Support for Data Parallelism in the CAL Actor Language

Essayas Gebrewahid
Centre for Research on Embedded Systems, Halmstad University
essayas.gebrewahid@hh.se

Mehmet Ali Arslan
Lund University, Computer Science
mehmet_ali.arslan@cs.lth.se

Andréas Karlsson
Dept of Electrical Engineering, Linköping University
andreask@isy.liu.se

Zain Ul-Abdin
Centre for Research on Embedded Systems, Halmstad University
zain-ul-abdin@hh.se

Abstract

With the arrival of heterogeneous manycores comprising various features to support task, data and instruction-level parallelism, developing applications that take full advantage of the hardware parallel features has become a major challenge. In this paper, we present an extension to our CAL compilation framework (CAL2Many) that supports data parallelism in the CAL Actor Language. Our compilation framework makes it possible to program architectures with SIMD support using high-level language and provides efficient code generation. We support general SIMD instructions but the code generation backend is currently implemented for two custom architectures, namely ePUMA and EIT. Our experiments were carried out for two custom SIMD processor architectures using two applications.

The experiment shows the possibility of achieving performance comparable to hand-written machine code with much less programming effort.

Keywords SIMD, CAL Actor Language, QRD

1. Introduction

To cope with the high computational demands of DSP applications, high performance embedded computing is shifting to heterogeneous manycores, which incorporate various features, such as reconfigurability, hardware accelerators, memory banks with parallel access, SIMD-based processors, etc. These features are intended to speed up potential bottlenecks of algorithms that frequently occur in DSP applications, e.g. baseband processing, video/audio processing and computer vision. Software developers are left with the challenge of developing applications in a way that takes full advantage of the advances in the hardware [21].

Recent embedded manycores and general purpose processors (GPP) have incorporated SIMD extensions to increase the overall performance of a system. Concurrent models of computation, such as the actor model, can be used to exploit task-level parallelism of application to program manycores. However, to takes full advantage of the hardware, additional levels of parallelism, i.e. data and instruction-level parallelism, within a task or an actor must be employed. In earlier work [11], we have exploited the parallelism between tasks for manycore architectures. In this paper, we focus on Single Instruction Multiple Data (SIMD) support on single processors within a manycore.

There are two widely used methods to exploit instruction- and data-level parallelism. In the first method, the programmer develops the code manually by calling APIs that give direct access to specific hardware instructions. This method requires the programmer to be an expert in both the application and the hardware domain. This leads to high software development cost, and extremely difficult, time-consuming and error-prone coding. The second method requires less involvement of the programmer; it starts with traditional sequential code and uses optimization tools to identify instruction patterns such as loops and replace them with highly optimized instructions. Although compilation techniques, like vectorization, essential to utilize these features have been available for four decades [6, 9, 10], their effective use is insufficient and limited to a specific way of coding [17]. Additionally, our targeted architectures have custom memory organization, and non-regular and complex instruction sets with run-time reconfigurability. This makes it difficult to use the available automatic vectorization technology.

To increase both programmability and resource utilization in manycores with SIMD support, we propose to use a high-level language, such as CAL, with explicit constructs for parallelism, vector type, and vector computation. In earlier works [11], we have used CAL to express the inherent task parallelism of applications and the Cal2Many compilation framework to map applications on manycore architectures.

However, the execution of operations within an actor was sequential, preventing the efficient utilization of architectures with SIMD support. In order to remedy this, we have extended CAL with a vector data type and vector operations. Since recent architectures are incorporating SIMD unit adding SIMD support in CAL enables the language to keep up with the changes in the hardware, both in embedded systems [15, 24] and GPP. It also exposes the hardware features to the programmer through a high-level abstraction that enables the programmer to employ both task and data-level parallelism without the need to be an expert in the targeted architecture.

WPMVP '16, March 13 2016, Barcelona, Spain.

ACM 978-1-4503-4060-1/16/03$15.00.
DOI: http://dx.doi.org/10.1145/2870650.2870656

Our proposed design starts with a set of applications specified in CAL Actor Language that are then mapped to manycore architectures using our CAL compilation framework. Our goal is to provide portable SIMD operations and optimizations for embedded manycores with SIMD support. The next section relates our approach with other methods to program architectures with SIMD support. In Section 3, the details of the programming approach are presented. Section 4 presents the CAL compilation framework. Section 5 presents experimental results and discussions. Finally, Section 6 presents conclusions and future work.

2. Related Work

The number of processors in a chip and the number and type of advanced features within a processor, such as SIMD units, are constantly increasing. In this paper, we focus on SIMD support on single processors within a manycore. Currently, to gain performance via SIMD units most developers rely on auto-vectorization of compilers such as GCC, Clang or Intel C++ Compiler (ICC). However, previous researches show that the SIMD units are underutilized [7, 17]. Manual code tuning is required to take full advantage of the available features. Most compilers allow direct accesses of SIMD units using intrinsics–architecture specific built-in functions. For example, GCC and Clang allow the use of MultiMedia eXtension (MMX), Streaming SIMD Extensions (SSE), 3DNow! and Advanced Vector eXtensions (AVX) x86 extensions by using a data type with vector attributes. Clang also supports OpenCL, PowerPC's AltiVec and ARM's NEON vector extensions using different vector types. The predefined vector types and specific set of operations makes the code non-portable and error-prone. In addition, different architectures have different instruction extension, register size, memory layout and programming styles, that entails that code written for one architecture has to be re-written for the another one, a hinder for portability.

To address the programmability issue, researchers have suggested developing an application by using a common set of macros or libraries that are translated into target specific data types and intrinsic calls. Array Building Blocks (ArBB) [18] addresses programmability and productivity using a C++ API and operator overloading. ArBB is a compilation framework that consists of a low-level C API and a C++ library implemented as an embedded language. ArBB claims to support both data- and thread-level parallelism. Developers use the C++ API to declare Scalar and Collections values. The compilation processes use the collection values to extract SIMD (data) and thread (task) parallelism.

This approach is based on C++, a language designed for a unified memory system and is difficult to adapt for manycores with a distributed memory system. We propose CAL Actor Language to develop applications using encapsulated concurrent actors with SIMD operations to exploit both task and data level parallelism.

Cilk Plus (a language) and OpenMP (an API) are also extensions to C and C++, and support thread- and data-level parallelism. Cilk Plus expresses parallelism using $cilk\_for$ a *for loop* to run iterations in parallel, $cilk\_spawn$ a function call to run the called and the caller function in parallel, and $cilk\_sync$ a barrier that halts the current function till the functions it spawned are complete. For data parallelism, Cilk Plus uses $\#pragma\ simd$ and $\_Simd$ for compiler vectorization of loop bodies, and an extended array notation to enable vectorization via vector operations. Similary, OpenMP uses annotations such as, $\#pragma\ omp\ parallel$, $\#pragma\ omp\ parallel for$, and $\#pragma\ declare\ simd$ to write a multi-threaded application with SIMD operations.

In both Cilk and OpenMP, the programmer pinpoints data and task parallel segments of an application. However, the actual parallelization process including the communication among tasks is automatically managed by the run-time system. This makes the parallelization implicit. In our approach, the programmer explicitly specifies the task and data-level parallelism using CAL actors and SIMD operation within an actor, respectively. The communication among tasks is also expressed by the programmer by connecting input and output ports of CAL actors.

Another common approach is to use a domain specific language (DSL) with program generators. For instance, SPIRAL [19] is an autonomous program generation system that generates vectorized code for linear transforms, like discrete cosine transforms and other DSP algorithms. The transforms and the algorithms are specified as mathematical formulas using a symbolic DSL. These formulas are then used for target-specific automatic code generation and optimizations. The symbolic DSLs have a high degree of expressiveness and could increase productivity. However, they are not well suited for compiling directly down to the manycore architectures; rather, they require transformations via a parallel intermediate representation.

3. The Programming Model

With the emergence of manycore architectures, actor-oriented dataflow programming languages are gaining acceptance; examples include CAL [8], Erlang [2], and SALSA [23]. The dataflow model [20] was introduced as a visual programming language by Sutherland in 1966. In the dataflow model, an application is organized as a flow of data between the nodes of a directed graph. The nodes are computational units, usually called actors or processes. Actors are autonomous, concurrent and isolated entities that execute asynchronously. Edges model channels by connecting explicitly defined actor inputs to outputs. The streams of data that flow through the edges (channels) are called tokens.

We have used CAL Actor Language to define the actors [8] and Network Language (NL) [14] to express the communication among the actors. CAL is a domain specific language that provides high-level abstraction for DSP applications independent of the underlying hardware. RVC-CAL, a subset of CAL, has been adopted by MPEG and ISO as a standard to specify video coding [5]. CAL actors have actions to perform a specific task, input/output ports to communicate with other actors and private variables to record the state of the actor. Actors do not have access to the state of other actors. Thus, interaction among actors happens only via input/output ports. Each execution of an actor may update the private state, consume tokens and produce tokens. During execution, an actor can take different *actions* depending on (1) the availability of tokens on the input port, (2) the actual values of the tokens and (3) the internal state of the actor. These conditions are called the *firing conditions* of an action.

Network Language (NL) sketches the network of the complete CAL application. In NL the programmer has to specify three sections: a *variable declaration section* to define variables that are used as attributes for actors and sub-networks, an *entity section* to declare actors or sub-networks, and a *structure section* to declare the channels that connect the dataflow network.

3.1 SIMD support for CAL

The aim of adding SIMD support in CAL is to enable efficient utilization of manycore architectures with specialized ISA to support vector and matrix operations, e.g. ePUMA [15] and EIT [24]. These architectures target high performance demanding DSP applications such as large antenna systems, imaging, and audio/video processing. These applications operate on a large amount of data with little data reuse and are usually computationally intensive. The applications are inherently massively parallel and usually exhibit both task and data level parallelism. Therefore, the target applications can be modeled as streaming applications that have encapsulated, concurrently operating computational kernels. Here, we can use CAL

actors to model the kernels and NL to express the communication among kernels. Within each kernel, we can use SIMD data types and operations to exploit data level parallelism.

In the backend, depending on the target architecture, the SIMD operations can easily be translated to a specialized hardware accelerator, optimized kernel, or even to an instruction that executes the operation in one cycle. Using the information from the SIMD data types, the memory management tools can easily explore the addressing patterns and configure the data in a way suitable to the underlying hardware.

3.1.1 SIMD data types

SIMD operations provide optimized instructions for repetitive computation that work on a large amount of data characterized by a *vector* type. In most SIMD-based architectures a *vector* is a native data type. In CAL, there is a type for *List*s that is used for arrays of scalar types. Now, in order to explore support for SIMD, we have added a type constructor to interpret a *List* also as a *vector* type. That is, for both vector and array of scalar types, we have used CAL *list* type. The list type is an array of scalar by default. To declare vector type the *make_vector* constructor must be used.

Like array of scalars, *vector* is a derived type that requires a size and a primitive data type. However, *vector* is first class type, i.e. SIMD operations can produce *vectors*.

The main difference between a vector and an array of scalars is that vector types can only be used with SIMD operations, and SIMD operations can only operate on vector types. While programming, a programmer may need to use SIMD operations on an array of scalars or access a specific vector element as a scalar. In this case, the programmer has to use *make_vector* explicitly to construct a vector type for the SIMD operation. Listing 1 shows the CAL code for matrix multiplication. Here A, B and C are 2D vectors and aC is an array of scalars. In the code, *make_vector* is used to define the three vectors and in line 16 to convert an array of scalars to a vector.

```
1  actor MM () ==> :
2   action ==>
3   var
4    List(type:List(type:int, size = 4),size = 4) A :=
         make_vector([[3,4,5,6], [4,5,6,7], [5,6,7,8],
           [6,7,8,9]]),
5    List(type:List(type:int, size = 4),size = 4) B :=
         make_vector(...),
6    List(type:List(type:int, size = 4),size = 4) C :=
         make_vector(...),
7    List(type:int, size = 4) aC
8   do
9     v_transpose(B);
10    foreach int i in 0 .. 3
11    do
12      foreach int j in 0 .. 3
13      do
14        aC[j] := v_dotP(A[i], B[j]);
15      end
16      C[i] := make_vector(aC);
17    end
18  end
19 end
```

Listing 1. CAL actor for Matrix Multiplication.

The restriction and having array and vector types gives the programmer an intuition to come up with data structures and algorithms that are more suitable for SIMD-based architectures. Moreover, the intent of the programmer becomes very clear and directs the compiler to use the convenient data alignment to optimize the code competently. Furthermore, the explicit data access pattern of the *vector* can be used for efficient utilization of load/store operations. This increases the overall performance of the system since the data access is one of the main parameters that affect performance [22].

3.1.2 SIMD operations

The CAL SIMD operations include single instruction vector operations for arithmetic, logic, comparison, and bitwise operations. The operations are defined element-wise and result in a vector of the same size as the arguments. There are also operations where one of the operands can be a *scalar*, in which case, the scalar will be expanded to a vector that has the same size as the vector operand. Additionally, we have backend specific intrinsic functions for frequently used kernels such as discrete cosine transforms (DCT) and other linear transforms. The intrinsic functions and the arguments give the compiler all structural information needed to configure the code in a way that eliminates unnecessary calling and data access overheads. Depending on the hardware resources, a call for an intrinsic function may result in an access to a hardware accelerator or a call for a kernel, i.e., an inlined optimized sequence of instructions. When generating code for ePUMA, we have used ePUMA's kernel library. The library contains FIR, FFT, DCT, matrix inversion, transpose and a number of DSP algorithms implemented in assembly code by an expert. For EIT, we have used the hardware accelerators for division, square-root and COordinate Rotation DIgital Computer.

Binary operations The syntax of the binary SIMD operations uses the ususal binary operators e.g $V_3 = V_1 + V_2$ for element-wise addition and $V_3 = V_1 * S_1$ for scalar multiplication of vector V_1.

APIs For the some SIMD operations we have used APIs e.g $v\_dotP(V_1, V_2)$ for dot product of two vectors. As can be seen in Listing 1, e.g. line 14, we can access individual rows of 2D vectors as 1D vectors and perform SIMD operations. For applications that require non-aligned vector reads, we can use operations such as, sorting, shuffling and masking. The following code snippet shows the use of a boolean vector as a mask and the v_select API to select an element from one of two vectors. Vector v_c selects an element from vector v_a if the corresponding element of the mask is $true$, otherwise it selects from vector v_b.

```
mask := make_vector([false,true,true,false]);
v_a := make_vector([1,2,3,4]);
v_b := make_vector([5,6,7,8]);
v_c := v_select(maks, v_a, v_b}; //[5,2,3,8]
```

Intrinsics As mentioned previously, we include intrinsic functions for frequently used kernels. Depending on the backend, the intrinsics are implemented via hardware accelerators, available SIMD operations, or a sequence of target specific instructions.

Overall, the CAL+SIMD support exposes the inherent parallelism of DSP applications, decreases the involvement of the programmer in low-level programing and gives the compiler more information to enhance the performance through efficient utilization of local memory, parallel data access, DMA transfers and application specific hardware features.

4. CAL Compilation Framework

The goal of the compilation framework is to separate the application development, the scheduling of CAL action firings and target-specific optimizations. An overview of the design flow is shown in Figure 1. The CAL compilation framework (Cal2Many) has three main parts: front end, two intermediate representations, and target specific backends. The frontend performs lexing, parsing, and generates an abstract syntax tree (AST). The frontend represents SIMD

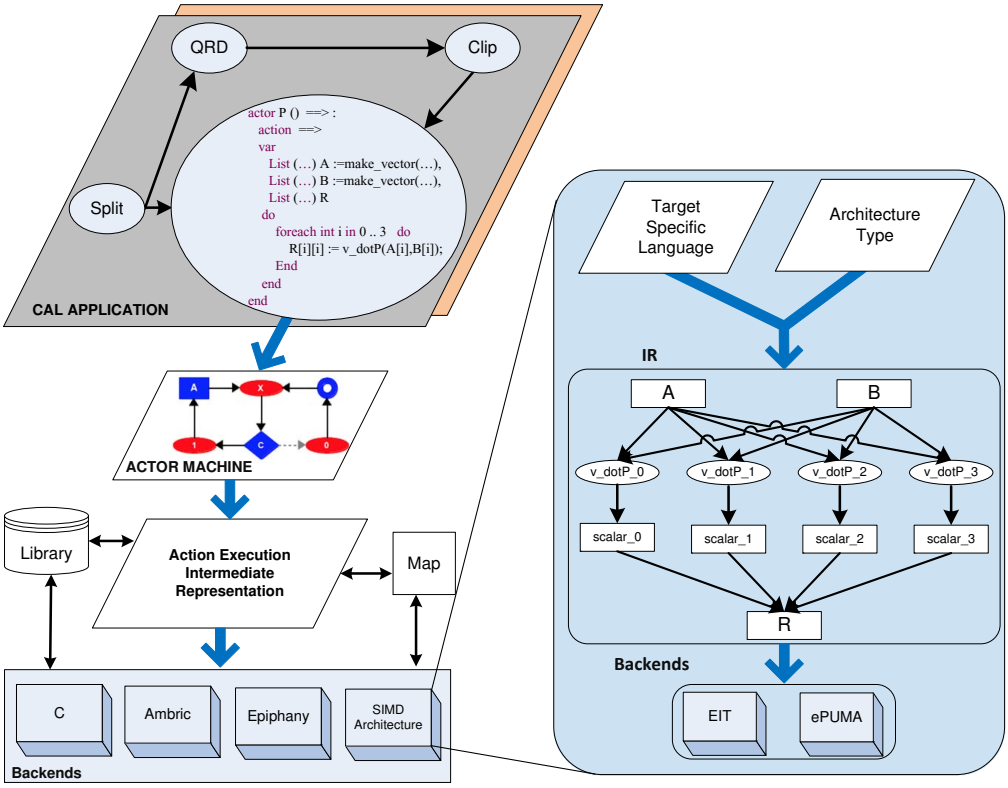

Figure 1. An overview of the design flow focusing on the SIMD backend.

operations and data types in the AST as it is, which gives the backend the required information for competent code generation. The two intermediate representations (IRs) are Actor Machines (AM) [13] and Action Execution Intermediate Representation (AEIR). The AST of each CAL actor is translated to an AM that is then translated to AEIR. The backend performs target-specific transformations and code generation. In earlier work, we have added backends for two manycore architectures and a C backend for general purpose processors [11]. In this work, we have extended Cal2Many by adding a new backend for embedded manycore architectures with SIMD support, such as ePUMA [15] and EIT [24].

4.1 Intermediate Representations

There are two IRs employed in our tool chain, namely Actor Machines (AM) and Action Execution Intermediate Representation (AEIR). AMs can be used for high-level optimizations such as, composing and splitting actors and for dataflow optimizations. AEIR can be used for low-level optimizations, such as loop optimizations, dead code elimination and for inlining functions.

The AM deals with scheduling the testing of conditions and the execution of actions. AMs consist of states that have knowledge about conditions and a set of instructions that can be performed in each state. These AM instructions can be: a *test* to test one of the firing conditions, an *exec* for the execution of an action, or a *wait* to change information about absence of tokens to unknown, so that a test on an input port can be performed after a while.

The next step is the transformation of AM to different programming language constructs, such as function calls to execute the AM instructions, *if* statements to test the conditions and flow control structures to traverse from the current AM state to the destination state. These constructs have different implementations in different programming languages and platforms. Thus, we have chosen to introduce an *Action Execution Intermediate Representation* (AEIR) that brings us closer to a sequential action scheduler and a hardware without having to select a target architecture. The translation of AM to AEIR deals with two main tasks. The first task is the translation of CAL constructs to imperative constructs. This includes CAL actions, variable declarations, functions, statements, and expressions. The second task is the translation of the AM into a sequential action scheduler. This is kept as a separate function that is made up of statements translated from the nodes of the AM and a scheme to traverse from AM states to destination states.

AEIR keeps the task and data-level parallelism explicit as actors and SIMD operations respectively. This makes AEIR suitable to recognize various analysis, optimizations and transformations possibilities for both sequential and parallel portions of a code. Having an explicit representation of SIMD operations enables the compiler to generate a vectorized code without using sophisticated technologies such as automatic vectorization [17]. The SIMD operations are expressed in high-level language which eases the programmability of SIMD architectures and enables portable performance across similar architectures. The operations can be mapped directly to native intrinsic calls or sequential instructions if SIMD operations are not supported. The backends use the SIMD representations to perform target-specific optimizations and to achieve efficient utilization of the SIMD architecture.

4.2 SIMD Backend

Our target platforms are EIT [24] and ePUMA [15], custom architectures with SIMD support. To program the architectures we have translated AEIR to Target Specific Language (TSL), a language that encapsulates the SIMD-like nature of the architectures. The TSL

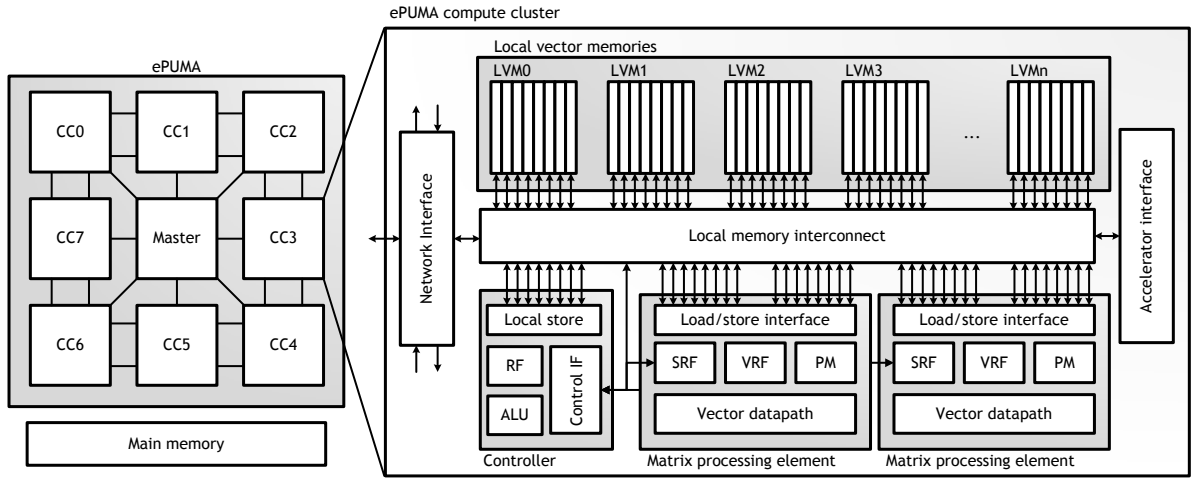

Figure 2. Overview of ePUMA architecture.

code is then compiled to an intermediate representation (IR) which is input to our scheduling and memory allocation procedure. The scheduling and the memory allocation are done in a single constraint programming (CP) model, which produces a schedule with memory allocation for a code generator that turn this schedule into machine code.

4.2.1 Target Architectures: ePUMA and EIT

ePUMA [15] is a heterogeneous architecture that targets digital signal processing applications. Figure 2 shows an overview of the architecture. It comprises a master processor, computing clusters (CCs), and network-on-chip. The master controls the overall application execution. It is responsible for delivering tasks to the clusters and to manage the usage of the off-chip memory and the network-on-chip.

The network-on-chip includes a star network for data transfers from the computing clusters to main memory, a bi-directional ring network for communication between computing clusters and a mailbox system for notification and synchronization.

The computing clusters perform the actual DSP computing. Each computing cluster has shared local vector memories (LVMs)–organized into multiple banks in order to provide conflict free parallel access, a cluster controller–a simple RISC core to manage communication and local memories, and matrix processing elements (MPEs)–single issue cores specialized for vector/matrix computing.

An MPE can operate directly on vectors and matrices stored in the LVMs. It can operate on any vector length, but the performance is limited by the datapath width that is 128 bits. The datapath is a single complex structure with 16 16-bit multipliers, and three levels of arithmetic units that can be interconnected freely. By using appropriate interconnection of multipliers and AUs, we can avoid the need to store intermediate results. This reduces power and improves performance.

EIT [24] is a highly reconfigurable coarse grained architecture that targets signal processing applications related to large antenna systems, aka Massive Multiple Input Multiple Output (MIMO). The architecture comprises a master processor (PE1), five processing elements (PE2-6) and two memory elements (MEs) interconnected via high-bandwidth low latency links. Figure 3 shows an overview of EIT architecture.

PE2-4 and ME2 are used to perform computationally intensive vector operations. PE3 has four parallel processing lanes with

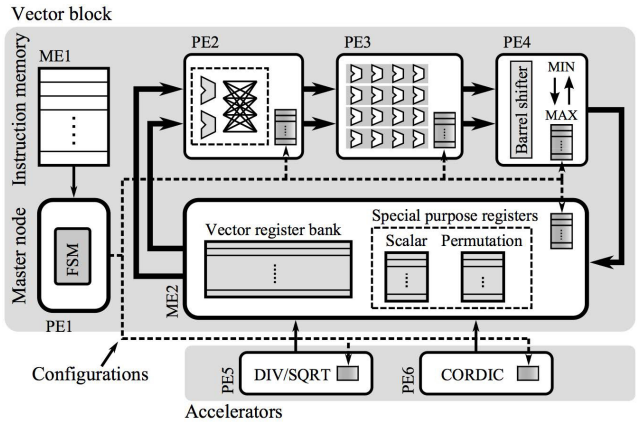

Figure 3. Overview of EIT architecture. Solid and dashed lines depict data and control bus, respectively

complex-valued multiply-accumulate (CMAC) units to performs all vector operations. PE2 and PE4 assist the vector computation by doing pre- and post-processing operations. From the software perspective, the three PEs form a seven stage pipeline that does load (one stage), pre-processing (one stage), vector processing (two stages), post-processing (two stages) and write-back operations (one stage).

The PE5 and PE6 perform scalar operations such as division / square-root and CORDIC (COordinate Rotation DIgital Computer).

The memory is organized in 16 banks to enable parallel access. Banks are further grouped into pages to regulate the access to different lines in the banks.

4.2.2 TSL and IR

In our previous work [3] we have devised a target specific language embedded in Scala, in order to ease programming custom vector architectures. In this work we translate our AEIR to this TSL that generates the intermediate representation (IR) for different architecture types. The IR is fed into the CP model which is responsible for scheduling and memory allocation [3]. Besides the generation of IR, the language enables debugging using any available debugging tool for Scala. For SIMD support, the TSL provides vector

and matrix data types and defines operations and conversions (both implicit and explicit) on them. These types and operations are then converted to the corresponding types and operations supported by the target architecture, represented in the IR.

The IR is a directed acyclic graph G : (V,E) where V denotes operation or data and E denotes the edges which represent the data dependency between the nodes. Nodes can be either operation nodes or data nodes. The graph is also bipartite. Every data node that is not an input of the application, is preceded by one operation node i.e. the operation that produces it. Similarly, every operation node is succeeded by a data node i.e. the data that is produced by it.

The two architectures have a different memory organization that is used as an input to generate an IR with different data nodes. For the EIT architecture, there are two types of data nodes: *vector* and *scalar*. Matrices are expanded into a series of vectors in order to let the CP model schedule and allocate the memory more freely. For the ePUMA architecture, the IR only includes scalar data nodes and operations defined on them. Therefore all the vector/matrix nodes and the corresponding operations are unrolled into scalar nodes and operations. This difference between the architectures is because of their respective memory addressing modes. The vector processor in EIT accesses the vector data through a vector memory that is vector-addressed. On the other hand ePUMA memory addressing is more flexible and can do scalar addressing as well as vector addressing.

4.2.3 Scheduling and memory allocation

The input to the CP model that handles scheduling and memory allocation procedure (hereafter referred to as *the scheduler*) is the IR.

For the EIT architecture, the scheduler assigns a start time for each node, finds a configuration for the vector pipeline on each cycle, and minimizes the schedule length. At the same time, it assigns a memory location for each data node and makes sure that the lifetimes of these data nodes do not overlap.

For the ePUMA architecture, reconfiguration is not an issue while an additional problem is deciding on the memory access patterns. We address this by generating the permutation vectors that are given as parameters to each memory access.

For both architectures, we have to make sure that we respect the precedence constraints between operations and not overload the resources (computational units or the memory) at any time point in the schedule. Precedence between two nodes is represented in the IR as an edge between them. An edge from node i to node j means that i has to be finished before j can start. For resource constraints, we use a couple of well-studied global constraints from the CP paradigm named `Cumulative` [1] and `Diff2` [4], that is used commonly for task scheduling problems.

5. Experimental Case Study

To show the feasibility of our approach, we have generated code for QR decomposition (QRD) and Matrix Multiplication (MATMUL) from a CAL + SIMD implementation and compared it with hand-written implementation. For EIT we used QRD to evaluate the performance of the generated instruction schedule and for ePUMA we generated assembly code for MATMUL and compared the execution time.

The CAL code for MATMUL has been shown in Listing 1. The QRD is based on a modified Gram-Schmidt algorithm [12]. The algorithm produces the upper-triangular matrix R row-by-row and the orthogonal matrix Q as a set of column vectors *q* from the columns of the data matrix A in a sequence of steps. In each step, first we pick a column *a* of matrix A. Then the dot product of this column with itself is calculated. Next, the square root of the result is taken to generate an element of matrix R. This element is later used to normalize the column *a* to produce a column of matrix Q. Finally, the column of matrix A is updated by subtracting a multiple of vector *q* with a value from matrix R. The algorithm for the modified Gram-Schmidt QR decomposition is shown in Algorithm 1.

Algorithm 1 Modified Gram-Schmidt algorithm

for $i = 1 : n$ **do**
  $r_{ii} \leftarrow \sqrt[2]{a_i * a_i}$
  $q_i \leftarrow \dfrac{a_i}{r_{ii}}$
  for $j = (i+1) : n$ **do**
    $r_{ij} \leftarrow q_i * a_i$
    $a_j \leftarrow a_j + r_{ij} * q_i$
  end for
end for

Table 1 shows several characteristics of the resulting schedules for QRD on the EIT architecture. The first column shows the structure of the IR graph: $|V|$ is number of nodes, $|E|$ is number of Edges and $|Cr.P|$ is length of the critical path (the longest sequence of nodes that cannot be parallelized). We incorporated modulo scheduling for overlapping several application instances to utilize the hardware better and improve throughput. Modulo scheduling [16] focuses on finding a schedule for one instance of the application that can be repeated regularly with a fixed interval (a.k.a. initiation interval (II)), respecting the dependencies and resource constraints. The net result of this technique is a more efficient use of the resources, thus yielding a better throughput (calculated as 1/II in the rest of the paper).

For the EIT architecture a reconfiguration is needed when two consecutive instructions are of different types e.g. an addition followed by a multiplication.

Table 1 includes results from two different models w.r.t. the way they handle the reconfigurations. The first model focuses on minimizing II without considering the reconfiguration overhead. The necessary reconfigurations are calculated in a post processing step. To contrast, the second model includes the reconfigurations as a secondary optimization process. The model excluding the reconfigurations is relatively easier to solve and the CP solver finds a schedule with optimal throughput (w.r.t. the model) within a second. The model including the reconfigurations on the other hand, is relatively harder to solve. When the solver is run with a time-out of 10 minutes, it finds a schedule that is only 6 % better than the previous model (*Optimization time* marks the time elapsed when the best solution before time-out was found).

We compare the performance of our schedule, in terms of average throughput, to a schedule manually created by the architecture designer. The manual schedule uses an ad-hoc overlapped execution of several instances of QRD in order to utilize the available resources [3]. The schedule spans an estimate of 300 clock cycles for running 12 overlapped instances of QRD, which corresponds to an average throughput of 0.040. Compared to the manual schedule, our schedule with reconfigurations performs around 20% worse. This is mainly due to the fact that the architect has comprehensive knowledge about the architecture and is capable of fine-tuned optimizations. On the other hand our method goes from an existing CAL-code to a schedule with memory allocation within seconds while the manual scheduling takes many man-hours and is a highly error-prone task.

For ePUMA, we have used matrix multiplication and compared the cycle count for hand-written assembly and generated code from CAL. Table 2 shows the results of an experimental case study of matrix multiplication on ePUMA. A single 4x4 matrix multiplication is not enough to fill the processor with work, so to ensure

Application	$(\|V\|, \|E\|, \|Cr.P\|)$	optimization excluding reconfigurations				optimization including reconfigurations		
		initial II (cc)	# rec.	actual II (cc)	throughput (iter./cc)	II (cc)	throughput (iter./cc)	optimization time (ms)
QRD	(106, 134, 163)	21	12	33	0.030	31	0.032	2692

Table 1. Pipelining with focus on limiting the number of reconfigurations

better processor utilization, we also include results for 2, 4 and 32 concurrent matrix multiplications. The generated code exhibits higher constant overhead and some issues related to memory access. Specifically, the hand-written code uses specialized addressing modes to more efficiently retrieve input data. Still, our generated code adds less than 15% of overhead compared to the hand-optimized assembly in case of 32 concurrent matrix multiplications. As can be seen in the table the CAL overhead decreases significantly with increase in workload, however, after 32 matrix multiplications the overhead stays nearly on the same level. Our current aim is to address the data access related issues, such as using more complex addressing modes, but also extend the implementation to use multiply- and accumulate (MAC) instructions.

Table 2. Comparison of generated code from CAL and hand-written assembly (cycle count).

Version	# of matrix muls.			
	1	2	4	32
Hand-written	52	94	177	1224
CAL	85	118	212	1400
CAL overhead	63.5%	25.5%	19.7%	14.4%

6. Conclusion and Future Work

Actor oriented dataflow programming provides a suitable model for programming manycore architectures in terms of task-level parallelism. Nowadays, manycores are incorporating various features to provide SIMD support. In this work, we have provided a SIMD support for CAL Actor Language to program custom reconfigurable architectures with SIMD-based processors and multi-memory banks with parallel access. As a result, the CAL language can express task parallelism across CAL actors and data parallelism within a CAL actor. The programming support is realized by extending our Cal2Many compilation framework to compile SIMD data type and operation, and by adding the SIMD backend. The experiment shows the practicality of our approach for programming SIMD based manycore architectures. Using CAL + SIMD, a programmer can write competitive code without knowing low-level architectural details and the compilation tool can use the SIMD type and operations for efficient utilization of the parallel hardware.

We have plans to continue evaluation of our appproach using more complex applications such as radar signal processing case studies that integrates both task and data-level parallelism. We will also like to target commercial architectures such as Xeon Phi in order to demonstrate the capability of CAL to exploit both the thread and SIMD parallelism.

Acknowledgments

This work has been supported by grants from The Foundation for Strategic Research and the Swedish national strategic research program ELLIIT.

References

[1] A. Aggoun and N. Beldiceanu. Extending chip in order to solve complex scheduling and placement problems. *Mathematical and Computer Modelling*, 17(7):57 – 73, 1993. ISSN 0895-7177.

[2] J. Armstrong. *Programming Erlang: software for a concurrent world*. Pragmatic Bookshelf, 2007.

[3] M. A. Arslan, K. Kuchcinski, F. Gruian, and Y. Liu. Programming support for reconfigurable custom vector architectures. In *Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores*, pages 49–57. ACM, 2015.

[4] N. Beldiceanu and E. Contejean. Introducing global constraints in CHIP. *Journal of Mathematical and Computer Modelling*, 20(12):97–123, 1994.

[5] S. S. Bhattacharyya, J. Eker, J. W. Janneck, C. Lucarz, M. Mattavelli, and M. Raulet. Overview of the MPEG reconfigurable video coding framework. *Journal of Signal Processing Systems*, 63(2):251–263, 2011.

[6] T. A. Budd. An APL compiler for a vector processor. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 6(3):297–313, 1984.

[7] J. Castrillon, L. Thiele, L. Schorr, W. Sheng, B. Juurlink, M. Alvarez-Mesa, A. Pohl, R. Jessenberger, V. Reyes, and R. Leupers. Multi/many-core programming: where are we standing? In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, pages 1708–1717. EDA Consortium, 2015.

[8] J. Eker and J. W. Janneck. CAL language report: Specification of the CAL actor language. Technical Memorandum UCB/ERL M03/48, University of California, Berkeley, CA, USA, 2003.

[9] F. Franchetti and M. Puschel. A simd vectorizing compiler for digital signal processing algorithms. In *Vehicle Navigation and Information Systems Conference, 1993., Proceedings of the IEEE-IEE*, pages 7–pp. IEEE, 1993.

[10] C. Galuzzi and K. Bertels. The instruction-set extension problem: A survey. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 4(2):18, 2011.

[11] E. Gebrewahid, M. Yang, G. Cedersjo, Z. U. Abdin, V. Gaspes, J. W. Janneck, and B. Svensson. Realizing efficient execution of dataflow actors on manycores. In *Embedded and Ubiquitous Computing (EUC), 2014 12th IEEE International Conference on*, pages 321–328. IEEE, 2014.

[12] G. H. Golub and C. F. Van Loan. *Matrix computations*, volume 3. John Hopkins Press, 1996.

[13] J. Janneck. A machine model for dataflow actors and its applications. In *45th Annual Asilomar Conference on Signals, Systems and Computers (ACSSC)*, pages 756–760. IEEE, 2011.

[14] J. W. Janneck. *NL–a network language*. ASTG, Processing Solutions Group, Xilinx Inc, 2006.

[15] A. Karlsson, J. Sohl, and D. Liu. epuma: A processor architecture for future dsp. In *Digital Signal Processing (DSP), 2015 IEEE International Conference on*, pages 253–257. IEEE, 2015.

[16] M. Lam. Software pipelining: An effective scheduling technique for vliw machines. In *ACM Sigplan Notices*, volume 23, pages 318–328. ACM, 1988.

[17] S. Maleki, Y. Gao, M. J. Garzaran, T. Wong, D. Padua, et al. An evaluation of vectorizing compilers. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 372–382. IEEE, 2011.

[18] C. J. Newburn, B. So, Z. Liu, M. McCool, A. Ghuloum, S. D. Toit, Z. G. Wang, Z. H. Du, Y. Chen, G. Wu, et al. Intel's array building blocks: A retargetable, dynamic compiler and embedded language. In *Code generation and optimization (CGO), 2011 9th annual IEEE/ACM international symposium on*, pages 224–235. IEEE, 2011.

[19] M. Püschel, J. M. Moura, J. R. Johnson, D. Padua, M. M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gačic, Y. Voronenko, et al. Spiral: Code generation for dsp transforms. *Proceedings of the IEEE*, 93(2): 232–275, 2005.

[20] W. R. Sutherland. On-line graphical specification of computer procedures. Technical report, DTIC Document, 1966.

[21] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3):202–210, 2005.

[22] D. Talla, L. K. John, and D. Burger. Bottlenecks in multimedia processing with simd style extensions and architectural enhancements. *Computers, IEEE Transactions on*, 52(8):1015–1031, 2003.

[23] C. A. Varela, G. Agha, W.-J. Wang, T. Desell, K. El Maghraoui, J. LaPorte, and A. Stephens. The SALSA programming language 1.1. 2 release tutorial. *Dept. of Computer Science, RPI, Tech. Rep*, pages 07–12, 2007.

[24] C. Zhang. *Dynamically reconfigurable architectures for real-time baseband processing*. PhD thesis, Lund University, 2014.