Particle Swarm Optimization of Memory usage in Embedded Systems

José L. Risco-Martín*
Department of Computer Architecture and Automation,
Complutense University of Madrid, Madrid, Spain
E-mail: jlrisco@dacya.ucm.es
*Corresponding author

Oscar Garnica
Department of Computer Architecture and Automation,
Complutense University of Madrid, Madrid, Spain
E-mail: ogarnica@dacya.ucm.es

Juan Lanchares
Department of Computer Architecture and Automation,
Complutense University of Madrid, Madrid, Spain
E-mail: julandan@dacya.ucm.es

David Atienza
Embedded Systems Laboratory (ESL)
Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland
E-mail: david.atienza@epfl.ch,
Department of Computer Architecture and Automation,
Complutense University of Madrid, Madrid, Spain
E-mail: datienza@dacya.ucm.es

J. Ignacio Hidalgo
Department of Computer Architecture and Automation,
Complutense University of Madrid, Madrid, Spain
E-mail: hidalgo@dacya.ucm.es

Abstract: In this paper, we propose a dynamic, non-dominated sorting, multi-objective particle-swarm-based optimizer, named Hierarchical Non-dominated Sorting Particle Swarm Optimizer (H-NSPSO), for memory usage optimization in embedded systems. It significantly reduces the computational complexity of others Multi-Objective Particle Swarm Optimization (MOPSO) algorithms. Concretely, it first uses a fast non-dominated sorting approach with $\mathbf{O(mN^2)}$ computational complexity. Second, it maintains an external archive to store a fixed number of non-dominated particles, which is used to drive the particle population towards the best non-dominated set over many iteration steps. Finally, the proposed algorithm separates particles into multi sub-swarms, building several tree networks as the neighborhood topology. H-NSPSO has been made adaptive in nature by allowing its vital parameters (inertia weight and learning factors) to change within iterations. The method is evaluated using two real world examples in embedded applications and compared with existing covering methods.

Keywords: Embedded Systems; Dynamic Memory Optimization; Particle Swarm Optimization; Multi-Objective Optimization; Evolutionary Computation.

Biographical notes: José L. Risco-Martín is is an Assistant Professor in Complutense University of Madrid, Spain. He received his Ph.D. from Complutense University of Madrid in 2004. His research interests are computational theory of modeling and simulation, with emphasis on DEVS, dynamic memory management of embedded systems, and net-centric computing.

1 INTRODUCTION

Optimizations with multiple objectives are needed in a great variety of real-life optimization problems. In these problems there are several conflicting objectives to be optimized and it is difficult to achieve the best solution. A multi-objective optimization problem is solved, when all its Pareto-optimal solutions are found. Indeed, the goal of multi-objective optimization is to find a set of optimal solutions in one simulation run, in contrast to classical optimization methods, which generally find one of the Pareto optimal solutions by converting the initial multi-objective optimization problem into a single-objective one. Unfortunately, it is impossible to find the whole set of Pareto-Optimal Solutions of a continuous front. Nevertheless, since the decision makers only require a restricted amount of well-distributed solutions along the *Pareto-Optimal Front (POF)*, the task of multi-objective optimization methods can be simplified to find a relatively small set of solutions.

In elitist *Multi-Objective Evolutionary Algorithm (MOEA)* and *Multi-Objective Particle Swarm Optimization (MOPSO)* methods, the elite solutions are transferred by an archive to the next generation, and the archive of the last generation is the output of the method. Consequently, restricting the size of the archive affects the diversity of solutions and the computational time. Therefore, most MOEA and MOPSO methods try to restrict the amount of solutions in the output, while keeping a good diversity along the POF.

Diversity of output solutions is studied by applying methods like niching, clustering or truncation by several researchers Deb (2001), Zitzler (1999). However, these techniques often need a high computational time and at last we only have a restricted number of solutions in the output Zitzler (1999). On the other hand, the multi-objective methods can reach a large set of non-dominated solutions, if they are executed for a large number of generations. Thus, although they also imply a high computational time, the decision maker has the flexibility to choose among several solutions from the whole POF.

In this paper, we address the problem of finding the POF by applying MOPSO. MOPSO methods have the property that the particles move towards the POF during generations. Consequently, by running a MOPSO with a restricted archive size, it is possible to find a well-distributed set of non-dominated solutions very close to the POF Mostaghim and Teich (2003). In this work, we exploit this knowledge to propose another MOPSO, called *Hierarchical Non-dominated Sorting Particle Swarm Optimizer (H-NSPSO)*, which covers the gaps between the non-dominated solutions. The particles in the population of the H-NSPSO are divided into sub-swarms after each generation by using the fast non-dominated sorting method in Deb et al. (2002), and subsequently these sub-swarms take the responsibility to recover the POF. To this aim, we maintain in H-NSPSO an external archive to store a fixed

number of non-dominated particles, the inertia weight and learning factors are modified between iterations, and a mutation operator is applied to each particle. This method is validated on different test functions and compared with other state-of-the-art approaches for a real world application in the domain of embedded systems design. Results show that H-NSPSO outperforms other evolutionary algorithms in terms of quality of the generated aproximation set, under the assumption that the hypervolume metric reflects the decision maker's preferences.

The remainder of the paper has the following structure. Definitions of PSO and a brief background are given in Section 2. In Section 3, the proposed H-NSPSO method is studied. Section 4 explains the experimental results when applying it to a real world problem of embedded systems design. In Section 5 we summarize the main conclusions of this work.

2 BACKGROUND

2.1 Multi-objective optimization

Multi-objective optimization aims at simultaneously optimizing several contradictory objectives. For such kind of problems, a single optimal solution does not exist, and compromises have to be made. Thus, without any loss of generality, we can assume the following formulation of the m-objective minimization problem:

$$
\begin{aligned}
Minimize \quad & \vec{z} = (f_1(\vec{x}), f_2(\vec{x}), \dots f_m(\vec{x})) \\
subject\ to \quad & \vec{x} \in X
\end{aligned}
\tag{1}
$$

where $\vec{x} = [x_1, x_2, \dots x_n]$ is the vector of decision variables, $f_i : \Re^n \to \Re$, $i = 1, 2, \dots, m$, are the objective functions and $X \subset \Re^n$ is the feasible region in the decision space. A solution $\vec{x} \in X$ is said to dominate another solution $\vec{y} \in X$ (denoted as $\vec{x} \prec \vec{y}$) if the following two conditions are satisfied:

$$
\begin{aligned}
& \forall i \in \{1, 2, \dots, m\}, f_i(\vec{x}) \leq f_i(\vec{y}) \\
& \exists i \in \{1, 2, \dots, m\}, f_i(\vec{x}) < f_i(\vec{y})
\end{aligned}
\tag{2}
$$

If there is no solution which dominates $\vec{x} \in X$, $\vec{x}$ is said to be a *Pareto Optimal Solution (POS)*. The set of all elements of the search space that are not dominated by any other element is called the *Pareto Optimal Front (POF)* of the multi-objective problem: it represents the best possible solution with respect to the contradictory objectives. A multi-objective optimization problem is solved, when its complete POS is found.

2.2 Particle swarm optimization

Particle Swarm Optimization (PSO) is a heuristic search technique that simulates the movements of a flock of birds that aim at finding food Eberhart and Shi (1998). The

relative simplicity of PSO and the fact that is a population-based technique have made it a natural candidate to be extended for multi-objective optimization.

Moore and Chapman proposed the first extension of the PSO strategy for solving multi-objective problems in an unpublished manuscript from 1991 Moore and Chapman (1999). There are currently over twenty five different proposals of multi-objective PSOs (or MOPSOs) reported in the literature Reyes-Sierra and Coello (2006).

In PSO, particles are "flown" through a hyper-dimensional search space. Changes to the position of the particles within the search space are based on the social-psychological tendency of individuals to emulate the success of other individuals.

The position of each particle is changed according to its own experience and its neighbors. Let $\vec{x}_i(t)$ denote the position of particle p_i, at time step t. The position of p_i is then changed by adding a velocity $\vec{v}_i(t)$ to the current position, i.e.:

$$\vec{x}_i(t) = \vec{x}_i(t-1) + \vec{v}_i(t) \qquad (3)$$

The velocity vector reflects the socially exchanged information and, in general, is defined in the following way:

$$\begin{aligned}
\vec{v}_i(t) &= W\vec{v}_i(t-1) \\
&+ C_1 r_1 \left(\vec{x}_{pbest} - \vec{x}_i(t-1)\right) \qquad (4) \\
&+ C_2 r_2 \left(\vec{x}_{leader} - \vec{x}_i(t-1)\right)
\end{aligned}$$

where:

- W is the inertia weight. It is employed to control the impact of the previous history of velocities.

- C_1 and C_2 are the learning factors. C_1 is the cognitive learning factor and represents the attraction that a particle has toward its own success. C_2 is the social learning factor and represents the attraction that a particle has toward the success of its neighbors.

- $r_1, r_1 \in [0,1]$ are random values.

- $\vec{x}_{pbest}$ is the personal best position of particle i, namely, the position of the particle that has provided the greatest success.

- $\vec{x}_{leader}$ is the position of the particle that is used to guide particle i towards better regions of the search space.

Particles tend to be influenced by the success of any other element they are connected to. These neighbors are not necessary particles close to each other in the decision variable space, but instead are particles that are close to each other based on a neighborhood topology, which defines the social structure of the swarm.

We can define for instance a fully-connected graph or star topology Engelbrecht (2002), which connects all the members of the swarm to one another. In this case, $\vec{x}_{leader}$
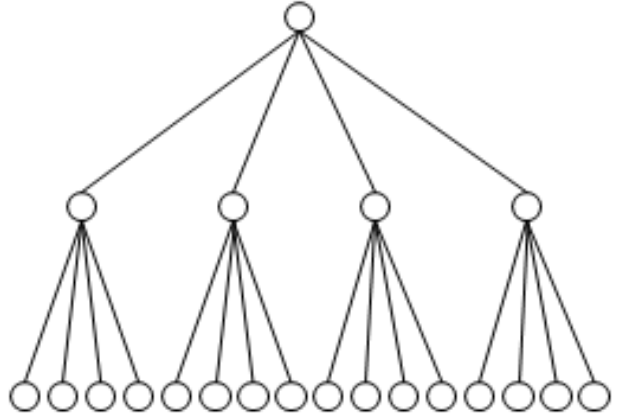

Figure 1: Tree network topology (each circle represents a particle). All particles are arranged in a tree, and it is influenced by its own best position so far and by the best position of its parent.

in equation (5) is defined by the position of the best particle of the entire swarm.

We can also define a tree network is shown in Figure 1. In this topology, all particles are arranged in a tree and each node of the tree contains exactly one particle Janson and Middendorf (2005), and $\vec{x}_{leader}$ is the first particle in the tree. A particle is influenced by its own best position so far ($\vec{x}_{pbest}$) and by the best position of the particle that is directly above in the tree (parent). If a particle has found a solution that is better than $\vec{x}_{pbest_{parent}}$ at the parent node, both particles are exchanged. As a result, this topology offers a dynamic neighborhood. This structure is also called hierarchical topology, where $\vec{x}_{leader} = \vec{x}_{pbest_{parent}}$.

We refer the interested reader to Kennedy (1999) for a complete survey of other neighborhood topologies.

3   HIERARCHICAL NON-DOMINATED SORTING PSO

Our hierarchical version of the Non-dominated PSO, i.e., H-NSPSO is introduced in this section.

H-NSPSO applies the main mechanisms of the NSGA-II Deb et al. (2002). Similarly, Xiaodong Li proposed a *Non-dominated PSO (NSPSO)* algorithm Li (2003). His approach is based on a fully- connected topology and incorporates the behavior of the NSGA-II to the PSO algorithm. In the NSPSO algorithm, once a particle has updated its position, instead of comparing the new position only against the $\vec{x}_{pbest}$ position of the particle, all the $\vec{x}_{pbest}$ positions of the swarm and all the new positions recently obtained are combined in just one set (given a total of $2N$ solutions, where N is the size of the swarm). Then, the approach selects the best solutions among them to define the next swarm (by means of a non-dominated sorting). This approach also selects the leaders randomly from the leaders set (stored in an external archive) among the best of them, based on two different mechanisms: a niche count
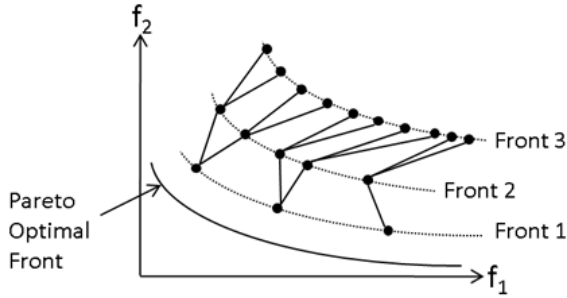
3

Figure 2: In this example particles of a swarm population are classified into 3 successive non-dominated fronts. Particles are arranged in several trees (subswarms).

and a nearest neighbor density estimator.

H-NSPSO introduces three modifications to NSPSO: (1) instead of a star topology, our algorithm makes use of a hierarchical topology dividing the population into subswarms, (2) we design a procedure to estimate the value of the inertia weight based on the crowding distance and learning factors, and (3) a mutation operator is applied.

3.1 Hierarchical topology

In H-NSPSO all particles are arranged in several tree networks that define the neighborhood structure. Each particle is neighbored to itself and the parent in the tree. We construct the trees by means of the fast non-dominated sorting algorithm proposed in Deb et al. (2002).

Figure 2 illustrates the process. First we sort the entire population according to the level of non-domination. To this end, we apply the fast non-dominated sorting algorithm proposed in Deb et al. (2002), obtaining three fronts in Figure 2. Front 1 is the best non-dominated set, since all particles in Front 1 are not dominated by any other particles in the entire population. Front 2 is the best non-dominated set when Front 1 is removed from the population, and so forth. Thus, the fast non-dominated sorting procedure applied to a population returns the list of non-dominated fronts. The overall complexity of this sorting algorithm is $O(mN^2)$, where m is the number of objectives and N is the size of the population. Next, we construct the trees according to the distribution of particles over the three fronts. Starting from Front 3 and Front 2, we assign to each particle in Front 3 a parent in Front 2. Then, the process is repeated for Fronts 2 and 1. If there exist just one front, then the NSPSO topology is applied as in Li (2003).

The design of multi-objective optimization algorithm not only requires good convergence quality, but also demands the appropriate distribution quality of the founded solutions in the whole objective space. Hence, we make use of a dynamic setting of inertia weight, learning factors and a mutation operator.

3.2 Dynamic setting of inertia weight

The inertia weight (W) value plays a crucial role in the convergence quality of particle swarm optimization algorithms. It controls the effect of the historic speed on the present one, and balances the use of the global research and the partial one. In particular, in H-NSPSO we make use of the crowding distance used in NSGA-II to calculate the inertia weight of each particle. Thus, the crowding distance serves in our case as an estimate of the size of the largest cuboid shape enclosing the particle i without including any other particle in the population.

Also, we allow the value of W to decrease from $W(0)$ to $W(T)$. The value of inertia weight at iteration t is obtained as:

$$W(t) = (W(T) - W(0)) \cdot \frac{t \cdot e^{-cd}}{T} + W(0) \cdot e^{-cd} \quad (5)$$

where T is the maximum number of iterations, t is the iteration number and cd is the crowding distance.

As a result, we can conclude from equation (5) that inertia weight W is $[0, W(0)]$. Particle's inertia weight with the smallest crowding distance tends to be $W(0)$ when $t = 0$ and $W(T)$ when $t = T$, and those with larger crowding distance tend to be 0. Such behavior promotes diversity, since a small crowding distance results into a large density of particles.

3.3 Learning factors

In the velocity update equation (5), higher values of C_1 ensure larger deviation of the particle in the search space, while the higher values of C_2 imply the convergence to the leader. To incorporate better compromise between the exploration and exploitation of the search space in PSO, time variant acceleration coefficients have been introduced in Ratnaweera et al. (2004). We also exploit this concept in the design of H-NSPSO. Thus, we ensure a better search for the POS in the following way: C_1 is allowed to decrease from its initial value $C_1(0)$ to $C_1(T)$, whereas C_2 can be increased from $C_2(0)$ to $C_2(T)$. Using the following equation as in Ratnaweera et al. (2004), the values of C_1 and C_2 are evaluated as follows:

$$
\begin{aligned}
C_1(t) &= (C_1(T) - C_1(0)) \cdot \frac{t}{T} + C_1(0) \\
C_2(t) &= (C_2(T) - C_2(0)) \cdot \frac{t}{T} + C_2(0) \quad (6)
\end{aligned}
$$

3.4 Mutation operator

In general, when the velocities of the particles are almost zero, it is not possible to generate new solutions which might lead the swarm out of this state. This behavior can lead to the whole swarm being trapped in a local optimum from which it becomes impossible to escape. However, since the leader attracts all members of its sub-swarm, it is possible to move the sub-swarm away from a current

location by mutating a single particle, if the mutated particle becomes the new leader. This mechanism potentially provides a means both of escaping local optima and of speeding up the search Stacey et al. (2003). Consequently, the use of a mutation operator is very important to avoid local optima and to improve the exploratory capabilities of PSO. To this end, when a solution is chosen to be mutated, each component is then mutated (randomly changed) or not with certain probability. Moreover, different mutation operators have been proposed in the literature, which mutate components of either the position or the velocity of a particle. In the case of H-NSPSO, we have employed a mutation operator that randomly changes the position of a particle in the population.

3.5 H-NSPSO algorithm

Overall, the proposed H-NSPSO can be summarized in the following algorithm:

Algorithm 1: H-NSPSO::Main()

---

for $k = 1$ **to** $popSize$ **do**
  $\vec{x}_k$ *is the position of particle* k, $\vec{x}_L$ *its lower bound and* $\vec{x}_U$ *the upper bound.* $\vec{v}_k$ *is the velocity*;
  for $i = 1$ **to** n **do**
    $\vec{x}_k(i) = rand(\vec{x}_L(i), \vec{x}_U(i))$;
    $\vec{v}_k(i) = 0$;
  end
end
$t = 0$;
while $t < T$ **do**
  childpop = pop.clone();
  childpop.assignCrowdingDistance();
  childPop.assignTopologyAndLeaders();
  foreach *particle p in childPop* **do**
    p.updateParameters(t, T);
    p.updateVelocityAndPosition();
    p.mutate();
    p.evaluate();
    p.updatePersonalBest();
  end
  childPop.add(pop);
  pop=childPop.reduceNSGA2();
  $t = t + 1$;
end

As the previous algorithm shows, initially, a random swarm *pop* is created. Next, we iterate the following procedure until the termination condition is satisfied: First, we create a copy of *pop*, called *childPop*. Then, we assign the crowding distance to each particle in the swarm. When all the crowding distances are set, we apply the proposed topology. Consequently, the fast non-dominated sorting algorithm is employed, dividing particles into non-dominated fronts. After that, we assign leaders according the following algorithm:

Algorithm 2: Pop::assignTopologyAndLeaders()

---

fronts=Pop.fastNonDominatedSort();
if $|fronts| = 1$ **then**
  assignLeadersAsInNSPSO();
  return;
end
$i = 2$;
while $i \leq |fronts|$ **do**
  frontParent=fronts[i-1];
  frontChild=fronts[i];
  $ratio = |frontChild|/|frontParent|$;
  $j = 1$;
  while $j \leq |frontChild|$ **do**
    pChild=frontChild[j];
    $p = j/ratio$;
    pChild.leader=frontParent[p].clone();
    $j = j + 1$;
  end
  $i = i + 1$;
end

In this case, we utilize the crowding distance to calculate the inertia weight W using equation (5). By means of equation (6) we calculate the learning factors (C_1 and C_2). In the following steps we apply the common functions of the PSO algorithm, including the mutation operator.

Next, we combine the previous swarm *pop* with the current one *childPop* into the new swarm, which contains $2N$ particles. Finally, the solutions of the combined swarm are sorted according to $\geq_n$, as defined in NSGA-II, and the first N points are selected. Regarding the complexity of one cycle of the entire algorithm, the basic operations being performed and the worst case complexities associated with them are the following ones:

- The crowding distance assignment is $O(mNlogN)$

- The non-dominated sort and leaders assignment has a complexity of $O(mN^2)$, and

- The sorting phase based on $\geq_n$ is $O(2Nlog(2N))$

As a result, the overall complexity of the complete algorithm is $O(mN^2)$.

---

4 MEMORY OPTIMIZATION

For having a comparison with the previous proposed H-NSPSO, a real world example on embedded applications design is studied here. In this section, we compare our algorithm with other state-of-the art results.

Latest multimedia embedded devices are enhancing their capabilities and are able to run applications reserved to powerful desktop computers (e.g., 3D games, video players). As a result, one of the most important problems designers face nowadays is the integration of a great amount of applications coming from the general-purpose domain in a compact and highly-constrained device. One major

Table 1: DDT library

DDT	Description
AR	Array
AR(P)	Array of pointers
SLL	Singly-linked list
DLL	Doubly-linked list
SLL(O)	Singly-linked list with roving pointer
DLL(O)	Doubly-linked list with roving pointer
SLL(AR)	Singly-linked list of arrays
DLL(AR)	Doubly-linked list of arrays
SLL(ARO)	Singly-linked list of arrays and roving pointer
DLL(ARO)	Doubly-linked list of arrays and roving pointer

Figure 3: DDTs optimization flow.

task of this porting process is the optimization of the dynamic memory subsystem. Thus, the designer must choose among a number of possible dynamically-allocated data structures or *Dynamic Data Types (DDTs)* implementations Antonakos and Mansfield (1999) (dynamic arrays, linked lists, etc.) the best one in each case, according to the specific restrictions of the target device and typical embedded design metrics.

4.1 The Dynamic Data Types exploration problem

A DDT is a software abstraction by means of which we can manipulate and access data. The implementation of a DDT has two main components. First, it has storage aspects that determine how data memory is allocated and freed at run-time and how this memory is tracked. Second, it includes an access component, which can refer to two different basic access patterns: sequential or iterator-based and random access.

In our case we have classified the DDT implementations in basic DDT and multi-layer implementations relevant for embedded multimedia applications. Table 1 contains the DDTs implemented Atienza et al. (2007).

Let us show an example of a DDTs exploration. Let us suppose an initial code containing two variables, *v1* and *v2*, instantiated as *vector* and *list*, respectively. After the exploration process, one candidate solution gives *v1* to be instantiated as *Single Linked List (SLL)* and *v2* as *Double Linked List of Arrays (DLLAR)*. Such instantiation policy tries to minimize memory accesses, memory usage and energy consumption of the final application. It is important to stress that it is unmanageable for the designer to get a totally complete exploration of all the possible DDT implementation combinations using the traditional way for real-life complex applications. For example, in one of the applications analyzed, we optimized memory accesses, memory usage and power consumption for 3128 variables.

In general terms, the application to optimize contains a set of n variables $\mathbf{V}$ which are candidates to be instantiated as a certain DDT from the set of possible implementation of DDTs library $\mathbf{D}$ presented in Atienza et al.
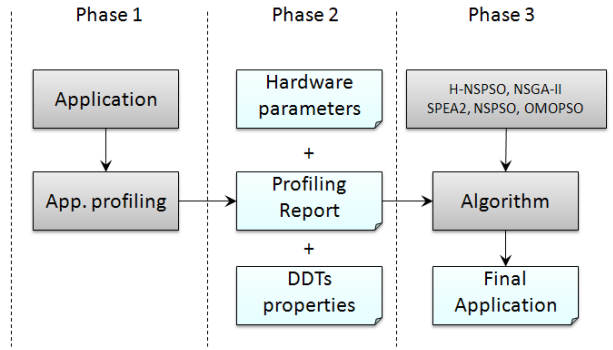
(2007), Daylight et al. (2004). Thus, the goal of our optimization flow is to obtain a set of pairs (variable, DDT) or $(\vec{v}, \vec{d}), v_i \in \mathbf{V}, d_i \in \mathbf{D}, 1 \leq i \leq n$, such that minimizes three objectives: memory accesses, memory usage and energy consumption. Additional constraints, such as minimum and maximum values for all three objectives may be defined.

The proposed optimization framework uses three different phases to perform the automatic exploration of DDTs. Figure 3 shows the different phases required to perform the overall DDTs optimization. In the first phase, we generate an initial profiling of the iterator-based access methods to the different DDTs used in the application. In the second phase, using this detailed report of the accesses, we extract all the information needed by the optimization phase. Finally, an exploration of the design space of DDTs implementation is performed using the algorithm selected. When the optimization process ends, it gives the DDT instantiation policy, i.e., which variable should be instantiated by which DDT. We also obtain the gain on memory accesses, memory usage and energy consumption. In addition to H-NSPSO, we have solved the problem using other relevant MOEAs, i.e. NSGA-II, SPEA2 Zitzler et al. (2002), OMOPSO Sierra and Coello (2005) and NSPSO.

We apply these algorithms to two multimedia embedded applications. The first benchmark is VDrift, which is a driving simulation game. The game includes 19 tracks, 28 cars, AI players, networked multiplayer mode, etc. Sourceforge (2007). We logged 49 variables in its source code. The second benchmark is a 3D Physics Engine for elastic and deformable bodies Kharevych and Khan (2002), which is a 3D engine that displays the interaction of non-rigid bodies. It includes 3128 dynamic variables in its source code for which we select the optimal DDT implementation.

4.2 Optimization model

The optimization process takes as input all the parameters obtained in the second phase in Figure 3 and minimizes three objectives: memory accesses (MA), memory usage (MU) and energy (E), defined by the following equation:

$$
\begin{aligned}
MA(\vec{v}, \vec{d}) &= f_{MA}(N_e, N_{ve}, N_r, N_w) \\
MU(\vec{v}, \vec{d}) &= f_{MU}(T_e, T_{ref}, N_e) \quad\quad (7) \\
E(\vec{v}, \vec{d}) &= f_E(N_r, N_w, N_{pa}, Hw)
\end{aligned}
$$

where,

- $\left(\vec{v}, \vec{d}\right)$ represents one candidate solution.

- $N_e\left(\vec{v}, \vec{d}\right)$ is the number of elements stored in the DDT in the worst case.

- $N_{ve}\left(\vec{v}, \vec{d}\right)$ is the average of the number of elements stored in the DDT.

- $N_r\left(\vec{v}, \vec{d}\right)$ is the number of read accesses.

- $N_w\left(\vec{v}, \vec{d}\right)$ is the number of write accesses.

- $T_e\left(\vec{v}, \vec{d}\right)$ is the size of the elements (in bytes)

- T_{ref} represents the size of the pointers, in bytes.

- $N_{pa}\left(\vec{v}, \vec{d}\right)$ is the number of cache misses.

- Hw represents the effect that hardware parameters (memory architecture, CPU power, line sizes, memory access time, etc.) have on the optimization.

In equation (7), f_{MA} and f_{MU} were taken from Atienza et al. (2007). Energy equation of the system is given by equation (8), where t_{ex} is the system's total execution time, CPU_{pow} is the total processor power excluding the cache power, C_{accE} is the cache access energ, C_{lineS} is the cache line size, $DRAM_{accP}$ is the active power consumed by the DRAM, $DRAM_{accT}$ is the DRAM latency time, and $DRAM_{bandW}$ is the bandwidth of the DRAM.

$$
\begin{aligned}
f_E &= t_{ex} \times CPU_{pow} + \\
&\quad (N_r + N_w) \times (1 - N_{pa}) \times C_{accE} + \\
&\quad (N_r + N_w) \times N_{pa} \times C_{accE} \times C_{lineS} + \\
&\quad (N_r + N_w) \times N_{pa} \times DRAM_{accP} \times \\
&\quad \left(DRAM_{accT} + \frac{C_{lineS}}{DRAM_{bandW}} \right) \quad\quad (8)
\end{aligned}
$$

There exist four components in the energy equation (8). The first term $t_{ex} \times CPU_{pow}$ calculates the processor energy given that execution time takes t_{ex} amount of time. The second term, $(N_r + N_w) \times (1 - N_{pa}) \times C_{accE}$ calculates the amount of energy consumed by the cache. The third term, $(N_r + N_w) \times N_{pa} \times C_{accE} \times C_{lineS}$ calculates the energy cost of writing to cache for each cache miss. The last term, calculates the energy cost of the DRAM to service all the cache misses.

Table 2: Coding a solution

d_1	d_2	d_3	$\ldots$	d_n	$d_j \in \mathbf{D}$
v_1	v_2	v_3	$\ldots$	v_n	$v_j \in \mathbf{V}$

The equation for calculating the system's total execution time t_{ex} is given by equation (9), where C_{accT} is the access time of the cache.

$$
\begin{aligned}
t_{ex} &= (N_r + N_w) \times (1 - N_{pa}) \times C_{accT} + \\
&\quad (N_r + N_w) \times N_{pa} \times DRAM_{accT} + \\
&\quad (N_r + N_w) \times N_{pa} \times \frac{C_{lineS}}{DRAM_{bandW}} + \\
&\quad T_{bus} \quad\quad (9)
\end{aligned}
$$

There exist four components in the system's execution time shown in equation (9). The first term $(N_r + N_w) \times (1 - N_{pa}) \times C_{accT}$ is for calculating the amount of time taken for the processor to access the cache. The second term $(N_r + N_w) \times N_{pa} \times DRAM_{accT}$ calculates the amount of time required for the DRAM to respond to each cache miss. The third term calculates the amount of time taken to fill a cache line on each cache miss. The bus communication time cost is supposed to be constant (T_{bus}). As the bus communication time is expected to be similar to other systems, such decision will not adversely affect the final results.

Units for time variables in the equations are in seconds, bandwidth is in Bytes/second, cache line size is in Bytes, power variable is in Watts, and energy unit is in Joules.

Table 2 shows the representation of a candidate solution (gray shaded cells). Each candidate solution represents the set of DDT that should be used to instantiate all the corresponding variables in the application from Table 1. For example, the second variable $v_2 \in \mathbf{V}$ will be instantiated by $d_2 \in \mathbf{D}$. A candidate solution contains n integer fields, where n is the number of the variables logged in the application, $n = size(\mathbf{V})$. The constraint a field must satisfy is $1 \leq d_i \leq size(\mathbf{D})$.

4.3 Experimental methodology

The model of the embedded system architecture consisted of a processor with an instruction cache, a data cache, and embedded DRAM as main memory. The data cache uses a write-through strategy. We utilized processor energy from Catthoor et al. (2002), and the access time and energy values for caches of 32KB and embedded 16MB DRAM main memory from Shivakumar and Jouppi (2001) and Hardee et al. (2004), respectively. The processor and memory specification is described in Table 3.

Since the size of possible DDT implementations is large and it is not possible to cover the exact set of the POF, we compare the obtained Pareto Front (PF) with each other using the hypervolume metric Zitzler et al. (2003). This

Table 3: System specification

Processor Energy	168mW, 100MHz
Embedded DRAM	100MHz
Energy	19.5 mW
Latency	19.5 ns
Bandwidth	50MB/s

metric calculates the volume (in the objective space) covered by members of a nondominated set of solutions Q. Let v_i be the volume enclosed by solution $i \in Q$. Then, a union of all hypercubes is found and its hypervolume (H_V) is calculated.

$$H_V = \bigcup_1^{|Q|} v_i \qquad (10)$$

The hypervolume of a set is measured relative to a reference point, usually the anti-optimal point or "worst possible" point in space. We do not address here the problem of choosing a reference point, if the anti-optimal point is not known or does not exist one suggestion is to take, in each objective, the worst value from any of the fronts being compared. If a set X has a greater hypervolume than a set Y, then X is taken to be a better set of solutions than Y. Since this metric is not free from arbitrary scaling of objectives, we have evaluated the metric by using normalized objective function values using PISA Bleuler et al. (2003). Note that in the case of PISA, a lower indicator value corresponds to a better approximation set.

Finally, to compare the performance of five algorithms, all parameters are set as follows:

- Population/Swarm size: 100 in the case of VDrift and 200 in the case of Physics.

- Number of iterations: 2000 for VDrift and 4000 for Physics.

- Crossover: Real-parameter SBX crossover operator ($\eta_c = 20$). Crossover probability of 0.9 (as suggested in Deb et al. (2002)) for NSGA-II and SPEA2.

- Mutation: Polynomial mutation operator ($\eta_m = 20$), with probability inversely proportional to the chromosome length (as suggested in Deb et al. (2002)) for NSGA-II and SPEA2. Uniform and non-uniform mutation applied to H-NSPSO and OMOPSO, as suggested in Sierra and Coello (2005).

- Coding strategy: Real encoding for all the five algorithms.

- $C_1(0) = 2.5$, $C_1(T) = 0.5$, $C_2(0) = 0.5$, and $C_2(T) = 2.5$ (as suggested in Ratnaweera et al. (2004)) for H-NSPSO. $C_1 = 2.0$ and $C_2 = 2.0$ for NSPSO.

- $W(0) = 0.7$ and $W(T) = 0.4$ (as suggested in Bergh (2002)) for H-NSPSO. $W = 0.4$ for NSPSO.
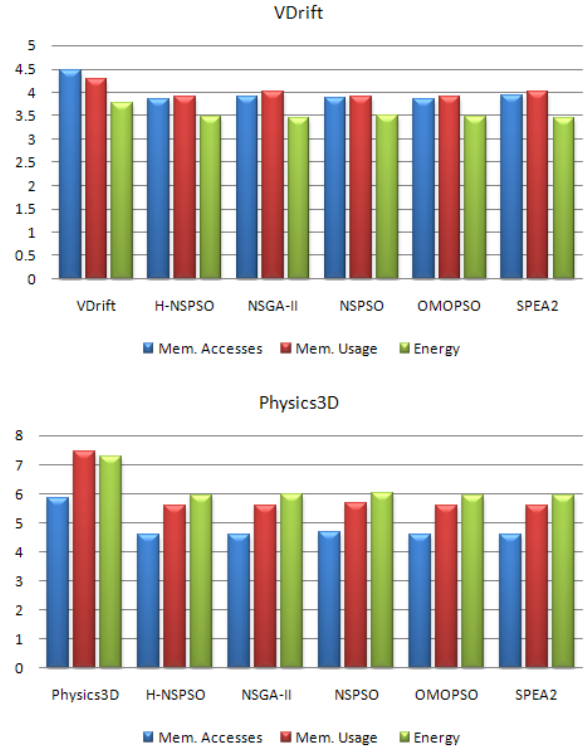

Figure 5: Comparison of the real application with results obtained by our design framework (logarithmic scale).

4.4 Results

We have explored DDTs for VDrift and Physics with each of the five algorithms proposed (H-NSPSO, NSGA-II, NSPSO, OMOPSO and SPEA2). The hypervolume values are calculated by averaging results of 30 trials. Figure 4 depicts the first attainment surfaces obtained for both applications.

With respect to VDrift, Figure 4 shows that the surfaces for H-NSPSO, OMOPSO and SPEA2 outperform the surfaces offered by NSPSO and NSGA-II. Regarding H-NSPSO, OMOPSO and SPEA2, no significant differences can be detected, as occurs in the case of Physics, where all the obtained surfaces are incomparable.

For comparison reasons we present Figure 5 to illustrate the optimization process that our methodology performs. In this test, each application is evaluated with their original DDTs and compared to the combination proposed by our framework. The figure shows clearly the achieved level of optimization and final gains after applying the proposed optimization flow in Figure 3. It should be noted that H-NSPSO offered the best gain in at least two objectives.

Table 4 shows the hypervolume or S-metric obtained for VDrift and Physics. In both cases, H-NSPSO algorithm reaches better values compared to the other MOEAs. Thus, the result set from H-NSPSO algorithm is taken to be a better set of solutions than those obtained from other algorithms.
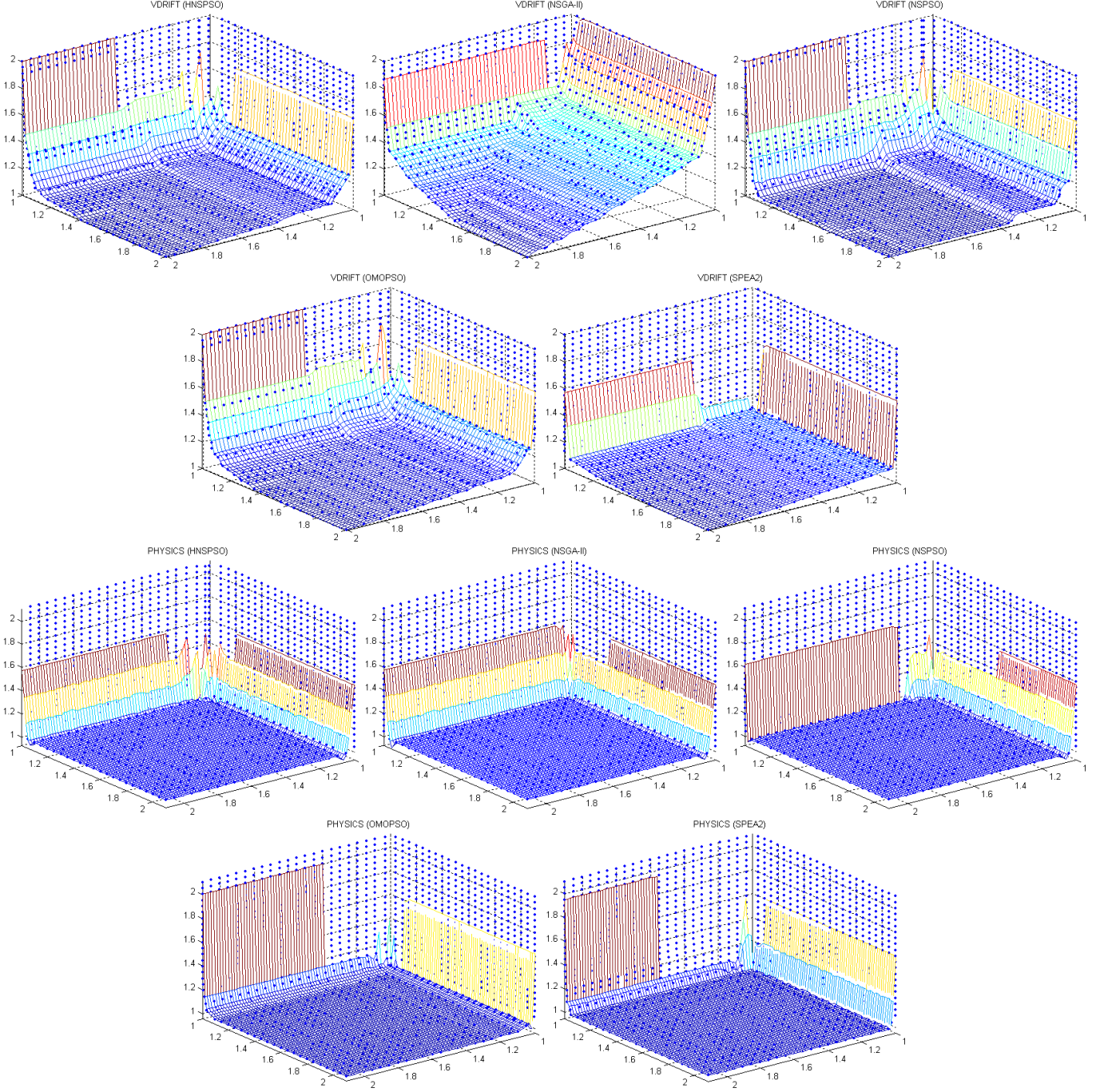
Figure 4: First attainment surfaces obtained for VDrift and Physics using H-NSPSO, NSGA-II, NSPSO, OMOPSO and SPEA2.

5 CONCLUSIONS

In the present article, a novel multi-objective PSO algorithm, called H-NSPSO, has been presented. H-NSPSO is adaptive in nature with respect to its inertia weight and learning factors. This adaptation enables it to attain a good balance between the exploration and the exploitation of the search space. The crowding distance is used to compute the inertia weight, promoting diversity. A mutation operator has been incorporated in H-NSPSO to resolve the problem of premature convergence to the local

Pareto-optimal front (often observed in the multi-objective PSOs). An external archive has been maintained to store the non-dominated solutions found during H-NSPSO execution. The selection of the leader is done from this archive, where all particles are arranged in several tree networks that define the neighborhood structure. H-NSPSO has been compared in a real world application: the optimization of Dynamic Data Types in embedded applications. Results show that H-NSPSO offers better results with respect to the other four algorithms tested.

Table 4: Hypervolume metric for VDrift/Physics.

Algorithm	VDrift	Physics 3D
H-NSPSO	-1.1784 ± 0.0092	-1.1992 ± 0.1715
NSGA-II	-0.9356 ± 0.0231	-1.1600 ± 0.1902
NSPSO	-1.1360 ± 0.0363	-1.1830 ± 0.1074
OMOPSO	-1.1657 ± 0.0114	-0.7215 ± 0.2449
SPEA2	-1.1492 ± 0.0374	-0.9002 ± 0.1899

REFERENCES

Antonakos, J. L. and Mansfield, K. C. (1999). *Practical Data Structures using C/C++*. Prentice Hall.

Atienza, D., Baloukas, C., Papadopoulos, L., Poucet, C., Mamagkakis, S., Hidalgo, J. I., Catthoor, F., Soudris, D., and Lanchares, J. (2007). Optimization of dynamic data structures in multimedia embedded systems using evolutionary computation. In *SCOPES '07: Proceedingsof the 10th international workshop on Software & compilers for embedded systems*, pages 31–40, New York, NY, USA. ACM.

Bergh, F. V. D. (2002). *An analysis of particle swarm optimizers*. PhD thesis, University of Pretoria, Pretoria, South Africa, South Africa.

Bleuler, S., Laumanns, M., Thiele, L., and Zitzler, E. (2003). PISA - a platform and programming language independent interface for search algorithms. In Fonseca, C. M., Fleming, P. J., Zitzler, E., Deb, K., and Thiele, L., editors, *Evolutionary Multi-Criterion Optimization (EMO 2003)*, Lecture Notes in Computer Science, pages 494–508, Berlin. Springer.

Catthoor, F., Danckaert, K., Kulkarni, C., Brockmeyer, E., Kjeldsberg, P. G., Achteren, T. V., and Omnes, T. (2002). *Data access and storage management for embedded programmable processors*. Kluwer Academic Publishers.

Daylight, E. G., Atienza, D., Vandecappelle, A., Catthoor, F., and Mendias, J. M. (2004). Memory-access-aware data structure transformations for embedded software with dynamic data accesses. *IEEE Transactions on VLSI Systems*, 12:269–280.

Deb, K. (2001). *Multiobjective Optimization using Evolutionary Algorithms*. John Wiley and Son Ltd.

Deb, K., Pratap, A., Agarwal, S., and Meyarivan, T. (2002). A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197.

Eberhart, R. C. and Shi, Y. (1998). Comparison between genetic algorithms and particle swarm optimization. In Porto, V. W., Saravanan, N., Waagen, D., and Eibe, A., editors, *Proceedings of the Seventh Annual Conference on Evolutionary Programming*, pages 611–619. Springer-Verlag.

Engelbrecht, A. (2002). *Computational Intelligence: An Introduction*. Halsted Press, New York, NY, USA.

Hardee, K., Jones, F., Butler, D., Parris, M., Mound, M., Calendar, H., Jones, G., Aldrich, L., Gruenschlaeger, C., Miyabayashil, M., Taniguchi, K., and Arakawa, I. (2004). A 0.6v 205mhz 19.5ns trc 16mb embedded dram. In *IEEE International Solid-State Circuits Conference (ISSCC)*.

Janson, S. and Middendorf, M. (2005). A hierarchical particle swarm optimizer and its adaptive variant. *Systems, Man, and Cybernetics, Part B, IEEE Transactions on*, 35(6):1272–1282.

Kennedy, J. (1999). Small worlds and mega-minds: effects of neighborhood topology on particle swarm performance. In *Proceedings of the 1999 Congress on Evolutionary Computation, 1999. CEC 99*.

Kharevych, L. and Khan, R. (2002). 3d physics engine for elastic and deformable bodies. Available: http://www.cs.umd.edu/Honors/reports/kharevych.html. University of Maryland, College Park.

Li, X. (2003). A non-dominated sorting particle swarm optimizer for multiobjective optimization. In Cantú-Paz, E., Foster, J. A., Deb, K., Davis, D., Roy, R., O'Reilly, U.-M., Beyer, H.-G., Standish, R., Kendall, G., Wilson, S., Harman, M., Wegener, J., Dasgupta, D., Potter, M. A., Schultz, A. C., Dowsland, K., Jonoska, N., and Miller, J., editors, *Genetic and Evolutionary Computation – GECCO-2003*, volume 2723 of *LNCS*, pages 37–48, Chicago. Springer-Verlag.

Moore, J. and Chapman, R. (1999). Application of particle swarm to multiobjective optimization. Department of Computer Science and Software Engineering, Auburn University.

Mostaghim, S. and Teich, J. (2003). Strategies for finding good local guides in multi-objective particle swarm optimization (MOPSO). In *2003 IEEE Swarm Intelligence Symposium Proceedings*, pages 26–33, Indianapolis, Indiana, USA. IEEE Service Center.

Ratnaweera, A., Halgamuge, S. K., and Watson, H. C. (2004). Self-organizing hierarchical particle swarm optimizer with time-varying acceleration coefficients. *Evolutionary Computation, IEEE Transactions on*, 8(3):240–255.

Reyes-Sierra, M. and Coello, C. A. C. (2006). Multi-objective particle swarm optimizers: A survey of the state-of-the-art. *International Journal of Computational Intelligence Research*, 2(3):287–308.

Shivakumar, P. and Jouppi, N. P. (2001). Cacti 3.0: An integrated cache timing, power, and area model. Technical Report 2001/2, Compaq Computer Corporation.

Sierra, M. R. and Coello, C. A. C. (2005). Improving pso-based multi-objective optimization using crowding, mutation and epsilon-dominance. In *EMO*, pages 505–519.

Sourceforge (2007). Vdrift racing simulator. Available: http://sourceforge.net/projects/vdrift.

Stacey, A., Jancic, M., and Grundy, I. (2003). Particle swarm optimization with mutation. In *Proceedings of the Congress on Evolutionary Computation, CEC03.*, volume 2, pages 1425–1430.

Zitzler, E. (1999). *Evolutionary Algorithms for Multiobjective Optimization: Methods and Applications.* PhD thesis, Swiss Federal Institute of Technology (ETH), Zurich, Switzerland.

Zitzler, E., Laumanns, M., and Thiele, L. (2002). SPEA2: Improving the strength pareto evolutionary algorithm for multiobjective optimization. In *Proceedings of the Evolutionary Methods for Design, Optimization and Control with Application to Industrial Problems*, pages 95–100, Barcelona, Spain.

Zitzler, E., Thiele, L., Laumanns, M., Fonseca, C., and da Fonseca, V. (2003). Performance assessment of multi-objective optimizers: an analysis and review. *Evolutionary Computation, IEEE Transactions on*, 7(2):117–132.