Exploiting Thread and Data Level Parallelism for Ultimate Parallel SystemC Simulation

Tim Schmidt, Guantao Liu, and Rainer Dömer
Center for Embedded and Cyber-Physical Systems
University of California, Irvine, USA

Abstract— Most parallel SystemC approaches have two limitations: (a) the user must manually separate all parallel threads to avoid data corruption due to race conditions, and (b) available hardware vector units are not utilized. In this paper, we present an advanced compiler infrastructure for automatic parallelization of SystemC models at the thread-level. In addition, our infrastructure exploits opportunities for data-level parallelization. Our experimental results show a nearly linear speedup of NxM, where N and M denote the thread and data-level factors, respectively. In turn, a 4-core multi-processor achieves a speedup of up to 8.8x, and a 60-core Xeon Phi processor reaches up to 212x.

1. INTRODUCTION

The functional complexity of embedded systems has increased dramatically over the last years. Additionally, many complex design properties like energy consumption or thermal heating must be considered during the design process. Designers use simulation as a tool to validate all kinds of characteristics of their prototypes. The combination of the increasing complexity and the number of validated properties makes the simulation intensively time-consuming.

SystemC [1] is the de facto standard for modeling, simulating, and validating embedded systems. The Accellera reference implementation performs simulations in a sequential fashion. In other words, at any time of the simulation at most one simulation thread is active. SystemC TLM-2.0 provides the concept of time decoupling to speed up simulations. Unfortunately, the benefit of a speed boost comes at the price of simulation inaccuracy [2]. Other work such as [3] and [4] propose a modified SystemC kernel that supports multithreading. However, designers must manually identify and resolve all potential race conditions in their models. Also, they have to ensure that the design is thread-safe. Consequently, overlooked and not protected race conditions often lead to incorrect simulation results.

We propose an automated compiler approach for parallelizing the simulation using *thread* and *data-level parallelism* to save simulation time. This approach is in contrast with existing works that require manual code transformation. First, our SystemC compiler [5] performs a fully automatic analysis to identify and exploit the available *thread-level parallelism*. Additionally, our compiler performs an analysis for *data-level parallelism* based on our SystemC-aware internal representation. The outcome is a report that lists source code locations for vectorization optimization. Finally, our associated parallel SystemC library simulates the thread-safe design in out-of-order parallel fashion similar to [6]. The data-level analysis is based on and guided by thread-level analysis which is aware of PDES and specifically SystemC semantics. Note that this distinguishes our compiler from general-purpose compilers (such as Intel icpc) that cannot automatically identify SIMD parallelism in the source code.

To the best of our knowledge, this work is the first to apply and exploit SIMD vectorization on top of thread-level parallel SystemC simulation.

1.1 Problem Definition

The official simulation kernel of SystemC schedules the individual threads in a sequential fashion although parallel multi- and many-core platforms are available. Consequently, the simulation time of modern embedded systems becomes a bottleneck in the design flow. Simulations may run for hours until unexpected events occur or the simulation crashes. Designers try to fix the behavior and then they have to rerun the simulation from the beginning. The results of their adjustments are available only after another long wait.

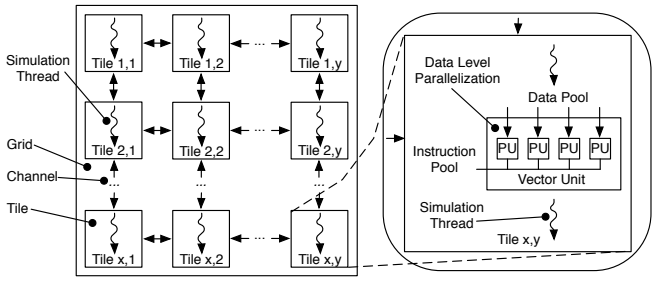

Figure 1: General structure of a Network-on-Chip design model with vector units in tiles

Many attempts have been made to perform SystemC simulations in parallel. Most of these solutions have two major limitations. First, the designer has to manually partition his design into different domains. Through the lack of automation for parallelization, designers merge the orthogonal concepts of design methodology and techniques to save simulation time. Second, traditional parallel discrete-event simulation is limited to thread-level parallelism. Vectorization units for high-performance computing are not utilized although they have been available in standard PCs for 8 years. General-purpose compilers like the Intel icpc have significantly higher difficulty in identifying potential parallelism because they are not aware of the intricate SystemC semantics.

DAC '17, June 18-22, 2017, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/3061639.3062243

Network-on-Chip (NoC) design is a popular pattern in the embedded systems domain. Figure 1 illustrates a typical NoC platform that is assembled by a hosting grid and tiles which are organized in rows and columns. A tile communicates with the other tiles located to the north, west, south, and east. This architecture is well-suited for distributed algorithms. Often, tiles follow the same computation pattern but they are computing different parts of the same problem. Such a NoC supports parallel simulation in multiple ways. On one hand, *thread-level parallelism* is given through the individual tiles which process in parallel. On the other hand, each tile can perform faster through *data-level parallelism*.

In this paper, we propose an automatic solution to run simulations in an ultimate parallel fashion. We extend the SystemC compiler [5] which first reads a design and identifies the associated design hierarchy. Next, it performs conflict analysis to identify potential race conditions and partitions each thread into a set of segments [6]. Each segment considers all statements that can be executed in a scheduling step of a thread. In addition to the thread-level parallelism, we add analysis to identify data-level parallelism. More specifically, our compiler identifies candidates for vectorization in the design by taking the information from the thread-level analysis into account. Finally, the designer can decide whether or not the identified code locations are suitable and worthwhile for vectorization.

1.2 Related Work

Parallel discrete-event simulation is a well-studied subject. Initial work has been contributed by [7] on thread-level parallelism.

The concept of a segment graph for parallel simulation of threads was first introduced in [6] for synchronous and out-of-order parallel simulation. Later, the segment graph infrastructure was used for may-happen-in-parallel analysis for safe ESL models in [8]. In contrast, we are using the segment graph to identify and exploit *thread* and *data-level parallelism* to achieve higher simulation performance.

Time decoupling is a widely-used method which can speed up the simulation of SystemC models. Parts of the model execute in an unsynchronized manner for a user-defined time quantum. However, this strategy generally suffers from inaccurate simulation results [2]. [4] and [9] propose a technique to parallelize time-decoupled designs. This technique requires the designer to manually partition and instrument the model in segments of time-decoupled zones. The authors state "Preparation for time-decoupled parallel simulation was done in less than one person-day." [9]. In contrast, our approach supports a 100% accurate simulation of designs and our compiler automatically instruments the design for parallel execution in minutes. In other words, the designer does not need to be familiar with the design to perform a safe and fast parallel simulation.

A tool flow for parallel simulation of SystemC RTL models was proposed in [10]. The model was partitioned according to a modified version of the Chandy–Misra–Bryant algorithm [11]. In contrast, our conflict analysis considers the individual statements of threads. Also, our solution is not restricted to RTL models.

An approach of static analysis for simulation of SystemC models on GPUs was provided in [12]. In contrast, our approach performs an automatic analysis on the thread *and* data-level parallelism. [13] proposes an automated tranformation of RTL applications to GPUs. Here, it is required that the RTL input description is synthesizable.

2. PARALLEL COMPUTATION

Parallel computation is a general strategy to optimize the execution time of programs. This technique is applied on various levels e.g. *instruction-level*, *data-level*, and *thread-level parallelism*, as well as through *distributed computing*. The objective of exploiting *instruction-level parallelism* is to maximize instruction throughput. Here, a CPU is organized as a pipeline, such as the traditional RISC hardware architecture. Each instruction is partially executed at a different stage in the pipeline. *Data-level parallelism* is often associated with the term *single instruction multiple data* (SIMD). Multiple operations of the same kind are executed in parallel with different data sets. This technique is often used to parallelize low-level loops. At the *thread-level parallelism*, the individual threads of a program are executing in parallel fashion. It must be verified that no race condition occurs among the individual threads. Otherwise, the simulation is compromised and incorrect behavior occurs. Finally, the computation can be distributed over a network of PCs which is called *distributed computing*.

2.1 Thread Level Parallelism

The thread-level analysis identifies potential race conditions between the individual threads. In detail, we partition each thread into segments. A segment considers all potentially executed expressions between two scheduling steps. A new scheduling step is triggered with a `wait()` function call which yields control back to the simulation kernel. Figure 3 shows a segment graph of the simple source code in Figure 2.

```
0 void foo() {
1   index++;
2   wait(1,SC_NS);
3   k=i+j;
4   if(test){
5     cnt2++;
6     wait(2,SC_NS);
7     cnt3++;
8   } else {
9     cnt=iter+k;
10  }
11  sqr=sqr*sqr;
12  wait(3,SC_NS)
13  iter2=sqr+9; }
```

Figure 2: Example Source Code

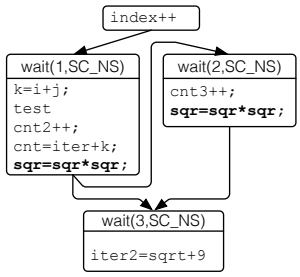

Figure 3: Segment Graph of Figure 2

The segment graph has similarities with a control flow graph but differs significantly in semantics. A new segment is only started when a `wait()` function call occurs. For instance, the exemplary graph in Figure 3 has four different segments where the first segment is the initial one and the three others are initiated with `wait()` calls. Also, it is possible that a statement occurs in multiple segments. The segments starting in line 2 and line 6 both include the statement `sqr=sqr*sqr`.

After the segment graph is created for all threads, the data conflict analysis takes place. All read and written variables are analyzed for each segment. Next, all segments are compared for possible read-after-write and write-after-write conflicts. Finally, we pass the conflict table to the parallel simulator for decision making at run time.

2.2 Data Level Parallelism

Vectorization, a.k.a. parallelization utilizes additional hardware units in parallel *lanes* to speedup the computation. Multiple operations of the same type are executed with different data in parallel. For instance, the Intel Advanced Vector Extensions (AVX) have a data width of 256 bit and they can process up to eight 32 bit integer operations or four 64 bit double operations in parallel. Due to some overhead of arranging data in lanes, data-level parallelism pays off best in loops.

Various criteria must be satisfied to vectorize a loop. First, the loop contains only straight-line code. In other words, each lane in the vectorization unit must perform the same operation. On one hand, `goto` and `switch` statements are not allowed because for each loop iteration the control flow would change. On the other hand, `if` statements are allowed, if they can be transformed into *masked* assignments. Second, vectorization includes unrolling of loops. So a loop can be only vectorized if the number of loop iterations is countable. Additionally, no loop index variable can be written in the loop body. Next, no data dependent exit conditions are allowed e.g. `if(...)break;`. Finally, no backward-carried data dependencies (e.g. `A[i]=A[i-1]+2;`) can be in the loop body. A full list of vectorization requirements is available in [14]. We do not follow function calls of built-in functions that are available as vectorized versions. Built-in mathematical functions, such as `sqrt()` and `sin()` are supported by the Intel *icpc* compiler for vectorization.

Listing 1 shows the structure of a *canonical* loop. The code fragment can be easily vectorized. Listing 2 shows two nested loops where the inner loop cannot be parallelized due to the data conflicts. On one hand, the inner loop writes two variables. First, the scalar `k` is modified, which cannot be vectorized. The parallel writing of the individual vector units causes conflicts. Second, the writing of `array[i]` cannot be parallelized either. The inner loop iterates over `j` and the writing of `array[i]` is interpreted as writing of a scalar. On the other hand, the outer loop *can* be parallelized. First, the variable `k` is a local variable in the loop body. Consequently, this variable is independent from the outer loop iterations. Second, the inner loop wont be parallelized when the outer loop is parallelized.

```
1 int array[10];
2 for(int i=0;i<10;i++){
3   array[i]=42;}
```

Listing 1: Canonical loop

```
1 int array[10];
2 for(int i=0;i<10;i++){
3   int k;
4   for(int j=0;j<10;j++){
5     k++;
6     array[i]=42;}}
```

Listing 2: Nested loop

Our heuristic analysis produces a list of loops which are potential candidates for vectorization. To confirm this, the designer must annotate them in the source code. Specifically, a `#pragma simd` is inserted before the loop to be vectorized. We should emphasize that this manual interaction is necessary because data-level parallelization requires application knowledge that only the designer can provide. The underlying compiler (the Intel icpc) can cover only C++ semantics. Our proposed compiler adds SystemC interpretation, but application-specific knowledge can only come from the human designer.

Algorithm 1 shows our proposed heuristic to identify data-level parallelism in a SystemC application. The recursive algorithm takes as input a statement s and analyzes if all reachable statements from s are vectorizable. The return value is a tuple where the first element indicates whether or not the statements are vectorizable. The second element is the list of the reachable statements. We provide as input statement the function body of each individual simulation thread. This information is only available with the support of our SystemC-aware compiler which also performs the thread-level analysis.

Algorithm 1 Identification of SIMD loop candidates

```
1: function VECTORIZABLE(s)
   returns (isVectorizable, expressions)
2:     if s ∈ {goto, continue, break, label} then
3:         return (NotVectorizable, ∅)
4:     end if
5:     if s ∈ {variabledeclaration, expression} then
6:         (r1, e1) ← ( Vectorizalbe, EXPR(s))
7:         for all f ∈ FUNCCALLS(s) do
8:             (r2, e2) ← VECTORIZABLE(GETFUNCDEFBODY(f))
9:             (r1, e1) ← (r1 ⊕ r2, e1 ∪ e2)
10:        end for
11:        return (r1, e1)
12:    end if
13:    if s ∈ {if} then
14:        (r1, e1) ← VECTORIZABLE(IFBLOCK(s))
15:        (r2, e2) ← VECTORIZABLE(ELSEBLOCK(s))
16:        return (r1 ⊕ r2, CONDITION(s) ∪ e1 ∪ e2)
17:    end if
18:    if s ∈ {switch} then
19:        for all c ∈ CASES(s) do
20:            VECTORIZABLE(c)
21:        end for
22:        return (NotVectorizable, ∅)
23:    end if
24:    if s ∈ {compound} then
25:        (r1, e1) ← (Vectorizable , ∅)
26:        for all s′ ∈ STATEMENTS(s) do
27:            (r2, e2) ← VECTORIZABLE(s′)
28:            (r1, e1) ← (r1 ⊕ r2, e1 ∪ e2)
29:        end for
30:        return (r1, e1)
31:    end if
32:    if s ∈ {for, while, do-while} then
33:        if ¬ISCANONICAL(s) then
34:            return (NotVectorizable, ∅)
35:        end if
36:        (rb, eb) ← VECTORIZABLE(BODY(s))
37:        rc ← CONFLICTCHECK(eb), INCRVAR(s))
38:        if (rb = Vectorizable ∧ rc = Vectorizable) ∨
39:          (rc = Vectorizable ∧
40:          rb ∈ {ScalarConflict, VectorConflict}) then
41:            Recommend this loop for vectorization
42:            return (Vectorizable, eb)
43:        end if
44:        if rc = VectorConflict ∧ rb = VectorConflict then
45:            return (VectorConflict, eb)
46:        end if
47:        if rc = ScalarConflict ∧ rb = ScalarConflict then
48:            return (ScalarConflict, eb)
49:        end if
50:        return (NotVectorizable, ∅)
51:    end if
52: end function
```

In Algorithm 1, our classification in the first tuple element has four different states which belong to three different classes. The classes are *Vectorizable*, *NotVectorizable*, and *Maybe* where *Maybe* includes *ScalarConflict* and *VectorConflict*. *NotVectorizable* considers the situations where the statement s includes directly or indirectly control flow statements such as `switch`, `goto`, `break`, or `label`. These jump statements cannot be vectorized. The class *Maybe* considers expressions which cannot be vectorized for the current loop but potentially for enclosing loops. `ScalarConflict`

reflects the situation in Listing 2 in Line 5. The loop writes the scalar `k` and cannot be vectorized. However, a potential outer loop can be vectorized as in Line 4. *VectorConflict* describes a similar situation where an array is written but the loop variable does not match the array index variable, as loop variable `j` and array index `i` in Listing 2. The status *Vectorizable* indicates that the statement is suitable for vectorization.

We introduce the operator $\oplus(r_1, r_2) \rightarrow r_3$ where all operands belong to any classification. The result is the more conflicting classification e.g. $Vectorizable \oplus ScalarConflict \rightarrow ScalarConflict$. If the result belongs to the set *Maybe* it can be either *ScalarConflict* or *VectorConflict*. We decided to maintain both types instead of merging them into one, in order to the provide more detailed information to the designer.

The core of the algorithm is the if-clause in Line 32 where we analyze if a loop is applicable for vectorization. Specifically, three conditions must be satisfied. First, the loop iterations must be countable. The function IsCanonical checks if this is the case. Second, all nested statements must be supported. The recursive function call in Line 36 analyzes all nested statements. Third, we have to check if all expressions are conflict-free. The function ConflictCheck analyzes if all write operations target local variables or vectorizable arrays. A loop is vectorizable under two circumstances: a) the second and the third check return *Vectorizable*. b) the second check may result in *ScalarConflict* or *VectorConflict*, but the result for the third is *Vectorizable*. This describes the situation where the inner loop is not vectorizable but the outer is. Otherwise, we distinguish if the result belongs to the class *Maybe* or *NotVectorizable*.

2.3 Tool Flow

First, our SystemC compiler reads the design `design.cpp` and translates it into a thread-safe design `risc_design.cpp` to achieve thread-level parallelism. Additionally, our compiler provides to the designer information with candidates for loop vectorization to exploit data-level parallelism in the terminal. After selecting from the provided source code locations, the user inserts the statement `#pragma simd` in front of the chosen loops. Finally, the design `risc_design.cpp` is compiled with the Intel *icpc*.

The feedback of the designer is needed. An example is the following C function: `void add(float *a,float *b,float *c,int n){for(int i=0;i<n;i++){a[i]=a[i]+b[i]+c[i];}}`. Here, arrays passed as pointers can only be vectorized if the user asserts that there is no vector dependence in the way. This is only possible with application knowledge, not by static compiler analysis. Our proposed compiler, which is aware of SystemC and its concurrent multi-threading semantics, can identify this loop as a potential candidate, but the final data independence assertion must come from the user who knows the application specifics (i.e. the pointers point to non-overlapping arrays).

3. EXPERIMENTS AND RESULTS

We have implemented the approach outlined above and demonstrate the combined speedup of *thread* and *data-level parallelism* on two different applications, namely a particle simulator on a NoC and a video Mandelbrot renderer unit. For both applications, we measure the sequential, the sequential with SIMD, the parallel, and the parallel with SIMD simulator run times. On one hand, the experiments execute on an Intel Xeon E3-1240 multi-core processor with 4 CPU cores. Each core has one simulation thread with a vectorization unit of 256 bit width. On the other hand, we use an Intel Xeon Phi™ Coprocessor 5110P many-core architecture. The coprocessor contains 60 cores where each core has a vectorization unit of 512 bit. To obtain unambiguous measurements, we turn CPU frequency scaling off for all experiments.

3.1 Network-on-Chip Particle Simulator

As a comprehensive example we select a Network-on-Chip (NoC) particle simulator model in SystemC to demonstrate the parallel simulation capabilities of our *thread* and *data-level parallelism* analysis. The abstract architecture of the particle simulator is illustrated in Figure 1. The grid is assembled of tiles where each tile communicates bidirectionally with a tile to its north, south, east, or west. For a 4x4 example, we have 16 tiles and one grid module. Each tile has one thread which computes the motion of the individual particles in a certain area of the model. The particles move continuously in 2D space. The moment when a particle crosses the boundary, the responsibility of computing and updating the position of the particles shifts from one tile to its neighbor tile. The entire design can be scaled up to any quadratic size. The user can configure via the command line the number of tiles as well as the number of particles, the gravity, and other options.

At the beginning of the simulation, the testbench sends the initial particles to each individual tile. This happens in a purely sequential fashion. Next, all tiles simulate the particles and then synchronize with their neighbor tiles. A tile is blocked when it communicates with one of its neighbors. At the end of the simulation, all the tiles send the particle positions back to the testbench.

First, our compiler performs a thread-level analysis of the design. Each thread is identified and analyzed for race conditions with all other threads in the design. The resulting conflict table is then passed to our parallel SystemC library which performs out-of-order scheduling.

Next, our infrastructure creates and inspects the call graph of all possible threads for vectorization. The *data-level parallelism* analysis generates a list of potential locations for the vectorization. After inspection of the list, the function `apply_force()` is selected for parallelization. This function computes the gravity influence of the individual particles on each other. The Intel *icpc* compiler cannot directly identify this function for vectorization. Our analysis builds a call graph for each thread and filters it for possible candidate locations. Suitable candidates are vetted by the designer using his application knowledge. To confirm vectorization, the designer adds `#pragma simd` before the for-loop in question.

3.1.1 Experiment on the Multi-Core Host

Figure 4 shows the speedup for the particle simulator on the four core machine. First, the blue line (diamonds) shows the speedup M for the sequential SIMD simulation between 1.6x and 2.1x. This confirms our SIMD discussion above where the maximal speedup of 4x cannot be reached due to the needed overhead of packing and unpacking the lanes. The increasing communication of particles among tiles results in lower parallelism (Amdahl's law) which explains the

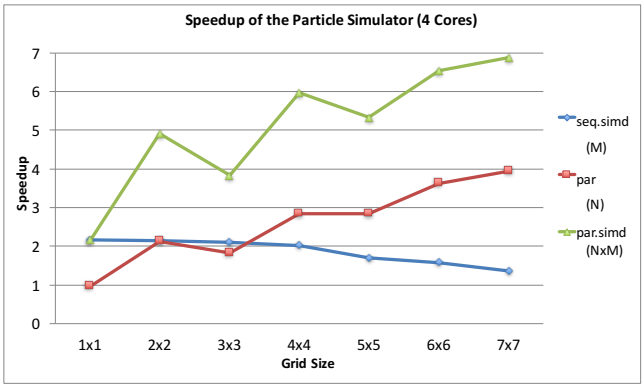

Figure 4: Speedup of the NoC Particle Simulator

decreasing speedup. For a 7x7 grid size, 49 threads are active and communicating synchronously bidirectional to the associated neighbor threads. So, intensive core to core communication limits the parallelism.

The red line (squares) shows the thread-level speedup N, generally increasing with higher grid sizes. The measurement and the resulting total speedup ($N \times M$) shown in green (triangles) show a zig-zag pattern. Grid sizes with an even number of rows and columns perform better than the grid sizes with odd number of rows and columns. This phenomenon is due to the implementation of the particle simulator, in particular due to its communication characteristics. Figure 5 compares the communication pattern of a 3x3 and 4x4 grid.

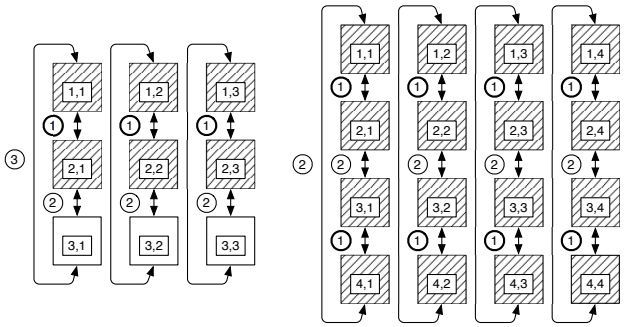

Figure 5: Communication pattern of the Particle Simulator

First, for the 4x4 grid the odd rows are communicating with the neighbors to the south (indicated as ①). In this case, eight communication pairs are active at the same time. Next, the even rows communicate to their south neighbors. Here, the last row communicates with the first row. Again, eight pairs are available for the parallel simulation (indicated as ②). So, at any point all four cores are fully utilized. The same applies for the horizontal communication.

For the 3x3 grid, the simulation has different characteristics. First the odd rows start communicating which is only the first row ①. Three pairs are available for execution which means that only three out of four cores are used. Next, the odd rows communicate ②. Again, only three pairs are available, one core stays idle. Finally, the last row communicates with the first row ③. In all phases, only three out of four cores are utilized. All over, the 4x4 grid can gain higher speed up due to the higher core utilization.

This explanation applies for the other gird sizes as well. For instance, the 6x6 grid size with 36 threads can be better utilized than a 5x5 with 25 threads on a 4 core machine.

Finally, the green (triangle) line shows the combination of the parallel and SIMD technique essentially the product of $N \times M$. The product of the sequential with SIMD and parallel simulation show the combined speedup. The maximal speedup is 6.8x which is impressive on 4 cores.

3.1.2 *Experiment on the Many-Core Host*

We simulated the particle simulator on the many-core architecture as well. The Intel Xeon Phi 5110P Coprocessor hardware has a ring architecture of cores. Two cores are communicating over a third core which hosts as a so-called tag directory. This communication scheme causes a high traffic congestion. In our simulations, the speedup is marginal (below 5x) and constantly decreases with the increasing number of threads. Similar results and the importance of a sophisticated thread to core mapping have been shown in [15] for this specific architecture. So, we decide not to investigate the NoC benchmark further on this platform.

3.2 Mandelbrot Renderer

The Mandelbrot renderer is a parallel video application to compute the Mandelbrot set [16]. Basically, the Device under Test (DUT) hosts a number of renderer units. Each unit computes a different slice of the Mandelbrot image. At compile time, the user defines how many slices are available. During the simulation, the DUT provides coordinates to the individual slices. A slice computes the Mandelbrot set for the given coordinates and sends the results back. The DUT receives the data from each unit and stores them. Finally, new coordinates are provided to all slices for the next frame. In contrast to the NoC particle simulator where tiles are intensively communicating, the individual renderer units are fully independent.

Our compiler automatically applies the *thread-level parallelism* to the individual threads of the renderer units. The *data-level parallelism* analysis identifies the central loop in the function `mandel_row()` as a candidate for vectorization, which we confirm.

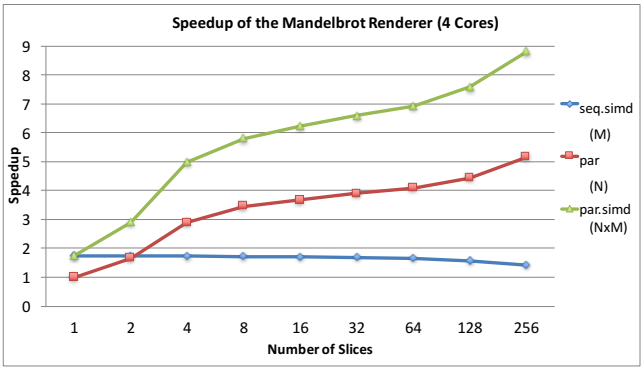

Figure 6: Speedup of the Mandelbrot Renderer

3.2.1 *Experiment on the Multi-Core Host*

Figure 6 shows the simulation speedup for the Mandelbrot renderer with up to 256 different units on the 4 core machine. First, the sequential simulation with SIMD support (blue diamonds) achieves a speedup M of about 1.7x.

Table 1: Simulation speedup for the Mandelbrot renderer on the multi- and many-core host architectures.

Slices	4 Core Machine											60 Core Machine with 4 Hyperthreads Each						
	Execution Time / CPU Utilization								Speedup			Execution Time				Speedup		
	seq		seq.simd		par		par.simd		seq.simd	par	par.simd	seq	seq.simd	par	par.simd	seq.simd	par	par.simd
1	41.53	100%	23.80	99%	41.74	99%	23.88	99%	1.74	0.99	1.74	393.76	56.94	393.52	56.70	6.92	1.00	6.94
2	41.66	99%	23.99	99%	25.14	165%	14.32	166%	1.74	1.66	2.91	393.75	56.92	234.15	33.44	6.92	1.68	11.77
4	41.91	99%	24.22	99%	14.53	297%	8.39	294%	1.73	2.88	5.00	393.79	56.92	129.45	18.58	6.92	3.04	21.19
8	42.36	99%	24.61	99%	12.28	354%	7.31	341%	1.72	3.45	5.79	393.75	56.92	67.36	9.82	6.92	5.85	40.10
16	43.07	99%	25.23	99%	11.74	372%	6.90	361%	1.71	3.67	6.24	393.77	56.93	34.62	5.43	6.92	11.37	72.52
32	44.58	99%	26.51	99%	11.44	381%	6.76	369%	1.68	3.90	6.59	393.80	56.97	18.47	2.87	6.91	21.32	137.21
64	46.69	99%	28.05	99%	11.41	382%	6.73	371%	1.66	4.09	6.94	393.90	57.12	9.59	1.89	6.90	41.07	208.41
128	50.85	99%	32.42	99%	11.46	381%	6.70	373%	1.57	4.44	7.59	393.97	57.14	8.51	1.85	6.89	46.29	212.96
256	58.99	99%	41.37	99%	11.43	382%	6.69	374%	1.43	5.16	8.82	394.20	57.35	7.90	2.03	6.87	49.90	194.19

The increasing number of slices slightly affect the speedup. Through the increasing number of threads in combination with the data need of the vectorization units, higher memory traffic occurs. Next, the *thread-level parallelism* (red squares) provides a speedup N of nearly 5.1x. This super-linear speedup is possible due to the poor cache utilization of sequentially scheduling 256 threads on the 4-core machine. Table 1 shows the increase from 41.5 seconds to 59 seconds for this sequential reference case. Finally, the combination of the *thread* and *data-level parallelism* $N \times M$ reach a total of up to 8.8x.

3.2.2 *Experiment on the Many-Core Host*

Finally, we simulate the Mandelbrot renderer on the Intel Xeon Phi many-core architecture. Figure 7 shows the simulation results. Due to the minimal communication needs compared to the particle simulator, highest speedups are reached. The vectorization unit with 512 bit can execute up to eight double-precision floating-point operations in parallel. A speedup M of 6.9x is achieved. The thread-level parallelization increases strongly on the 60 cores with a speedup N of 50x. Afterwards, the speed slows down. Due to the 60 physical cores and use of hyper threads. Finally, the combination of the thread and data level parallelization $N \times M$ generates a speedup of up to 212x.

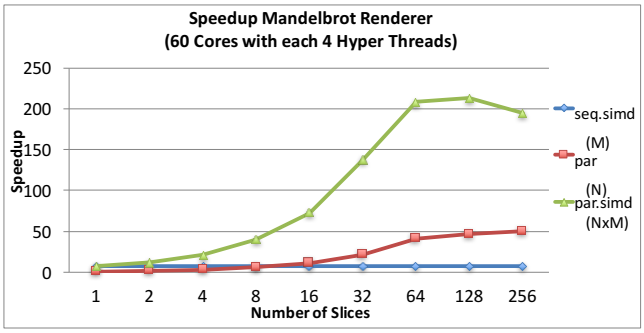

Figure 7: Speedup of the Mandelbrot Renderer

4. CONCLUSION AND FUTURE WORK

In this paper, we present our compiler infrastructure for the automatic parallelization of SystemC models at the *thread level*. Additionally, our infrastructure performs an analysis to identify source locations that are suitable for *data-level parallelization*. The vectorization analysis depends strongly on our SystemC aware compiler which identifies the simulation threads of the individual models and channels. To the best of our knowledge, this work is the first to apply and exploit SIMD vectorization on top of *thread-level* parallel SystemC simulation. We demonstrated our techniques on NoC particle simulator and Mandelbrot renderer SystemC benchmarks on a multi- and many-core architecture. On the 4 core machine, we achieved a speedup of up to 8.8x through the combination of *thread* and *data-level parallelism*. On the 60 core architecture, we gained up to 212x of simulation speedup. In future work, we plan to investigate both techniques on less than highly-parallel benchmarks and other SystemC modules.

ACKNOWLEDGMENT

This work has been supported in part by substantial funding from Intel Corporation for the project titled "Out-of-Order Parallel Simulation of SystemC Virtual Platforms on Many-Core Architectures". The authors thank Intel Corporation for the valuable support.

5. REFERENCES

[1] "IEEE Standard SystemC Language Reference Manual, IEEE Std 1666-2011," 2011.

[2] G. Glaser, G. Nitschey, and E. Hennig, "Temporal Decoupling with Error-Bounded Predictive Quantum Control," in *Forum on Specification and Design Languages (FDL)*, 2015.

[3] N. Ventroux and T. Sassolas, "A New Parallel SystemC Kernel Leveraging Manycore Architectures," in *DATE*, 2016.

[4] J. H. Weinstock, R. Leupers, G. Ascheid, D. Petras, and A. Hoffmann, "SystemC-Link: Parallel SystemC Simulation using Time-Decoupled Segments," in *DATE*, 2016.

[5] G. Liu, T. Schmidt, and R. Dömer, "RISC Compiler and Simulator, Beta Release V0.3.0: Out-of-Order Parallel Simulatable SystemC Subset," Center for Embedded and Cyber-Physical Systems, Technical Report 16-06, 2016.

[6] W. Chen, X. Han, and R. Dömer, "Out-of-Order Parallel Simulation for ESL Design," in *DATE*, 2012.

[7] R. M. Fujimoto, "Parallel Discrete Event Simulation," *Commun. ACM*, vol. 33, no. 10, 1990.

[8] W. Chen, X. Han, and R. Dömer, "May-Happen-in-Parallel Analysis based on Segment Graphs for Safe ESL Models," in *DATE*, 2014.

[9] J. H. Weinstock, C. Schumacher, R. Leupers, G. Ascheid, and L. Tosoratto, "Time-Decoupled Parallel SystemC Simulation," in *DATE*, 2014.

[10] C. Roth, S. Reder, H. Bucher, O. Sander, and J. Becker, "Adaptive Algorithm and Tool Flow for Accelerating SystemC on Many-Core Architectures," in *Digital System Design (DSD), 17th Euromicro Conference*, 2014.

[11] K. M. Chandy and J. Misra, "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs," *IEEE Transactions on Software Engineering*, no. 5, pp. 440–452, Sept 1979.

[12] R. Sinha, A. Prakash, and H. D. Patel, "Parallel Simulation of Mixed-abstraction SystemC Models on GPUs and Multicore CPUs," in *ASPDAC*, 2012.

[13] N. Bombieri, F. Fummi, and S. Vinco, "On the Automatic Generation of GPU-oriented Software Applications from RTL IPs," in *CODES+ISSS*, 2013.

[14] "Intel Corporation - Requirements for Vectorizable Loops," https://software.intel.com/en-us/articles/requirements-for-vectorizable-loops, accessed: 2016-08-09.

[15] G. Liu, T. Schmidt, R. Dömer, A. Dingankar, and D. Kirkpatrick, "Optimizing Thread-to-Core Mapping on Manycore Platforms with Distributed Tag Directories," in *ASPDAC*, 2015.

[16] B. Mandelbrot, "Fractal aspects of the iteration of $z \to \lambda z(1-z)$ for complex λ and z," Annuals of the New York Academy of Sciences, 1980.