardware library-based approach where only the custom operations need to be described in an hardware description language. Rest of the implementation is generated automatically for the designer. In addition to producing the RTL implementation, this step also generates the possible files and interfaces required for integration on, for example, different FPGA platforms.

VI. Evaluation

The output from TCEMC design flow is an RTL implementation description which places no constraints on the actual implementation platform. However, it is insightful to provide an assessment of the feasibility of implementing the designed MCASIPs on current FPGA chips. Such estimation is given in the first part of this section. The second part verifies the capability of TCEMC to produce different implementations from the wide MCASIP design space when the lack of parallelism in the input application is not the limiting factor.

A. FPGA Implementation Feasibility Analysis

The size of a single TCEMC core naturally depends on its configuration, ranging from about 1000 *look-up tables (LUTs)* upwards. For example, $5.4k$ LUTs were used in the design published in [16]. Even the smallest low-cost Altera Cyclone FPGAs include about $3k$ LUTs, whereas mid-range Arria II GX devices have already several dozens, and the high-end Stratix IV up to several hundreds [19] of kLUTs. Hence, it easy to see that a MPSoC with dozens or even hundreds of cores is implementable on current FPGA technology. For example, a system with 14 Nios cores has been reported in [20].

Table I provides an approximation on how many MCASIPs could fit to the different category FPGAs. We have taken three FPGA families, selected the smallest and largest devices from each, and listed their sizes in terms of LUTs and embedded memory. In this approximation we assume that each core uses 1200 to 5000 LUTs and 32 KiB of on-chip SRAM for the local memories, and that 90% of the on-chip memory can be effectively utilized. The global shared memory is assumed to be off-chip. From this approximation can be seen that tens of cores can be integrated into a single chip in mid-range and high-end FPGA devices.

Another thing notable in the approximation is that in most of the cases, the maximum number of cores is memory-bound. Hence, the single core configuration does not often make much difference to the maximum core count. The division between memory-bound and logic-bound MCASIPs depends on the core size and the local memory size. For example in Arria II GX, the MCASIP implementation becomes memory-bound when the local memory size is only $8KB$ or larger with a small core. For a 5000 LUT sized core, the local memory size can be $32KB$ before the MCASIP core count is bounded by the embedded memory available instead of the datapath logic. For example with an MCASIP consisting of minimal 1200 LUT cores, less than half of the logic is utilized by the datapath logic, and about 72% or less in case of using a 5000 LUT core.

TABLE I: Approximation on how many cores could fit on a single FPGA chip.

Category	Example	Eq. LUTs	Emb. SRAM [kB]	# of cores
Low cost	Cyclone	3 - 20	7 - 37	0 - 1
Midrange	Arria II GX	45 - 256	425 - 1475	9 - 46
High end	Stratix IV	73 - 813	921 - 4162	14 - 130

B. Multicore Customization

The design space of the customizable homogeneous multicore ASIPs is vast. At one end, there is a dual-core architecture with maximal per-core resources to satisfy the high level of instruction level parallelism (ILP) present in the program. At the other end, there is an army of light-weight cores for a program with only task level parallelism (TLP) or which benefits only from special instruction acceleration.

In order to ensure the interesting points in the design space can be reached, we created a simple benchmark with capability to scale the parallelism at multiple levels. The program implements the dot product of vectors using the LDDM model and the Dthreads API. The code is fully presented as the example in Section IV-B.

The single-core customization flow was used to design three different base architectures as follows:

swfp	A simple core with only an integer ALU. Thus, performs the floating point (FP) operations in software. Other resources include a 16 x 32-bit general purpose register file and 5 transport buses.
swfp2	Same as *swfp* except with double the number of integer datapath resources.
hwfp	Sames as *swfp2* but with an floating point unit (FPU) to speed up the FP computation. This exemplifies the operation set customization capability.

The core count of each of the single core architectures was increased from one core to 16 cores and the minimum processor cycles were obtained using the fast architectural simulator that simulates the LDDM hierarchy but assumes ideal latencies without access conflicts.

The results are illustrated in Fig. 6. The results show that the core count increase helps the software floating point architectures the most, as expected, because of their lower single thread performance. The additional ILP capabilities of *swpf2* is slightly visible with somewhat reduced cycle counts in each core multiplicities.

Due to the control complexity of the software FP emulation code, the compile-time exploitable ILP is rather limited in the input program. Adding the FPU for *hwfp* improved the performance drastically both because of the hardware acceleration itself and also due to increased static ILP from avoiding the need for the hard-to-parallelize FP emulation code. The

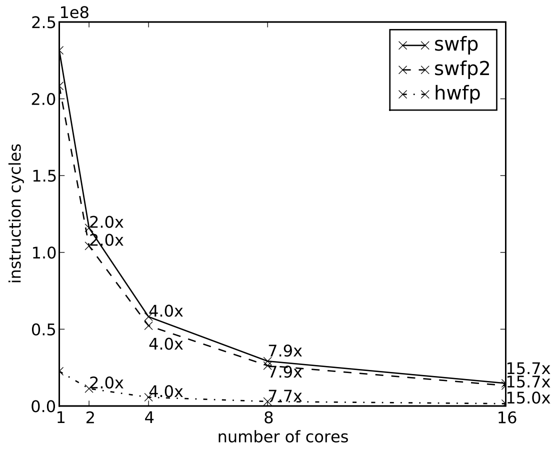


Fig. 6: The effect to the minimum processor cycles from increasing the core count. The potential speedup in comparison to the single core is written on each design point in the diagram.

maximum speedup from additional cores for *hwfp* started to decrease slightly sooner than with the software FP cores as faster thread execution resulted in running out of threads to execute quicker and also made the threading overheads more visible.

VII. Conclusions

In this paper we proposed a design flow for the design of *Multicore Application-Specific Instruction Set Processors (MCASIP)*. At the core of the design flow is a multiple address-space architecture template that enables scalability both at instruction and task levels for parallel input programs. The template allows two choices for the memory model: local default data memory (LDDM) and shared default data memory (SDDM). For the LDDM memory model, the paper proposed a threading runtime library implementation called Dthreads. The presented distributed memory threading runtime implementation plays a key role in the ability to enable task level scalability in case of LDDM.

The preliminary simulation results show high potential for the proposed design flow, the MCASIP template, and the threading runtime library. The FPGA implementability was estimated with the conclusion that the design of soft multiprocessors is feasible using the design flow, with the limiting factor usually being the size of the local memory. MCASIPs with more than hundred cores can be potentially fit to the current high-end FPGA chips.

The next steps in our work include additional benchmarking and FPGA implementations of the MCASIPs generated using the design flow.

Acknowledgment

The 1st author was funded by the Academy of Finland and the Foundation of Nokia Corporation.

References

[1] W. Wolf, A. Jerraya, and G. Martin, "Multiprocessor system-on-chip (MPSoC) technology," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 27, no. 10, pp. 1701 –1713, Oct. 2008.

[2] A. C. Sodan, J. Machina, A. Deshmeh, K. Macnaughton, and B. Esbaugh, "Parallelism via multithreaded and multicore cpus," *Computer*, vol. 43, pp. 24–32, 2010.

[3] I. Paolo and R. Leupers, *Customizable Embedded Processors: Design Technologies and Applications*. San Francisco, CA: Elsevier Inc., 2007.

[4] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki, "Synergistic processing in cell's multicore architecture," *IEEE Micro*, vol. 26, pp. 10–24, March 2006. [Online]. Available: http://portal.acm.org/citation.cfm?id=1130719.1130803

[5] *NVIDIA CUDA Compute Unified Device Architecture: Programming Guide v2.0*, NVIDIA, June 2008.

[6] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *J. ACM*, vol. 46, pp. 720–748, September 1999.

[7] H. Corporaal, "TTAs: missing the ILP complexity wall," *J. Syst. Architecture*, vol. 45, no. 12-13, pp. 949–973, 1999.

[8] ——, *Microprocessor Architectures: From VLIW to TTA*. Chichester, UK: John Wiley & Sons, 1997.

[9] A. Cilio, H. J. M. Schot, and J. A. A. J. Janssen, "Architecture Definition File: Processor Architecture Definition File Format for a New TTA Design Framework," Project Document, Tampere Univ. of Tech., Tampere, Finland, 2003-2006. [Online]. Available: http://tce.cs.tut.fi/specs/ADF.pdf

[10] P. Jääskeläinen, V. Guzma, A. Cilio, and J. Takala, "Codesign toolset for application-specific instruction-set processors," in *Proc. SPIE Multimedia on Mobile Devices*, Jan. 29–30 2007, pp. 65 070X–1 – 65 070X–11.

[11] P. Stenstrom, "A survey of cache coherence schemes for multiprocessors," *Computer*, vol. 23, no. 6, pp. 12 –24, June 1990.

[12] *POSIX, IEEE Std 1003.1, 2004 Edition*, IEEE. [Online]. Available: http://www.unix.org/version3/ieee_std.html

[13] *OpenCL Specification v1.0r48*, Khronos Group, Oct. 2009. [Online]. Available: http://www.khronos.org/registry/cl/

[14] *ISO/IEC TR 18037 - Programming languages - C - Extensions to support embedded processors (draft)*, ISO, April 2006. [Online]. Available: http://www.open-std.org/jtc1/sc22/wg14/www/projects#18037

[15] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. Int. Symp. Code Generation Optimization*, March 20–24 2004, p. 75.

[16] O. Esko, P. Jääskeläinen, P. Huerta, C. S. de La Lama, J. Takala, and J. I. Martinez, "Customized exposed datapath soft-core design flow with compiler support," *International Conference on Field Programmable Logic and Applications*, vol. 0, pp. 217–222, 2010.

[17] "TTA-based codesign environment (TCE) home page." [Online]. Available: http://tce.cs.tut.fi

[18] P. Jääskeläinen, C. Sánchez de La Lama, P. Huerta, and J. Takala, "OpenCL-based design methodology for application-specific processors," in *Embedded Computer Systems (SAMOS), 2010 International Conference on*, July 2010, pp. 223 –230.

[19] "Altera corporation web site, [online]. available: http://www.altera.com," Jan. 2011.

[20] A. Kulmala, E. Salminen, and T. D. Hämäläinen, "Instruction memory architecture evaluation on multiprocessor FPGA MPEG-4 encoder," in *Proc. IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*, Krakow, Poland, Apr. 2007, pp. 105–110.