Autonomic Diffusive Load Balancing on Many-core Architecture using Simulated Annealing

Hyunjik Song and Kiyoung Choi
School of Electrical Engineering and Computer Science
Seoul National University
Seoul, Republic of Korea
pupasong@dal.snu.ac.kr, kchoi@snu.ac.kr

***Abstract*—Many-core architecture is becoming an attractive design choice in high-end embedded systems design. There are, however, many important design issues, and load balancing is one of them. In this work, we take the approach of diffusive load balancing which enables autonomic load distribution in many-core systems. We modify the existing scheme by adding the concept of simulated annealing for more effective load distribution. The modified scheme is also capable of managing a situation of non-uniform granularity of task loading, which the existing ones cannot. As experiments, we tried various existing schemes as well as the proposed one to map a synthetic application with 30 threads on a many-core architecture with 21 cores and 4 memory tiles. The experiments show that the modified scheme gives results better than the existing approaches.**

***Keywords*-diffusive load balancing; simulated annealing.**

I. INTRODUCTION

The rapid increase of semiconductor density has enabled today's high-performance embedded systems to have many-core architecture. However, to utilize maximally the ability of the system, it is very important to map parallel threads properly onto the many-core architecture. There have been numerous studies on this topic, which can be classified into two types: design-time solution and run-time solutions. Design-time solutions determine the mapping during design steps and use the result as a static thread mapping during run time [1]. The major advantage is to remove the overhead of transient thread migration and mapping calculation which run-time counterparts suffer from. The main limitations of design-time solutions are (1) the scalability problem in exploring the entire design space due to the exponential complexity of the mapping problem with respect to the number of parallel threads and (2) lack of adaptivity to changing application behavior. In run-time solutions, known as dynamic load balancing, decentralized methods are favored since centralized ones suffer from the scalability problem (e.g., in gathering global load information and mapping calculation). Our work is based on diffusive load balancing [2][3], which is one of representative decentralized methods for load balancing. Decentralized methods, however, can result in globally gradual load imbalance, which forbids diffusive load balancing to achieve perfect load balancing. Our idea is to borrow the concept of simulated annealing [4] which has a good characteristic to escape local optimum. When the diffusive load balancing scheme suffers from globally gradual load imbalance, allowing thread migration from under-loaded core to over- loaded core with some probability helps proceed toward the perfect load balancing. The load balancing problem gets more complicated when the loadings of threads are not uniform. Whereas existing diffusive load balancing schemes have little capability to manage such multi-threaded applications, our modified approach can effectively handle the situation. The remainder of this paper is organized as follows. Section II reviews related work and Section III introduces diffusive load balancing in terms of negotiation process. Section IV presents our idea using the concept of simulated annealing. Section V gives experimental results and Section VI concludes the paper.

II. RELATED WORK

Dynamic load balancing policies are classified into direct and iterative ones. Direct load balancing maps threads onto cores in one step [5][6]. Iterative load balancing maps threads incrementally onto cores by migrating threads to neighbor cores and by repeating the migration steps until equilibrium is reached [2][3][7][8][9]. Direct load balancing methods remove redundant neighbor-to-neighbor thread migrations which iterative methods suffer from. However, direct methods have a significant limitation of high overhead in gathering global information (via core to core communication to exchange load information) and calculating thread mapping based on the global information (running a bin packing algorithm during run time). Iterative methods determine load balancing decisions mostly based on local load information. Thus, the overhead of mapping decision is low. However, the quality of mapping is limited due to the lack of global information and thread migration overhead as explained before. In our work, we aim to obtain perfect or close to perfect load balancing with only local information by adopting the concept of simulated annealing. The approach proposed in [10] also uses the concept of simulated annealing to tackle the diffusive load balancing problem to obtain better load balancing. In this

 ISBN: 978-1-61208-257-8

work, we implement the approach more elaborately and demonstrate its effectiveness by comparing it with various existing diffusive load balancing approaches. In addition, we extend the approach to the case of non-uniform thread loading applications.

III. Run-time Diffusive Load Balancing

A. Negotiation

Diffusive load balancing tackles load balancing problem by mimicking the physical phenomenon of diffusion. The diffusion forces a system towards a stable (minimum- energy) state with homogeneous distribution (of density of molecules, pressure, etc.). It achieves this by displacing physical objects (mostly, molecules in nature) along the direction of decreasing energy obtained by comparing local states (e.g., local level of concentration or pressure) in a decentralized manner. Diffusive load balancing tries to achieve balanced load distribution in the same way as diffusion in nature. In diffusive load balancing, a thread scheduler running on each core takes a policy of load balancing that is similar to physical laws applied to diffusion. Each core performs load balancing autonomously in a fully decentralized manner as in nature by utilizing only local load information. That is, each core tries to balance the load distribution in a local area. The collective efforts of each core's load balancing force the global load distribution towards the homogeneous state, i.e., equal load distribution. Each core identifies the state of its load (under-loaded, balanced, or over-loaded) by comparing its own load and that of its neighbors (i.e., local information). If its load state is not balanced, it tries to make it balanced by negotiating with its neighbors on load redistribution, i.e., thread migration. Negotiation is the process to determine, if any, sender and receiver cores to migrate threads. The coverage of neighborhood in negotiation called negotiation coverage (i.e., participants in the negotiation) is one of the most important parameters since it affects the quality of diffusive methods. The case of involving only direct neighbor cores may suffer from lack of global knowledge while too wide a coverage may cause inefficiency due to the increased overhead of load information collection and negotiation. There are four types of negotiation coverage proposed previously: direct neighborhood (DN), average extended neighborhood (AN-d), direct neighborhood repeated (DNR) [2] and direct neighborhood with distorted load information (DND) [3]. In DN, each negotiation covers only two direct neighbor cores. One core initiates the negotiation if it detects load imbalance when comparing its load with that of the other core. Then the initiating core asks the other to balance the load by sending (receiving) thread(s) to (from) the other. The other does not accept the request when the migration reverse the sender/receiver roles of the two cores since, if accepted, it will cause the ping-pong situation where the two cores will continue to exchange the same thread in a ping-pong manner. In AN-d, given a center core, its d-hop neighbor cores participate in the negotiation. Note that as the negotiation coverage increases, the overhead of negotiation increases, so we only consider only 1-hop, i.e., we are only concerned about AN-1(AN) scheme. DNR allows cores to forward a thread from one direct neighbor core to another if there is load difference between them, and DND is the same as DN except that each core gives distorted load information to consider the neighbors' load status.

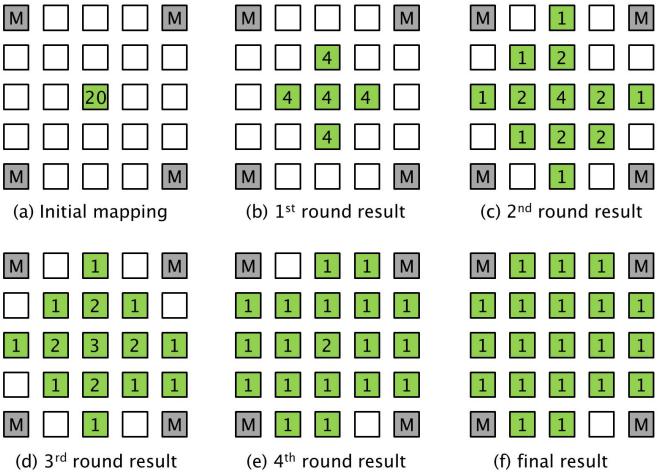

Figure 1. An illustration of diffusive load balancing

B. Motivational Example

Fig. 1 shows an example of load distribution obtained by applying the diffusive load balancing to a multi-threaded application on a 21 core architecture. In the figure, a rectangle represents a tile with a core or a memory. The memory tile is annotated with 'M'. The followings are our assumptions in the example.

- The multi-threaded application with 20 threads has 21 core tiles and four memory tiles.
- Any existing scheme including DN, AN, DNR, or DND can be used as the negotiation coverage and the minimum thread load is '1'
- Each thread utilizes one target memory and each memory is utilized by three distinct threads.
- In Fig.1(a), we assume that the entire thread set consisting of 20 threads is initially mapped on a core at (2,2) when the simulation starts ((0, 0) indicates the tile at the upper left corner and (0, 4) indicates the one at the upper right corner).

The threads are mapped onto shaded core tiles in the figure. The number on each shaded core tile represents its load level. Fig. 1(b)-(f) show the results of diffusion rounds.

Note that the diffusion rounds require a large number of intermediate neighbor-to-neighbors thread migrations. Consider the transition from Fig. 1(e) to (f). The core at (2, 2) is over-loaded since its load level is higher than the global

 ISBN: 978-1-61208-257-8

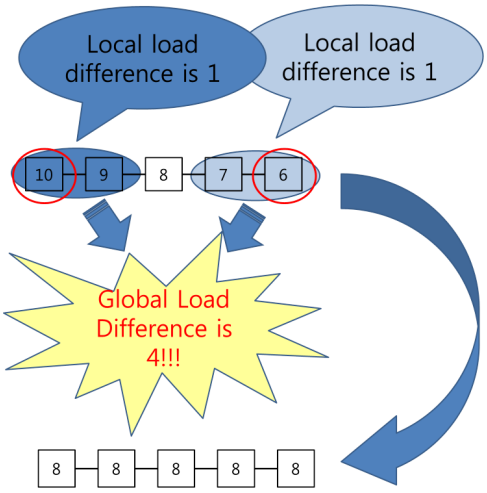

Figure 2. Gradual load imbalance

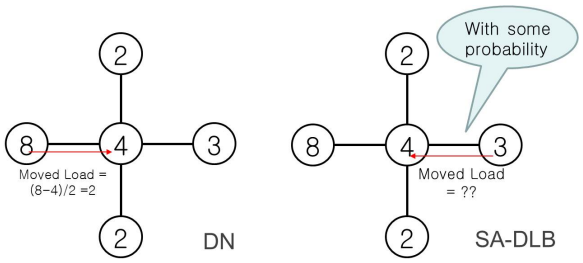

Figure 3. Simulated annealing-based diffusive load balancing

average load level. On the other hand, the core at (0, 1) and (4, 3) is under-loaded. In the diffusive load balancing with DN as the negotiation coverage, the transition from Fig. 1(e) to (f) cannot take place because any trial to balance the local load with the core at (2, 2) will only change the role of sender and receiver and because the over-loaded core does not know which core is the right receiver due to the lack of global knowledge about under-loaded cores. This is a globally gradual load imbalance problem and unless we have some smart scheme, the transition from Fig. 1(e) to (f) cannot happen. This will be discussed in the next section in more detail.

In terms of performance, global load imbalance can cause significant degradation of the overall system performance. For instance, when a mapping similar to Fig. 1(e) continues, the maximum load '2' determines the total execution time which is 50% longer than the case of ideal load balancing where the maximum load of the 12 cores is '1'.

IV. Using Simulated Annealing

A. Globally Gradual Load Imbalance

Diffusive load balancing, when implemented, has a major limitation that it lacks global knowledge. It prevents diffusive load balancing from achieving perfect load balancing. Diffusion of threads, i.e., thread migration is intrinsically based on local load information. Thus, there can be globally gradual load imbalance, since it is difficult to be captured by local information as our motivational example shows in Fig. 2. Even though every adjacent pair of two cores is balanced, global load difference is 4. Sharing global load information among cores could resolve the problem. However, it incurs prohibitively high overhead in continuously collecting the global information from the large number of cores. Thus, the real implementation of diffusive load balancing can fail to achieve perfect load balancing due to the lack of global knowledge. It is like a situation that the global optimum cannot be obtained because the solution is not able to escape a local optimum in typical optimization problems. In order to get the global optimum, it is quite effective to exploit metaheuristics such as genetic algorithm, ant-colony optimization, tabu search, and simulated annealing.

B. Uniform Load Granularity

Because diffusive load balancing is executed in run-time, applying metaheuristics with huge overhead is not feasible. We accordingly borrow only the concept of simulated annealing, which includes cooling schedule and probability of accepting moves, and so on. The approach in [9] also adopts the concept of simulated annealing for load balancing. However, it does not have the concept of cooling schedule, which will make it difficult to converge to an optimal state. Moreover, there is no experiment given to show the effect of adopting the simulated annealing approach. In the proposed simulated annealing-based diffusive load balancing (SA-DLB) approach, each core performs load balancing based on DN negotiation scheme. The distinguished element of SA-DLB, however, is that an under-loaded core can migrate its threads to an over-loaded one in accordance with a calculated probability, whereas the previous works such as DN, AN, DNR, and DND, etc. do not admit it as shown in Fig. 3. To do so, each core maintains a temperature value internally by lowering the value with a given period. The higher the temperature is, it is more likely for threads to be migrated from the under-loaded core to the over-loaded core, and as the temperature falls down, that kind of migration is observed less frequently. Therefore, the effect of escaping from a local optimum can be seen in the global view, which means there is a good chance of better load balancing. When the statistics of the application loading is constant in time (e.g., the number of threads and the loading of each thread can be modeled as ergodic processes), SA-DLB outperforms other schemes, which has been confirmed by the experiment presented in Section V. A detailed description is shown in Algorithm 1. First, initialize the Temperature value, the most important parameter in simulated annealing. Then, we determine which core will be a sender or receiver. Either an over-loaded core or an under-loaded core can be a sender,

 ISBN: 978-1-61208-257-8

Algorithm 1 SA-based Migration (Uniform Load Granularity)

$t \leftarrow 0$
$initialize\ T//Temperature$
repeat
  $r \leftarrow random(0\ or\ 1)$
  if $r = 0$ **then**
    set over-loaded core as a sender
    and under-loaded core as receiver
  else if $r = 1$ **then**
    set under-loaded core as a sender
    and under-loaded core as receiver
  end if
  $load\_gap_{current} \leftarrow load_{sender} - load_{receiver}$
  $n \leftarrow random(0 \ldots load_{sender})$
  for $i = 0$ to n **do**
    $load_{sender} \leftarrow load_{sender} - 1$
    $load_{receiver} \leftarrow load_{receiver} + 1$
  end for
  $load\_gap_{new} \leftarrow load_{sender} - load_{receiver}$
  $gain \leftarrow |load\_gap_{current}| - |load\_gap_{new}|$
  if $gain \geq 0$ **then**
    commit migration
  else
    if $random[0,1) < e^{\frac{|load\_gap_{current}| - |load\_gap_{new}|}{kT}}$ **then**
      commit migration
    end if
  end if
  $T \leftarrow g(T,t)$
  $t \leftarrow t + 1$
until (halting-criterion)

and the other is set to be a receiver. A sender core generates a random number at most its number of threads. For example, a sender that has three threads can generate 0, 1, 2, or 3. Then the sender calculates the load gap betweet the sender and the receiver that will be obtained after migrating threads as many as the generated random number. The computed load gap is used to determine whether the probabilistic move will be accepted or not. This process is done iteratively with the temperature value decreasing. In our environment, the temperature value is decreased by 10% at every step of iteration as described with g(T,t) in Algorithm 1. The halting-criterion is satisfied when the temperature reaches 10% of the initial value.

C. Non-uniform Load Granularity

As explained in previous subsection, in the SA-DLB method, it is assumed that every thread has the same load size, which is also assumed in the previous approaches such as DN, AN, DND, and DNR. Therefore, the load of each core is calculated as the number of threads assigned to that

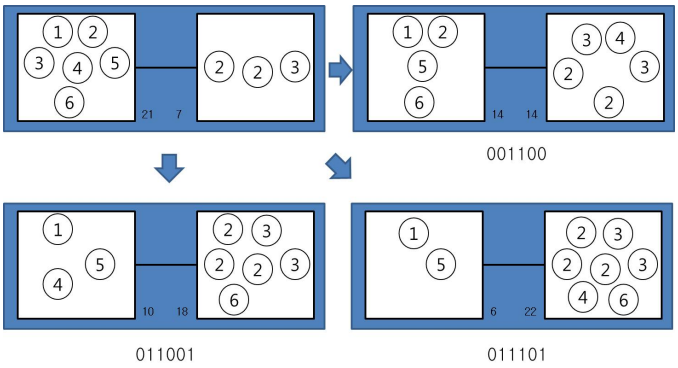

Figure 4. Move generation for non-uniform load situation

core. In this case, any thread can be migrated and it is easy to determine how much loads are to be migrated. For example, to migrate load of amount 10 to a receiver core, the sender core can just send 10 threads. However, the problem becomes more complicated when the application has threads with various kinds of load quantity, i.e., non-uniform load granularity. Because the number of threads does not mean the quantity of load of a core, it is no longer simple to determine how many threads are to be migrated. This implies that it is no more suitable to apply the existing methods such as DN, etc. If we apply SA-DLB, however, we can easily determine which threads are to be migrated and how much load is sent to the target core. Fig. 4 shows the example of move generation of SA-DLB in the situation of non-uniform load granularity. Each white rectangle represents a processor core and each circle represents a thread assigned to the core. Each thread is annotated with its load value and this value is also used as the identifier of the thread (threads with the same load value do not need to be distinguished). In the upper left pair of Fig. 4, one core has six threads with 21 load and the other core has three threads with 7 load. If the DN method is applied to balance the two cores, the left core could send 7 loads to the right one, which makes both cores have the same load, i.e. 14. However, it is not easy to select appropriate threads, so achieving load balance in the global view would be much difficult. Before applying SA-DLB, we assume that the over-loaded core manages the process of load balancing between two cores. We call the managing core a 'controller'. The controller generates random one-bit binary numbers as many as its number of threads. If the generated random numbers are '011001' as in the lower left case of Fig. 4, the threads '2','3', and '6', which correspond to '1', are regarded as candidates to migrate. If these three threads are moved to the right core, the cores will have 10 and 18 load respectively, and this makes the load difference of the two cores 8, which is smaller than the original gap, 14. In this case, the migration is accepted because the load gap is decreased, implying that the degree of imbalance between two cores is decreased. If

 ISBN: 978-1-61208-257-8

Algorithm 2 SA-based Migration (Non-Uniform Load Granularity)

$t \leftarrow 0$
$initialize\ T//Temperature$
repeat
  $r \leftarrow random(0\ or\ 1)$
  if $r = 0$ **then**
    set over-loaded core as a sender
    and under-loaded core as receiver
  else if $r = 1$ **then**
    set under-loaded core as a sender
    and under-loaded core as receiver
  end if
  $load\_gap_{current} \leftarrow load_{sender} - load_{receiver}$
  for $i = 0$ to $number\_of\_sender\_threads$ **do**
    $rand \leftarrow random(0\ or\ 1)$
    if $rand = 1$ **then**
      $load_{sender} \leftarrow load_{sender} - load_i$
      $load_{receiver} \leftarrow load_{receiver} + load_i$
    end if
  end for
  $load\_gap_{new} \leftarrow load_{sender} - load_{receiver}$
  $gain \leftarrow |load\_gap_{current}| - |load\_gap_{new}|$
  if $gain \geq 0$ **then**
    commit migration
  else
    if $random[0,1) < e^{\frac{|load\_gap_{current}| - |load\_gap_{new}|}{kT}}$ **then**
      commit migration
    end if
  end if
  $T \leftarrow g(T, t)$
  $t \leftarrow t + 1$
until (halting-criterion)

the generated random numbers are '011101' as in the case of lower right in Fig. 4, the resulting gap between the cores is 18 which is bigger than the original value. Even though the imbalance becomes worse, the migration for '011101' is not discarded immediately. Instead, the migration could be accepted with some probability which is shown in Algorithm 2 in detail. For convenience, we call the algorithm a SA-DLB-NU.

V. EXPERIMENTS

A. Target Many-Core Architecture

The target many-core architecture consists of 21 ARM946ES core tiles, 4 SRAM memory tiles and 5x5 mesh NoC as shown in Fig.1. The NoC performs XY and worm-hole routing without virtual channel. The NoC router has five ports (one for the local NI, and the other four for neighboring routers). A flit has 8 bytes and a packet has 1 9 flits. The

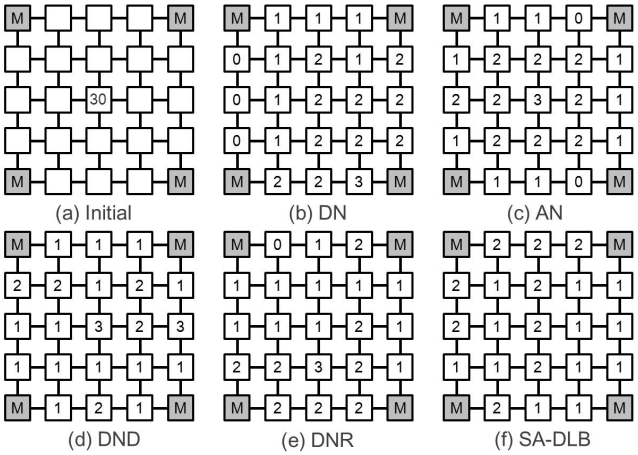

Figure 5. Simulation Results

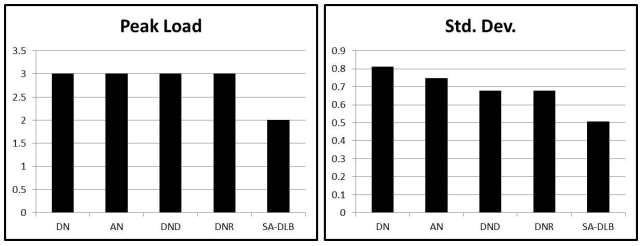

Figure 6. Peak load and standard deviation

NI has a buffer for two packets for pipelined operation. We designed the entire system with a commercial transaction level simulation environment, Carbon SoC Designer [11].

B. Experimental Results

Fig. 5 shows the results of load distribution obtained by applying the diffusive load balancing to a multi-threaded application on a 21 core architecture (memory tiles are placed at locations different from those in Fig. 1 to see the effects more clearly but the results are not much different). Each white rectangle represents a processor core. The memory tile is annotated with 'M'.

We start with an entire thread set consisting of 30 threads initially mapped on a core at (2, 2) as shown in Fig. 5(a). Simulation results of previous approaches including DN, AN, DND, and DNR are shown in Fig. 5(b)-(e), respectively, together with that of the proposed SA-DLB in Fig. 5(f). In the result of DN, AN, and DNR, threads are well distributed in the sense that for every pair of two neighboring cores the load difference is less than or equal to one. However, we can easily observe global imbalance; in DN scheme, the core at (3, 4) has three threads while cores at (0, 1), (0, 2), and (0, 3) have no thread. Moreover, in the DND scheme, cores at (2, 2) and (1, 2) have a gap of 2. On the other hand, SA-DLB can distribute the thread perfectly as shown in Fig. 5(f). In Fig. 6, the peak value and standard

 ISBN: 978-1-61208-257-8

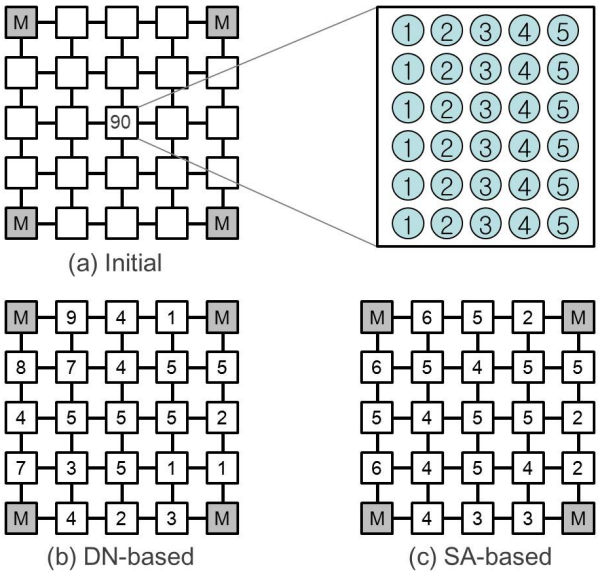

Figure 7. Simulation results

Table I
SIMULATION RESULTS

	DN-based	SA-based
average	4.28	4.28
standard deviation	2.24	1.27
maximum load	9	6
minimum load	1	2
difference	8	4

deviation of distributed load are shown. Even though the improved version of DN such as AN, DNR, and DND has better standard deviation, our SA-DLB gives the best result in terms of standard deviation. Although SA-DLB provides a better performance for load balancing, it tends to take 3∼4 times longer than other approaches, since it has higher computational complexity.

Fig. 7 shows the result of SA-DLB in the case of non-uniform load granularity. Initially, the many-core system has 30 threads with total load amount of 90 on a core at (2, 2). Because there is no previous work that is capable of managing non-uniform cases, we compare our SA-DLB-NU with a DN-like scheme that can handle such cases. The DN-like scheme tries to balance the load with its direct neighbor by migrating threads in a way to make two involving cores to have as same load as possible. Table I summarizes the results shown in Fig. 7. SA-DLB-NU has less standard deviation and the maximum loading is also less than the DN-based diffusion scheme.

VI. CONCLUSION AND FUTURE WORK

In this contribution, we have introduced a new negotiation scheme for run-time diffusive load balancing on many-core SoC architecture. By adopting the concept of escaping local optimum in simulated annealing, we can distribute parallel threads more evenly than by using existing negotiation techniques. Our new negotiation scheme can manage the situation of a multi-threaded applications with various thread load granularity. Future work will include making our approach applicable to applications with communicating threads. In addition to the communication of threads, we believe that SA-DLB can also be used for balancing load of applications with dynamically varying statistics.

ACKNOWLEDGMENT

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MEST) (No. 2012-0006272)

REFERENCES

[1] Y. Ahn, K. Han, G. Lee, H. Song, J. Yoo, X. Feng, and K. Choi, "Socdal: System-on-chip design accelerator," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 13, no. 1, pp. 17:1–17:38, Feb. 2008.

[2] A. Corradi, L. Leonardi, and F. Zambonelli, "Diffusive load-balancing policies for dynamic applications," *IEEE Concurrency*, vol. 7, pp. 22–31, Jan. 1999.

[3] F. Zambonelli, "How to improve local load balancing policies by distorting load information," in *Proceedings of the Fifth International Conference on High Performance Computing*, ser. HiPC '98, Washington, DC, USA, Dec. 1998, pp. 318–339.

[4] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, pp. 671–680, 1983.

[5] J. A. Stankovic and I. S. Sidhu, "An adaptive bidding algorithm for processes, clusters and distributed groups," in *ICDCS*, 1984, pp. 49–59.

[6] M.-Y. Wu, "On runtime parallel scheduling for processor load balancing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 8, no. 2, pp. 173–186, Feb. 1997.

[7] G. Cybenko, "Dynamic load balancing for distributed memory multiprocessors," *J. Parallel Distrib. Comput.*, vol. 7, no. 2, pp. 279–301, Oct. 1989.

[8] A. Cortés, A. Ripoll, F. Cedó, M. A. Senar, and E. Luque, "An asynchronous and iterative load balancing algorithm for discrete load model," *J. Parallel Distrib. Comput.*, vol. 62, no. 12, pp. 1729–1746, Dec. 2002.

[9] E. Jeannot and F. Vernier, "A practical approach of diffusion load balancing algorithms," in *Proceedings of the 12th international conference on Parallel Processing*, ser. Euro-Par'06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 211–221.

[10] H. Song and K. Choi, "Simulated annealing-based diffusive load balancing on many-core soc," in *Proceedings of the 8th ACM international conference on Autonomic computing*, ser. ICAC '11. New York, NY, USA: ACM, 2011, pp. 187–188.

[11] "Soc designer plus," Jan. 2013. [Online]. Available: http://www.carbondesignsystems.com

 ISBN: 978-1-61208-257-8