DEVS-based parallel framework for Multi-Objective Evolutionary Algorithms

Alejandro Moreno
Dpto. de Informática y Automática
UNED
Madrid, Spain
amoreno@bec.uned.es

Luís de la Torre
Dpto. de Informática y Automática
UNED
Madrid, Spain
ldltorre@bec.uned.es

José L. Risco-Martin
Dpto. de Arquitectura de Computadores y Automática
UCM
Madrid, Spain
jlrisco@dacya.ucm.es

Eva Besada-Portas
Dpto. de Arquitectura de Computadores y Automática
UCM
Madrid, Spain
evabes@dacya.ucm.es

Joaquín Aranda
Dpto. de Informática y Automática
UNED
Madrid, Spain
jaranda@dia.uned.es

José L. Ayala
Dpto. de Arquitectura de Computadores y Automática
UCM
Madrid, Spain
jayala@fdi.ucm.es

ABSTRACT

Discrete Event Specification (DEVS) is a sound formal modeling and simulation framework based on concepts derived from dynamic systems theory. DEVS provides a framework for information modeling with several advantages to analyze and design complex systems: completeness, verifiability, extensibility, and maintainability. Multi-Objective Evolutionary Algorithms (MOEAs) are stochastic optimization heuristics where the exploration of the solution space of a certain problem is carried out by imitating the population genetics stated in Darwin's theory of evolution. MOEAs have been successfully applied to many NP-hard combinatorial optimization problems. Optimization of these problems by MOEAs often demand a high computational cost. MOEA generally must iterate a substantial number of times to find a valid and good enough solution to a given problem. The research work presented in this document describes a generic DEVS framework that is able to automatically parallelize a MOEA with multithreading. This framework has been applied to the 3D thermal aware floorplanning problem, reaching excellent results. The parallel implementation of MOEA using DEVS multithread environment has allowed us to obtain optimized configurations with a speed-up of 4.51.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Theory

Keywords

DEVS, MOEA, multithread, parallel, floorplanning

1. INTRODUCTION

Discrete Event Specification (DEVS) is a sound formal modeling and simulation framework based on concepts derived from dynamic systems theory. DEVS provides a framework for information modeling with several advantages to analyze and design complex systems: completeness, verifiability, extensibility, and maintainability. The research work proposes a parallel simulation framework built upon Discrete EVent System Specification (DEVS) Modeling and Simulation (M & S) formalism and Multithreading paradigm. Speaking of a parallelization framework in terms of the capacity of running multiple simulation modeling functions in parallel. Accelerating the overall simulation execution versus serial lay out models or non-parallel simulation frameworks. In multiprocessing systems (such as multi-core systems that include multiple complete processing units) multithreading aims to increase utilization of a single core by using thread-level as well as instruction-level parallelism. Hence, as described in section 7, simulation performance may increase in multi-core machines according to the modeling scheme and the number of processing units available.

Multi-Objective Evolutionary Algorithms (MOEAs) are stochastic optimization heuristics where the exploration of the solution space of a certain problem is carried out by imitating the population genetics stated in Darwin's theory of evolution. Selection, crossover and mutation operators, derived directly from natural evolution mechanisms, are applied to a population of solutions, thus favoring the birth and survival of the best solutions. MOEAs have been successfully applied to many NP-hard combinatorial optimization problems and work by encoding potential solutions (individuals) to a problem by strings (chromosomes), and by combining their codes and, hence, their properties. In order to apply MOEAs to a problem, a genetic representation of each individual has first

to be found. Furthermore, an initial population has to be created, as well as defining a cost function to measure the fitness of each solution. As a second step, we need to design the genetic operators that will allow us to produce a new population of solutions from a previous one, by capturing the interdependencies of the different topological constraints working concurrently. Then, by iteratively applying the genetic operators to the current population, the fitness of the best individuals in the population converges to targeted solutions, according to the metric/s to be optimized and the weight of each of these metrics. For an overview of MOEAs the reader is referred to [4].

The optimization of NP-hard combinatorial optimization problems carried out by MOEAs may require elevated computational cost. MOEA generally must iterate a substantial number of times to find a valid and good enough solution to a given problem. Each iteration consists of a series of functions or operators: selection, crossover, mutation, evaluation, and replacement. In some cases, the evaluation takes most computational cost and as well as crossover, mutation and sometimes selection can be executed in parallel since their performance is independent to the same operator applied over the current population. But basically, the importance relies on distributing the evaluation of the population in several processing units causing an acceleration in the algorithm performance. The evaluation operator of the case study presented in section 6 in a non-paralellized environment takes up to 99.5% of processing time.

The research work presented in this document describes a generic DEVS simulation framework modeled within DEVS formalism and parallelized with multithreading. This framework is able to parallelize MOEAs. A configurable platform that enables the end-user to define each genetic operator, solution variables type (a solution refers to an individual and a variables to chromosomes), population size, number of generations and specify the number of DEVS atomic modules that will work in parallel. The parallel implementation of the MOEA using DEVS multithread environment has allowed us to obtain optimized configurations with a speed-up of 4.51.

This paper is organized as follows. Section 2 collects some relevant aspects of DEVS. Section 3 analyzes the structure of coupled multi-thread. Section 4 presents each DEVS/-MOEA model types, whereas section 5 illustrates the overall simulation workflow. Then section 6 presents the experiments and results of a floor planner problem parallelized through DEVS/MOEA. Finally, in section 8 some conclusions are drawn.

2. DEVS

The Discrete Event System Specification is a general formalism for discrete event system modeling based on set theory [15]. It allows representing any system by three sets and five functions: input set (X), output set (Y), state set (S), external transition function (δ_{ext}), internal transition function (δ_{int}), confluent function (δ_{con}), output function (λ), and time advanced function (ta). DEVS provides a framework for information modeling with several advantages to analyze and design complex systems: completeness, verifiability, extensibility, and maintainability. DEVS can also approximate continuous systems using numerical integration methods. Thus, simulation tools based on DEVS are potentially more general than others including continuous simulation tools [9]. DEVS defines system behavior as well as system structure. System behavior in DEVS is described using input and output events as well as states. To this end, DEVS has two kinds of models to represent systems: atomic model and coupled model. The atomic model is the irreducible model definition that specify the behavior for any modeled entity. The coupled model is the aggregation/composition of two or more atomic and coupled models connected by explicit couplings between ports. An atomic model is defined by the following equation:

$$M = \langle X, S, Y, \delta_{int}, \delta_{ext}, \delta_{con}, \lambda \rangle \quad (1)$$

where,

- X is the set of input values
- S is the state space
- Y is the set of output values
- $\delta_{int} : S \to S$ is the internal transition function
- $\delta_{ext} : Q \times X^b \to S$ is the external transition function
  - $Q = \{(s, e) : s \in S, 0 \leq e \leq ta(s)\}$ is the total state set, where e is the time elapsed since last transition
  - X^b is a set of bags over elements in X
- δ_{con} is the confluent transition function, subject to $\delta_{con}(s, \emptyset) = \delta_{int}(s)$ [15]
- $\lambda : S \to Y$ is the output function
- $ta(s) : S \to \Re_0^+ \cup \infty$ is the time advance function.

The formal definition of a coupled model is described as:

$$N = \langle X, Y, D, EIC, EOC, IC \rangle \quad (2)$$

where,

- X is the set of external input events
- Y is the set of output events
- D is a set of DEVS component models
- EIC is the external input coupling relation
- EOC is the external output coupling relation
- IC is the internal coupling relation.

The coupled model N can itself be a part of component in a larger coupled model system giving rise to a hierarchical DEVS model construction.

DEVS models can be simulated with a simple ad-hoc program written in any language. In fact, the simulation of a DEVS model is not much more complicated than the simulation of a Discrete Time Model [12]. The problem arises with models composed by many subsystems where ad-hoc programming becomes very hard. One of the simplest ways to implement these complex models is writing a program with a hierarchical structure equivalent to the hierarchical structure of the model to be simulated. This is the method used in [15], where a class called *Simulator* is associated to each atomic DEVS model and a different class called *Coordinator* is related to each coupled DEVS model. At the top of the hierarchy there is a Coordinator, usually called the *Root Coordinator*, that manages the global simulation time.

3. MULTITHREAD DEVS MODEL

This section analyzes the structure of the upper level coupled DEVS model. As seen on figure 1, this model is composed by two types of components, one A type model and 1 to N B type models. Both are composed of one input port 'in' and one output port 'out'. Ports are typified with a solution variables type (T), and are capable to admit an undetermined number of solutions. Couplings are set in such a way that the output port of the model A is linked with all B models, and all B models outputs are coupled with the A model input. Type A model is a unique model in charge of initializing the algorithm, submitting the current population of size P to all B models and collecting the outputted population from each B model. Type B model receives the current population, processes and sends the new population of size $P/(N+1)$ (Adding model A to the division) to the sole A type model.

In order to automatically generate the coupled model lay out, the end-user must define the number of B models, solution variables type *(T)*, population size *(P)*, and genetic operators. The population size needs to meet a slight restriction, due to MOEA constraints and to favor parallelization. Crossover operator needs a pair number of parents, then the resultant of *P/(N+1)* must be an even number, and to avoid penalizing the parallelization, the module of the former division should be 0, so all models work with the same number of individuals. In figure 1 models are symbolized by squared boxes, ports by small appendixes attached to models, and bonds betwen ports as undirected edges.

4. DEVS/MOEA MODEL

Now, both aforementioned models behaviors are described in detail in the succeeding subsections. These models are defined within DEVS formalism, therefore, both models are decomposed through their DEVS modeling sets and functions. Listings 1 and 2 illustrate A and B type DEVS models expressed in pseudo codes not too far from the real ones in c++. These models may be enunciated in a more formal manner as in section 2, but in this case the reader will find the following pseudo code easier to deal with.

4.1 Model A

The A type model is assembled by input and output sets described in section 3, state set, and functions. As depicted in Listings 1, modelA keeps 9 state variables:

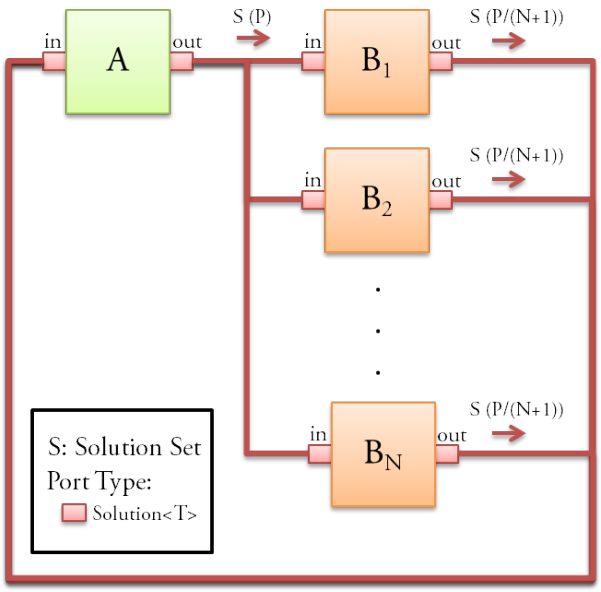

Figure 1: Multithread DEVS coupled model

- *sigma*: time for next internal event.
- *modules*: number of modules that divides the coupled model *(N+1)*.
- *sol*: vector of solutions of size P.
- *sol_state*: vector of solutions of size *P/(N+1)*.
- *selection, crossover, mutation, evaluation, replacement*: operators.

The internal transition function (δ_{int}) performs the hard work: selection, crossover, mutation and evaluation(99.5% processing time) over *P/(N+1)* individuals of population stored in variable 'sol'. The new generation is stored in variable 'sol_st'. Then sigma is set to ∞ and waits for inputs. On the other hand, when inputs are received, the external transition (δ_{ext}) executes the replacement operation with the past population stored in 'sol' together with the sum of input solutions from port 'in' and the solution set stored in 'sol_st'. Variable 'sol' is updated with the resultant of the reduction of both populations and variable 'sol_st' is initialized to an empty set. Sigma is set to 1, causing the execution at *current simulation time + 1* of the output function (λ), followed by δ_{int}. λ simply places the value of state variable 'sol' to the output port 'out'. Both the confluence function (δ_{con}) and the time advanced function (ta) have been omitted, since the former will never take place due to the configuration of the system, and the latter only returns the value of the state variable 'sigma' (This condition also comprehends model B).

Listing 1: A Type DEVS model

```
ModelA{
  Set input = {("in",vector<Solution<T>>)};
  Set output = {("out",vector<Solution<T>>)};
  Set state = {("sigma",int),("modules",int),
  ("sol",vector<Solution<T>>),
```

```
  ("sol_st",vector<Solution<T>>),
  ("sel",SelectionOp),("crs",CrossoverOp),
  ("mut",mutationOp),("eva",EvaluationOp),
  ("rep",ReplacementOp)};

  void delt_int(Set state){
    //extract values from state
    int modules = state.get("modules");
    vector<Solution<T>> sol = state.get("sol");
    vector<Solution<T>> sol_st = state.get("sol_st");
    SelectionOp sel = state.get("sel");
    CrossoverOp crs = state.get("crs");
    MutationOp mut = state.get("mut");
    EvaluationOp eva = state.get("eva");
    //selection,crossover,mutation and evaluation
    for(int i=0;i<((sol.size()/modules)/2);i++){
      Solution<T> par0=sel.run(sol);
      Solution<T> par1=sel.run(sol);
      Solution<T> offsprings[2]=crs.run(par0,par1);
      mut.run(offsprings[0]);
      mut.run(offsprings[1]);
      eva.run(offsprings[0]);
      eva.run(offsprings[1]);
      sol_state.add(offsprings[0]);
      sol_state.add(offsprings[1]);
    }
    //update sigma
    state.set("sigma",INFINITY);
  }

  void delt_ext(Set state,Set input,int elapsedTime){
    //extract values from state
    vector<Solution<T>> sol = state.get("sol");
    vector<Solution<T>> sol_st = state.get("sol_st");
    ReplacementOp rep = state.get("rep");
    //extract values from input
    vector<Solution<T>> input_sol = input.get("in");
    //replacement
    sol_st.add(input_sol);
    rep.run(sol,sol_st);
    //clear state solutions
    sol_st.clear();
    //update sigma
    state.set("sigma",1);
  }

  void lambda(Set state,Output output){
    //extract values from state
    vector<Solution<T>> sol = state.get("sol");
    //add solutions to output port
    output.add("out",sol);
  }
}
```

4.2 Model B

The B type model is assembled by input and output sets described in section 3, state set, and functions. As depicted in Listings 2, modelB keeps 7 state variables:

- *sigma*: time for next internal event.
- *modules*: number of modules that divides the coupled model *(N+1)*.
- *sol_state*: vector of solutions of size *P/(N+1)*.
- *selection, crossover, mutation, evaluation*: operators.

On this ocasion δ_{ext} performs the hard work: selection, crossover, mutation and evaluation over *P/(N+1)* individuals of population received at input port 'in'. The new generation is stored in variable 'sol_st'. Then sigma is set to 0 which implies the immediate execution of λ, followed by δ_{int}. λ just places the value of state variable 'sol_st' to the output port 'out'. Whereas δ_{int} sets sigma to ∞. The model is passivated and waits for inputs.

Listing 2: B Type DEVS model

```
ModelB{
  Set input = {("in",vector<Solution<T>>)};
  Set output = {("out",vector<Solution<T>>)};
  Set state = {("sigma",int),("modules",int),
  ("sol_st",vector<Solution<T>>),
  ("sel",SelectionOp),("crs",CrossoverOp),
  ("mut",mutationOp),("eva",EvaluationOp)};

  void delt_int(Set state){
    //extract values from state
    vector<Solution<T>> sol_st = state.get("sol_st");
    //clear state solutions
    sol_st.clear();
    //update sigma
    state.set("sigma",INFINITY);
  }

  void delt_ext(Set state,Set input,int elapsedTime){
    //extract values from state
    int modules = state.get("modules");
    vector<Solution<T>> sol_st = state.get("sol_st");
    SelectionOp sel = state.get("sel");
    CrossoverOp crs = state.get("crs");
    MutationOp mut = state.get("mut");
    EvaluationOp eva = state.get("eva");
    //extract values from input set
    vector<Solution<T>> sol = input.get("in");
    //selection,crossover,mutation and evaluation
    for(int i=0;i<((sol.size()/modules)/2);i++){
      Solution<T> par0=sel.run(sol);
      Solution<T> par1=sel.run(sol);
      Solution<T> offsprings[2]=crs.run(par0,par1);
      mut.run(offspring[0]);
      mut.run(offspring[1]);
      eva.run(offspring[0]);
      eva.run(offspring[1]);
      sol_state.add(offspring[0]);
      sol_state.add(offspring[1]);
    }
    //update sigma
    state.set("sigma",0);
  }

  void lambda(Set state,Output output){
    //extract values from state
    vector<Solution<T>> sol_st = state.get("sol_st");
    //add solutions to output port
    output.add("out",sol_st);
  }
}
```

5. DEVS/MOEA WORKFLOW

DEVS simulation workflow of the MOEA framework is presented in figure 3. DEVS functions (δ_{int}, δ_{ext}, λ) executions are symbolized by circular nodes. These nodes may belong to an A type model or to one of the B models. The information data flow is marked by directed arrows between nodes. As depicted on figure 3 legend, exchanged data among DEVS functions may be output messages or state sets. Output messages are exchanged between A and B models output and external transition functions. Whereas states are passed among transitions of the same model. Each column points out one or more DEVS function that may run in parallel within the same time band. A MOEA generation consist of four steps (columns) where the first and third step only transmit data. The first step spreads the entire population (P size vector of solutions) from model A to N B models, while the third sends populations (*P/(N+1)* size vector solutions) from N B models to model A. The second step takes

almost all of the computational load, performs the genetic operators selection, crossover, mutation and evaluation of P individuals. Finally, the fourth step carried out by model A, executes the sole genetic operator (replacement) not parallelized among different models. This interval causes a lost of performance in multi core and multithread systems since all processing units do not work at 100% and instruction level paralelism is not applied.

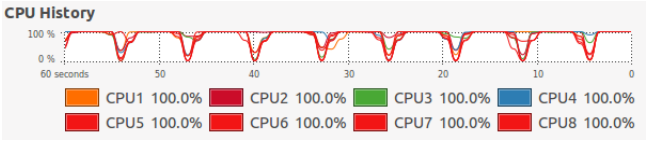

Figure 2: DEVS/MOEA simulation CPU monitor

Figure 2 shows the behavior of a multi core machine during the simulation of a DEVS/MOEA model. All CPU's are 100% busy most of the simulation except for the instant where the replacement is executed and output messages are sent from model A to N B models.

6. CASE STUDY

In order to validate our generic MOEA simulation framework modeled within DEVS formalism and parallelized with multithreading we analyze the optimization of a NP-hard case, the 3D thermal aware floorplanning problem.

Nowadays, many core system are being implemented and they present a unique opportunity to improve (sometimes in some orders of magnitude) the performance of the architecture. This is achieved increasing the high performance algorithms to specifically tailored architectures. Special HPC applications like N-Body Simulations, Molecular Dynamics, and Terrain Rendering [11] can experience order of magnitude or greater speedups when compared with architectures that are specifically tailored to their needs. Similar examples from the HPC community include the Los Alamos National Labs Roadrunner system.

This trend on executing the target applications in many parallel cores is not only one of the characteristics of the current data-centers, but also multi-processor systems-on-chip (MPSoCs) have reached the category of many-core systems. Intel Labs has created an experimental Single-chip Cloud Computer, (SCC) a research microprocessor containing the most Intel Architecture cores ever integrated on a silicon CPU chip with 48 cores [1]. It incorporates technologies intended to scale multi-core processors to 100 cores and beyond, such as an on-chip network, advanced power management technologies and support for message-passing. In this regard, three-dimensional (3D) multi-processor chips have been proposed as an effective mechanism to improve the performance of the system by reducing interconnect delays and increasing the density of the logic, making the idea of "many-core single-chip" into a reality. Since power densities are already a major concern in 2D architectures, the move to 3D architectures will accentuate the thermal problem. Consequently, it is mandatory to devise efficient 3D floor planning mechanisms that optimize the thermal profile of these complex 3D multi-processor architectures.

Most of the algorithms presented for the 3D thermal aware floor planning problem are based on a Mixed Integer Linear Program (MILP) [7, 10], Simulated Annealing (SA) [5, 7] or Genetic Algorithm (GA) [14]. MILP has proven to be an efficient solution. However, when MILP is used for thermal aware floor planning, the (linear) thermal model must be added to the topological relations and the resultant algorithm becomes too complex [6]. Regarding SA and GA, both are based on the representation of the solution. Some common representations are polish notation [3], combined bucket array [5] and O-tree [14]. Most of these representations do not perform well, because they were initially used to reduce area. In the thermal aware floor planning problem, hottest elements must be placed as far as possible in the 3D IC. We focus our attention on the approach given in [2]. Such approach reaches excellent results, but at high computational cost. Thus, we propose our framework as a mechanism to automatically parallelize the aforementioned MOEA.

Basically, this MOEA uses a cycle crossover. In the same way, mutation can be executed in two different ways, both with the same probability, where some blocks are chosen and swapped, and others are rotated 90 degrees. The hard task is done by the evaluation function, which performs a pseudo-exhaustive exploration to find the best placement of each individual and takes a 99.5% of the total execution time.

This MOEA is parallelized using DEVS/MOEA framework explained in the previous sections. Each solution (individual of the population) is defined by an array of variables of type component ($T = component$). These components may vary from cores, memories to crossbars. Selection operator uses binary tournament and the rest of operator have been already described. These operators meet the parallelization constraint so as to execute them within DEVS/MOEA, i.e., none of them are dependent of results from same operators applied over the current population.

In order to evaluate the parallelized floorplanner, we study an architecture inspired in the Niagara platform (see Figure 4). This architecture is composed of 48 SPARC cores, 72 memories and 6 crossbars used for inter-processor communication adding up a total of 126 components. In [13] we find that the power consumption of the SPARC is 4W at 1.4GHz. The power consumption values and areas of the memories are found with the CACTI software [8]. The floorplanner will place the processors, the local memories and the crossbars in 4 layers. This architecture represent the current state-of-the-art in 3D many core integration.

7. RESULTS

The subsequent sections analyze the speed-up obtained in the optimization of the Niagara 48 cores platform floor planning within DEVS/MOEA framework and the thermal maps for the simulation of one of the non-dominated optimized systems.

7.1 Speed-up

The results shown at table 1 denote the speed-up obtained with DEVS/MOEA applied over the Niagara floor planning problem. DEVS/MOEA Experiments were modeled vary-

ing from *N=1* to *N=9* B type models. Speed-up's were computed in relation with the execution time of a generic MOEA algorithm. Population size was set to *100* individuals and to *250* generations so as to compare fairly execution times among each configuration.

Alg.	*MOEA*	*N=1*	*N=2*	*N=3*	*N=4*	*N=5*	*N=6*	*N=7*	*N=8*	*N=9*
time(sec)	8053	3957	3224	2904	2526	2051	1916	1787	2062	1971
speed-up	1	2.04	2.50	2.78	3.19	3.93	4.20	4.51	3.91	4.09

Table 1: Execution time and speed-ups

The experiments were performed with $Intel®Core^{TM}$ $i7-2630QM$ processor with technology $Intel®HT$ that allows every kernel to work with 2 jobs at the same time. A processor composed of 4 cores (8 threads) running at 2.00GHz. Table 1 shows the obtained speed-ups and the execution time of the optimization of the Niagara architecture with a number B models moving from 1 to 9. As expected, due to the capacity of the mentioned processor, speed-up kept increasing until N reached 7 (see table 1). The explanation is naive, with $N = 7$, the overall DEVS/MOEA model is divided in 8 modules working in parallel which matches the number of maximum threads manageable in parallel by the aforementioned processor (*1 A model + 7 B models = 8 threads*). If N keeps increasing, as seen on figure 5, speed-up starts decreasing since CPU usage efficiency falls. The maximum speed-up reaches 4.51, which is an excellent number to consider the enhancement performance of multithread DEVS/-MOEA framework in multi-core and multi-thread systems.

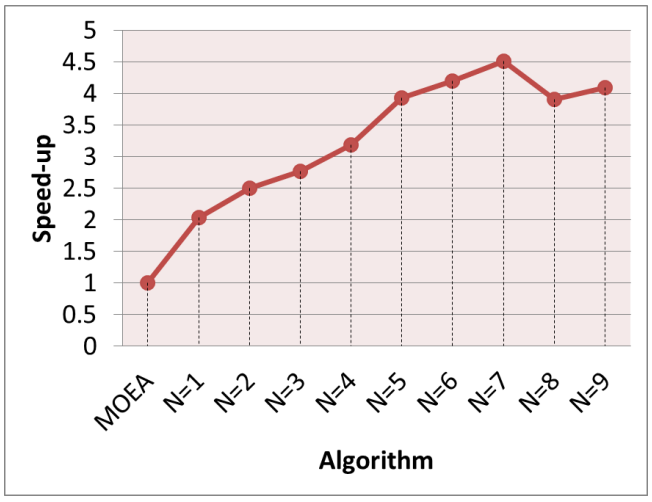

Figure 5: Speed-up analysis

7.2 Thermal analysis

Figure 6 shows the thermal maps for the simulation of one of the non-dominated optimized systems. Three thermal metrics commonly used have been calculated: the mean temperature, thermal gradient and maximum temperature for every layer of the configuration. The comparison with the baseline system (48 core original system depicted in Figure 4) shows that the floorplanner is capable of optimizing the maximum temperature in 62 degrees, the mean temperature in 6 and the thermal gradient is decreased in 72 in the best case (see Table 2). This can be explained because the floorplanner spaciates heat sources (cores) as much as possible, trying to place them at the border of the chip, helping on the cooling down of the cores. The floorplanner also takes into account vertical heat spread, and each layer will have a different layout, avoiding placing heat sources one over the other. As can be seen in the optimized floorplans the cores are mainly placed in the first and last layer, leaving inner layers with heat sinks. Using this approach heat is spread equally in all the chip achieving big reductions in mean and maximum temperature, as well as a decrease in the thermal gradient. It determines a more homogeneous thermal distribution, which is translated into a reduced reliability risk and diminished leakage currents. On the other hand, the wire length of the optimized configuration is a 3.11% greater than the original, which translates into a reduced loss in performance. However, in the result set of non-dominated solutions, we may find configurations in which the wire length is reduced a 4.64%, but with higer cost in temperature (the mean temperature is 2 degrees greater than in the original configuration, for example).

Original	Layer 1	Layer 2	Layer 3	Layer 4
T_{max}	411.82	410.21	407.06	402.34
T_{mean}	345.78	345.17	343.99	342.21
T_{grad}	109.75	108.18	105.07	100.42
Optimized	Layer 1	Layer 2	Layer 3	Layer 4
T_{max}	349.98	348.97	349.67	348.85
T_{mean}	339.70	339.00	338.31	337.63
T_{grad}	38.14	37.01	37.10	34.86

Table 2: 48-core thermal results

8. CONCLUSIONS

The research work presented in this document describes a generic DEVS framework that is able to automatically parallelize a MOEA with multithreading. MOEAs have been successfully applied to many NP-hard combinatorial optimization problems. Optimization of these problems by MOEAs may demand a high computational cost. Hence, DEVS/-MOEA framework has been design to automatically parallelize the optimization of a floorplanner problem. The parallel implementation of the MOEA using DEVS multithread environment has allowed us to obtain optimized configurations of platforms composed of 48 cores with a speed-up of 4.51, which is an excellent number to consider the enhancement performance of multithread DEVS/MOEA framework in multi-core and multi-thread systems.

9. REFERENCES

[1] SCC External Architecture Specification (EAS), 2011. Intel Labs.

[2] I. Arnaldo, J. L. Risco-MartÃn, J. L. Ayala, and J. I. Hidalgo. Power profiling-guided floorplanner for thermal optimization in 3d multiprocessor architectures. In *21st International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS 2011)*, 2011.

[3] J. Berntsson and M. Tang. A slicing structure representation for the multi-layer floorplan layout problem. In *EvoWorkshops*, pages 188–197, 2004.

[4] C. Coello-Coello, D. A. V. Veldhuizen, and G. B.

Lamont. *Evolutionary Algorithms for Solving Multi-Objective Problems*. Kluwer, 2002.

[5] J. Cong, J. Wei, and Y. Zhang. A thermal-driven floorplanning algorithm for 3d ics. In *Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design*, ICCAD '04, pages 306–313, Washington, DC, USA, 2004. IEEE Computer Society.

[6] M. Ekpanyapong, M. B. Healy, C. S. Ballapuram, S. K. Lim, and H. hsin S. Lee. Thermal-aware 3d microarchitectural floorplanning. Technical report, Georgia Institute of Technology Center for, 2004.

[7] M. Healy, M. Vittes, M. Ekpanyapong, C. S. Ballapuram, S. K. Lim, H.-H. S. Lee, and G. H. Loh. Multiobjective microarchitectural floorplanning for 2-d and 3-d ics. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 26(1):38–52, 2007.

[8] HPlabs. www.hpl.hp.com/research/cacti/.

[9] E. Kofman. Discrete Event Simulation of Hybrid Systems. *SIAM Journal on Scientific Computing*, 25(5):1771–1797, 2004.

[10] X. Li, Y. Ma, and X. Hong. A novel thermal optimization flow using incremental floorplanning for 3d ics. In *Proceedings of the 2009 Asia and South Pacific Design Automation Conference*, pages 347–352, Piscataway, NJ, USA, 2009. IEEE Press.

[11] V. T. B. Minor and G. Fossum. Terrain Rendering Engine (TRE). Technical report, IBM, 2005.

[12] J. Nutaro. On constructing optimistic simulation algorithms for the discrete event system specification. *ACM Trans. Model. Comput. Simul.*, 19(1):1–21, 2008.

[13] OpenSPARC. http://www.opensparc.net/pubs/preszo/07/n2isscc.pdf, 2007.

[14] M. Tang and X. Yao. A memetic algorithm for vlsi floorplanning. *IEEE Transactions on Systems, Man, and Cybernetics, Part B*, 37(1):62–69, 2007.

[15] B. P. Zeigler, T. Kim, and H. Praehofer. *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems*. Academic Press, 2000.

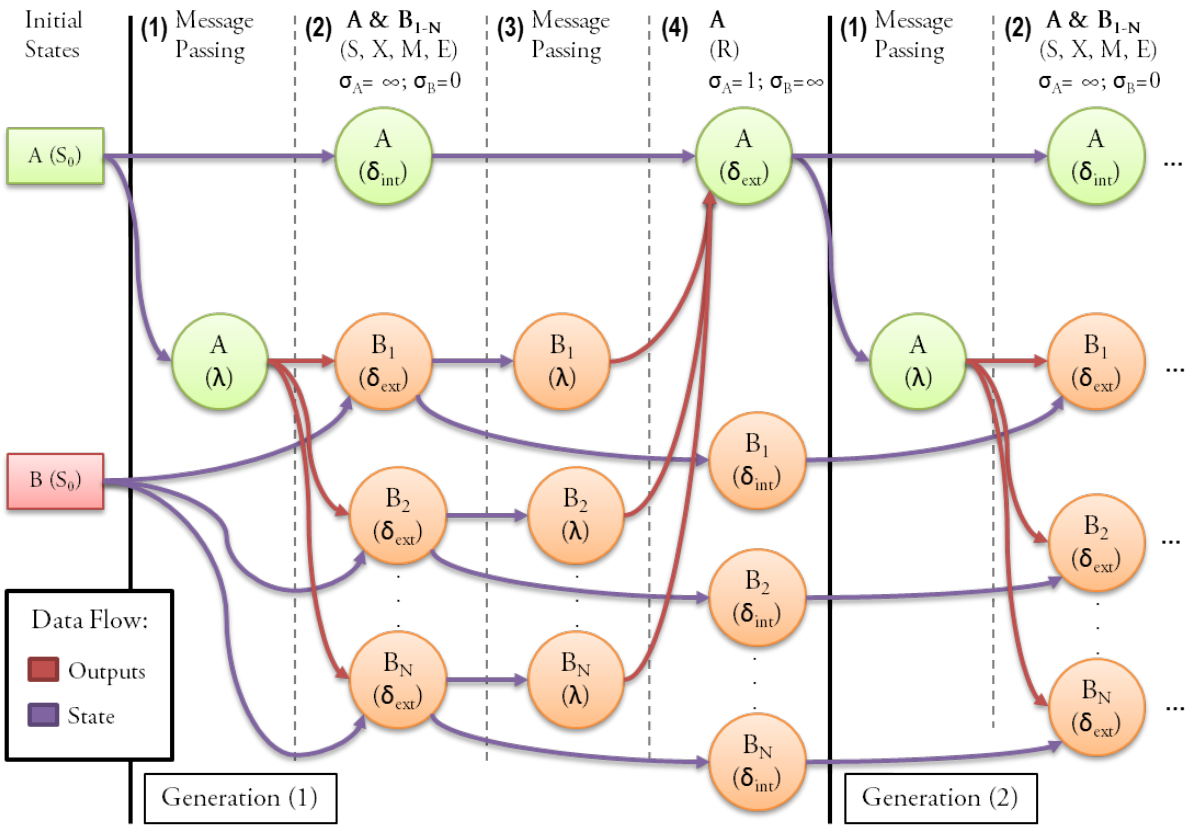

Figure 3: DEVS/MOEA Workflow

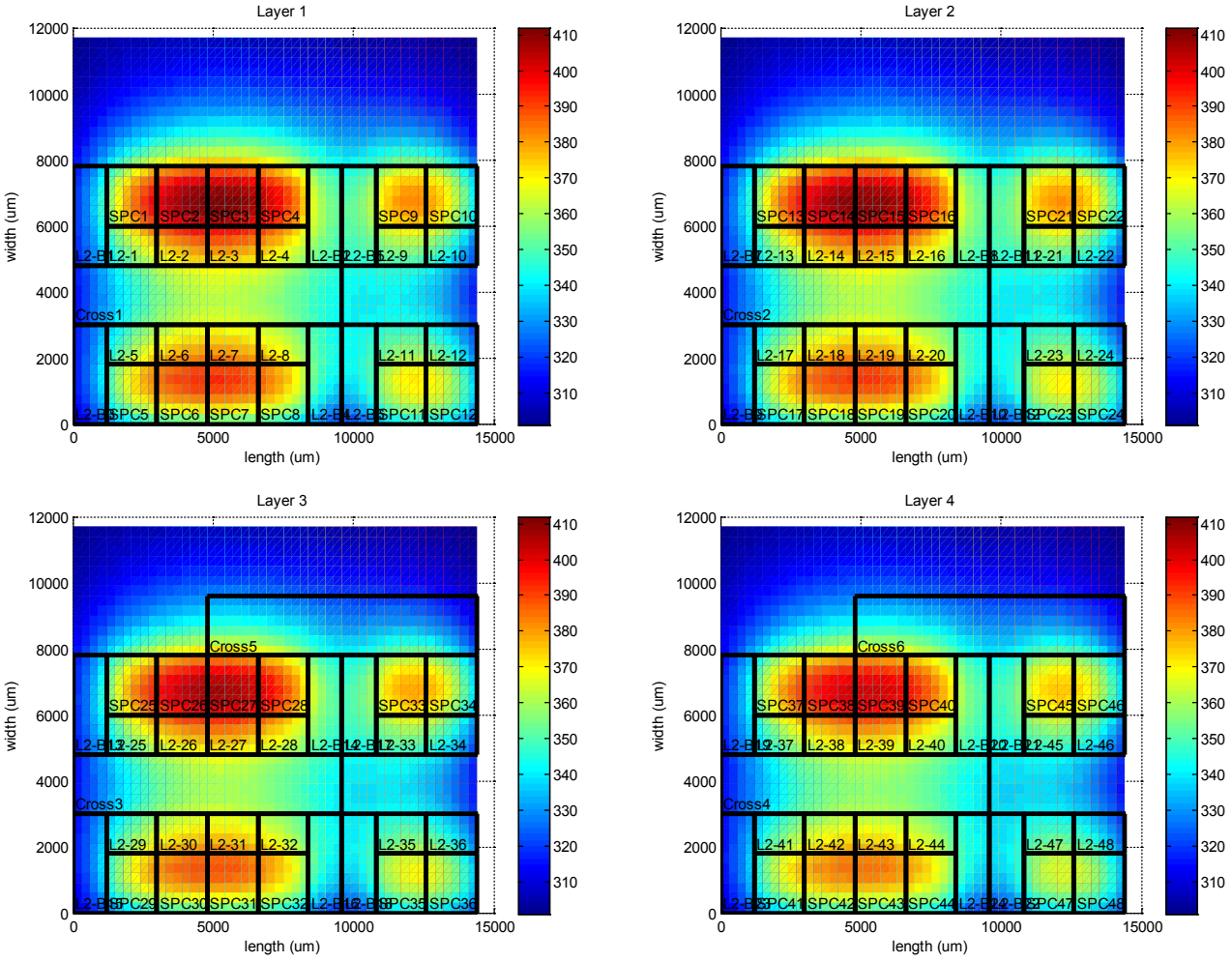

Figure 4: Thermal map of the 4 layers for the original configuration of the 48 cores platform

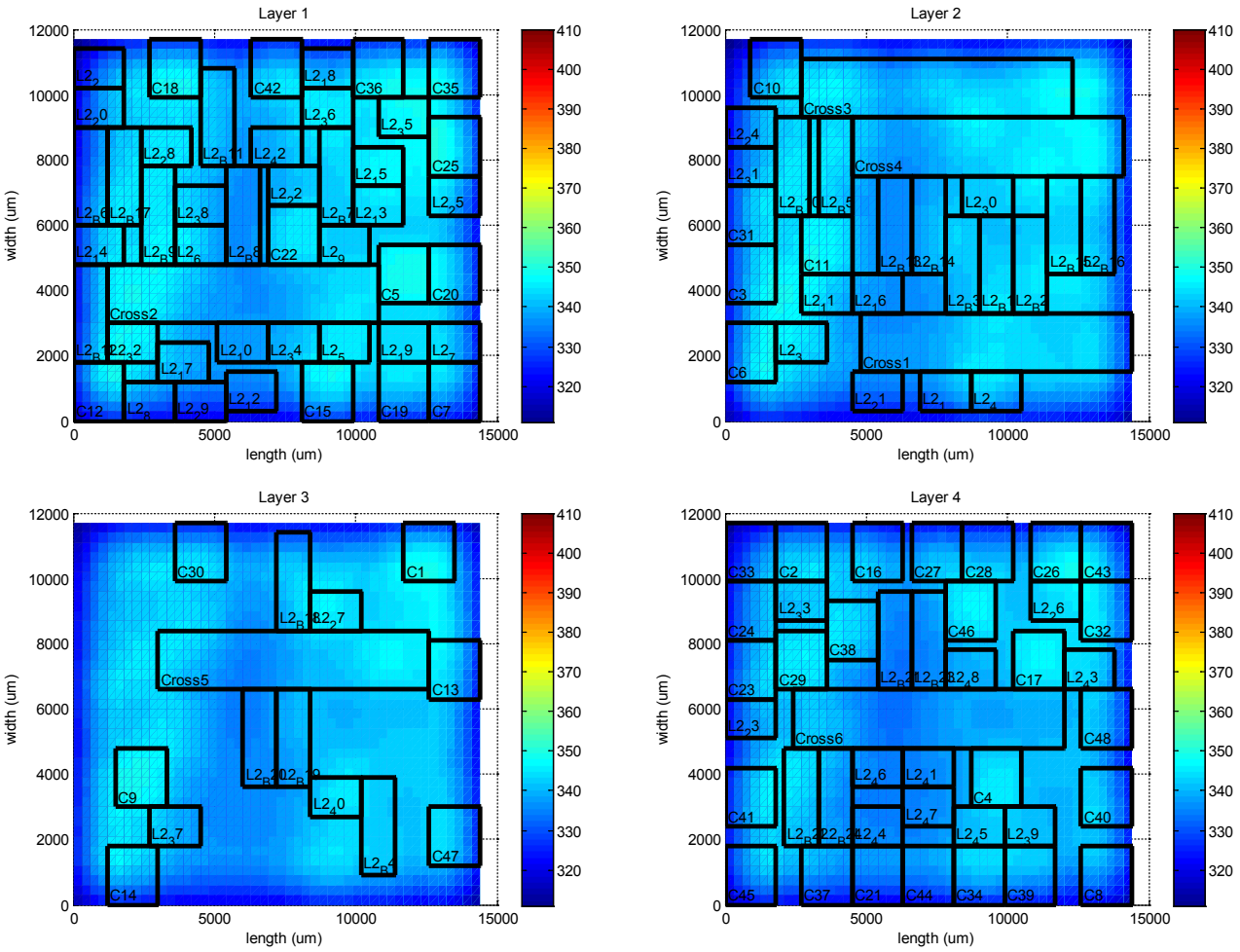

Figure 6: Thermal map of the 4 layers for the optimized configuration of the 48 cores platform