LUTMap: A dynamic heuristic application mapping algorithm based on lookup tables

Thomas Canhao Xu and Ville Leppänen

Department of Information Technology, University of Turku, 20014, Turku, Finland
canxu@utu.fi

Abstract. In this paper, we propose and investigate a dynamic heuristic mapping algorithm with lookup table optimizations. Distributed and parallel computing are trends due to the performance requirement of modern applications. Application mapping in a multiprocessor system is therefore critical due to the dynamic and unpredictable nature of the applications. We analyse the communication delay among different tasks in an application. A fundamental algorithm is analysed to optimize the average delay of the mapping region. We discuss and evaluate the effectiveness of the algorithm in terms of average intra-application latency. Results from synthetic applications revealed that average latencies from the mapping regions of the fundamental algorithm have reduced up to 23% compared with the incremental mapping. By noticing the time overhead of the algorithm due to extra number of search spaces, we introduce a mechanism with lookup tables to speed up the process of searching optimized mapping regions. The lookup table is examined with both size and construction time. Experiments shown that the lookup table is small enough to fit into the cache, and the table can be constructed in milliseconds in most practical cases. The results from real applications show that the average execution time of applications of the proposed algorithm has reduced by 15.2% compared with the first fit algorithm.

1 Introduction

Distributed and parallel computing are more and more common nowadays. An important reason behind this trend is applications: distributed and parallel applications used to be widely adopted in high-performance and scientific computing, as well as computer servers. Currently the trend is shifted to daily computing, desktop and even mobile applications utilize the power of distributed and parallel computing extensively. Not only games and real-time audio/video processing, but also signal processing and real-time remote sensing are heavily developed to make use of more processor cores. Commercial desktop and server processors already have more than 20 cores, while it is possible to integrate 10 cores in a mobile processor [17]. The cores are well-utilized by applications: a study shows that even browsers, email clients and video watching applications use as much as 8 cores [11]. It can be predicted that the number of cores in a single chip will

still grow in the near future. Mesh interconnect is proposed for massive multi-core processors to alleviate the communication overhead [4]. Processors based on mesh network are manufactured both for research and commercial use [7].

Applications should be mapped to different nodes in the system in order to execute. *Application mapping* consists of finding a mapping region for a given application with several *tasks*. There are usually several constraints and requirements to meet, for example performance, efficiency, temperature and system congestion. The metrics can be affected seriously with different mappings. For instance to reduce thermal hot-spot, tasks in an application should be mapped dispersedly, however system performance may reduce in that case. On the other hand, a congregated mapping can potentially decrease the communication delay among tasks, while certain part of the system can be congested.

Various mapping algorithms have been discussed and studied by previous researches [14] [18] [22]. For more complex dynamic systems, the mapping algorithm itself should be efficient since finding the optimal solution is usually an NP-hard problem [23] [5]. Because of the computation complexity, researchers have focused on heuristic and stochastic algorithms. Chou et al. introduced an incremental mapping algorithm in [3]. The algorithm selects nodes closest to the master node, and then maps the application to the region. A mapping algorithm designed for big data applications is proposed in [21]. The algorithm is optimized for both cache and memory accesses, as well as intra-application communication. To meet the deadline and energy constraints of real-time applications, [9] and [2] discussed mapping algorithms optimized for such conditions. Instead of contiguous mapping, [6] investigated mixed mapping for mixed-critical concurrent applications, i.e. non-contiguous mapping is applied to increase system throughput while real-time applications are still mapped contiguously for meeting the deadline requirements. A greedy heuristic approximation algorithm is proposed for three-dimensional networks in [23]. Here the authors investigated application mapping in a multi-layer network with limitations of inter-layer connections.

Heuristic algorithms can produce decent mapping results with relatively low time cost, provided that the heuristics are chosen appropriately. On the other hand, multiple results can be evaluated and compared in the algorithm, and quality of the mapping region improves as a result [24]. However the time cost increases linearly as the number of extra searches increases. This can be a problem especially for time critical systems. *Lookup table* has been widely used in computer science to reduce processing time [10]. Applications include arithmetic operations, image processing, cache/memory and even hardware design [8]. A lookup table is a pre-initialized array that replaces the runtime computation with table lookup operations. The additional cost is usually the size and construction time of the table. In this paper, we propose and investigate a novel application mapping algorithm based on lookup table. The algorithm chooses mapping region based on the information stored in the lookup table, and the table is constructed and updated in the background dynamically. We select both synthetic and real applications with different configurations to compare the proposed algorithm with other algorithms.

2 Application Mapping and Communication Delay

The performance of applications is highly affected by communication overhead of different tasks in the application. Figure 1 shows the process of mapping. An application with 5 tasks is being mapped to a 3×3 system in which only 7 nodes are available. Firstly a mapping region is chosen for the given application, e.g. in the figure a 5-node region with cross pattern is selected, and then the tasks are mapped to the region. Obviously the selection of mapping region is crucial in the process. To reduce the possibility of congestion, the region should be as congregate and contiguous as possible. We investigate the metric of *Manhattan Distance* (MD), which represents the number of hop counts between two nodes. Apparently the average pairwise MD of the nodes in the mapping region determines how congregate the region is. In Figure 1, the 5-node region is formed based on a 2×2 square, while assume another 5-node region with a straight continuous line (not shown in the Figure). Both regions are contiguous, however in terms of average pairwise MD, the first region is better. Several problems must be addressed in dynamic systems with multiple concurrent applications: first the system is fragmented by different applications, second it might be unworthy to find the optimal result.

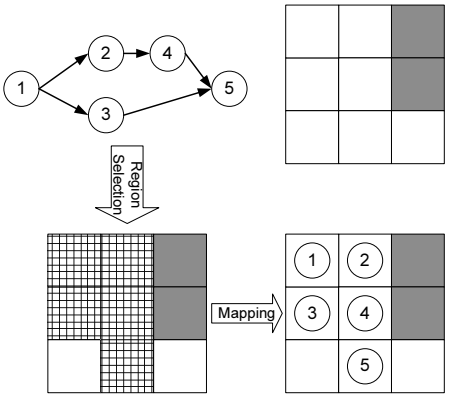

Fig. 1: Mapping a 5-task application to a 3×3 mesh network, grey nodes are already occupied.

3 The LUTMap Algorithm

In this section, we first define a model for the destination platform. We then propose an algorithm to create the lookup tables dynamically to mitigate the computation overhead of calculating the mapping regions.

3.1 Platform Model

Definition 1 *A multiprocessor system consists of a mesh network $P(X, Y)$ of width X, length Y with $X \times Y$ nodes. Each node n_i is denoted by a coordinate (x, y), where $0 \leq x \leq X-1$ and $0 \leq y \leq Y-1$, and $i = y \times X + x$. The Manhattan Distance between n_i and n_j is $MD(n_i, n_j)$.*

Definition 2 *A Task Graph (TG) is a directed acyclic graph, $TG = (N, E)$, where N is the set of nodes and E is the set of directed edges associated with the graph. The amount of traffic (weight of the edge) between nodes n_i and n_j is represented as $w_{i,j}$. $\forall (n_i, n_j) \in E$.*

Definition 3 *An application $A_k(TG_k)$ consists of a list of task nodes N_k, and a list of communication volume between different nodes E_k. To execute the application, it must be mapped to $|N_k|$ nodes.*

Definition 4 *$R_l(A_k)$ is a mapping region in $P(X,Y)$, which consists of a set of $|N_k|$ nodes for $A_k(TG_k)$. Notice that several mapping regions can be evaluated for $A_k(TG_k)$.*

Definition 5 *Average Intra-application Latency ($AIL_{R_l(A_k)}$) is the average latency between internal nodes for an application A_k with a mapping region $R_l(A_k)$. The $AIL_{R_l(A_k)}$ is calculated as:*

$$AIL_{R_l(A_k)} = \frac{\sum_{(n_i,n_j)\in R_l(A_k)} w_{n_i,n_j} \times MD(n_i,n_j)}{|N_k|} \quad (1)$$

Definition 6 *Congregate Degree (CD_{n_i}) is the maximum number of available nodes for n_i in the $x+y+$ (right-up) direction, in a square shape. The CD_{n_i} updates as systems state changes.*

3.2 Region Selection and Mapping

It is obvious that system performance will be affected by different mappings of applications. As an indicator, the AIL shows the average node-node access delay for a mapping decision. This metric can be explained with Figure 2. For example, Application C is mapped to nodes n_2, n_3, n_4 and n_{43}, while Application D is mapped to nodes n_6, n_7, n_{14} and n_{15}. To calculate AIL, the average MD between a node and all other nodes in a mapping region is considered. Here we assume $\forall i,j : (n_i,n_j) \in E$, $w_{i,j} = 1$ for simplicity. Take n_2 in C for instance, $MD(n_2,n_3) = 1$, $MD(n_2,n_4) = 2$ and $MD(n_2,n_{43}) = 6$, hence the average MD for n_2 to other nodes in C is 2.25 ($\frac{9}{4}$). Correspondingly the metric can be calculated for n_3, n_4 and n_{43} as well. At last by multiplying the weight of edges, the average value of them is the AIL_C. Obviously both regions C and D are a possible mapping for a 4-task application, however in terms of AIL, D is preferable compared to C. Lower AIL means lower internal delay of an application, which translates to higher performance and lower power consumption.

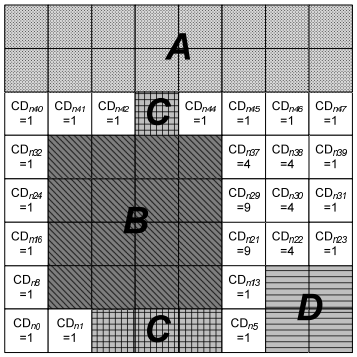

Fig. 2: Example of 4 applications A (16 tasks), B (16 tasks), C (4 tasks) and D (4 tasks) running on an 8×8 mesh.

To determine the optimal (lowest) AIL for an application, all possible permutations can be enumerated and compared, however the computation complexity

can be too high for many systems. Demaine et al. proposed an algorithm that can calculate the optimal value of all pairwise MD of a mesh grid in $\mathcal{O}(n^{7.5})$ [5]. However the algorithm only works in a static empty system. In case of a dynamic system with multiple concurrent applications, the true optimal solution is an $NP\text{-}Hard$ problem [5] [12] [25]. As a compromise of efficiency and computation complexity, several methods are proposed to achieve sub-optimal results with much lower computation cost, such as simulated annealing, linear programming, heuristic and stochastic algorithms. Here we first explore an algorithm based on the proof of [5], that an optimal mapping region is *convex* and the shape should be *near circular*. Take Figure 2 for instance, two applications A and B are both with 16 tasks, however AIL_{R_A} is higher than AIL_{R_B} ($AIL_{R_A} = 3.125$, $AIL_{R_B} = 2.5$). Notice that B is not an optimal result in terms of AIL, [5] has given the optimal result with $AIL = 2.484$. It is noteworthy that a circular shape is usually closer to optimal than a square, nonetheless the computation complexity of generating circular shapes is higher than that of squares. We consider an algorithm based on square shape, the *Congregate Degree* is calculated for all the nodes. Figure 2 shows the CDs of all free nodes.

3.3 Fundamental Algorithm

For a given application A_k, the number of tasks can be equal, larger or smaller than the CD of a node. As a consequence the region, i.e. the free nodes represented by the CD of a node, should be adjusted, or left untouched in case the two numbers are equal. The fundamental algorithm searches for a limited number of mapping regions. R_{max} is defined to control the maximum number of regions in the list of candidate regions (*search space*). Increasing R_{max} takes more time for region evaluation and can possibly generate better results. If $CD_{n_i} > |N_k|$ and the number of tasks can be represented by a square, the smallest square region that is closest to $|N_k|$ is added to the candidate list. Notice that CD_{n_i} represents the highest number of free nodes in a square shape, where smaller squares are evaluated as well in the algorithm. The region is expanded in case the square is smaller than $|N_k|$. The expansion strategy follows the minimum AIL rule, that for all the free nodes, only those closest to the current region are added. Take an 11-task application for instance (Figure 2), n_{21} and n_{29} are favourable due to the CDs, two nodes are expanded for both square regions. At last, the candidate regions in the list are compared with AIL. The algorithm always starts with nodes with largest CD. The extra searches will start from free nodes with smaller CDs than the previous one.

The proposed algorithm is firstly evaluated by using synthetic traces. We compare the fundamental algorithm with a different number of search space ($SS*$), the First Fit algorithm (FF), the incremental mapping algorithm in [3] (INC), a greedy mapping algorithm that always chooses nodes nearest to the centre of the mesh network ($PROX/PRO$), the Nearest Neighbour algorithm (NN) and random mapping algorithm ($RAND/RAN$). We use Task Graph Generator [19] to generate 10,000 applications of 1 to 16 tasks with equal possibility. The applications enter and leave the system with first-in-first-out se-

quence. Provided that the number of free nodes is smaller than the number of tasks of the incoming application, the earliest application will be removed after execution. We investigate systems with different node utilizations. Lower utilized networks have higher number of free nodes and consequently should generate better results. Researchers reported that for modern large-scale systems, the processor utilization can maintain over 80% [13]. The results are presented in Table 1.

Table 1: Results of different mapping algorithms with different Node Utilizations (NU), the unit of time (μs) represents the average time for each mapping decision. Normalized AIL ($NAIL$) is shown for clarity. System configuration: Core i7 920 2.67 GHz, 8GB RAM

NU/0.5	**SS1**	**SS2**	**SS4**	**SS8**	**SS16**	**SS32**	**FF**	**INC**	**PRO**	**NN**	**RAN**
AIL	77.70	75.02	72.60	71.19	70.44	70.14	111.41	75.24	121.31	92.05	186.82
NAIL	110.79	106.96	103.51	101.51	100.44	100.00	158.85	107.28	172.96	131.24	266.37
Time	71.6	97.5	156.5	273.9	519.9	1011.5	44.9	68.1	46.7	52.0	47.1
NU/0.6	**SS1**	**SS2**	**SS4**	**SS8**	**SS16**	**SS32**	**FF**	**INC**	**PRO**	**NN**	**RAN**
AIL	81.00	77.31	73.48	71.68	70.66	70.31	115.68	77.92	133.28	92.62	186.38
NAIL	115.21	109.95	104.51	101.94	100.50	100.00	164.52	110.83	189.55	131.73	265.08
Time	66.0	89.8	140.2	246.6	460.6	886.9	42.6	62.5	44.3	48.7	44.7
NU/0.7	**SS1**	**SS2**	**SS4**	**SS8**	**SS16**	**SS32**	**FF**	**INC**	**PRO**	**NN**	**RAN**
AIL	84.61	79.46	74.89	72.42	71.19	70.66	121.18	82.10	143.31	92.44	185.02
NAIL	119.74	112.45	105.98	102.49	100.75	100.00	171.48	116.19	202.80	130.82	261.83
Time	59.9	81.1	125.0	215.0	398.7	759.5	41.0	57.2	41.3	45.8	42.8
NU/0.8	**SS1**	**SS2**	**SS4**	**SS8**	**SS16**	**SS32**	**FF**	**INC**	**PRO**	**NN**	**RAN**
AIL	90.82	84.48	77.79	74.49	73.00	72.16	125.05	86.55	154.75	93.41	185.37
NAIL	125.85	117.07	107.80	103.23	101.16	100.00	173.29	119.94	214.45	129.45	256.89
Time	54.2	72.0	108.4	182.5	333.0	628.6	39.3	52.2	39.0	42.9	40.6
NU/0.9	**SS1**	**SS2**	**SS4**	**SS8**	**SS16**	**SS32**	**FF**	**INC**	**PRO**	**NN**	**RAN**
AIL	101.20	92.09	83.60	79.24	77.40	75.90	128.04	93.54	164.69	101.68	184.58
NAIL	133.33	121.33	110.14	104.41	101.97	100.00	168.70	123.25	216.99	133.97	243.19
Time	49.3	64.2	94.6	155.9	278.3	523.4	37.6	47.0	37.5	40.6	39.7
NU/1.0	**SS1**	**SS2**	**SS4**	**SS8**	**SS16**	**SS32**	**FF**	**INC**	**PRO**	**NN**	**RAN**
AIL	124.83	116.69	107.32	101.73	99.44	98.96	130.20	106.57	176.84	121.40	183.79
NAIL	126.14	117.92	108.45	102.80	100.49	100.00	131.57	107.69	178.70	122.68	185.73
Time	44.4	56.4	79.8	126.7	221.0	408.1	36.4	42.1	36.0	38.3	40.8

In terms of AIL, the fundamental algorithm achieved improved results with increased number of search space. Higher node utilization leads to higher AIL for all algorithms. $RAND$ provides a baseline for unoptimized mapping, while traditional widely-used algorithms such as FF, NN and $PROX$ did provide better results compared with $RAND$. Unarguably the INC algorithm is superior in all cases than the traditional algorithms. However depending on the system utilization, the results of $SS32$ are 7.28% to 23.25% better compared with INC. We notice that different number of search space did have an impact to the quality of the result. For example, the average AIL for all utilizations improved around 19% by increasing the number of search spaces from 1 to 8. However the improvement is smaller from $SS8$ to $SS32$: the AILs of $SS8$ and $SS16$ under 90% utilization are 4.41% and 1.97% worse than that for $SS32$. The gap is smaller when the utilization is low, and widens gradually as system utilization increases. The time consumed by extra search spaces increases linearly, and in general all algorithms take longer time with lower utilization due to additional free nodes.

3.4 Lookup Table

For a given application, the mapping region should be determined before it can be mapped to the system. Calculating the mapping region usually costs much more time than mapping the application, especially for the proposed algorithm. As aforementioned, the fundamental algorithm sacrifices computation time for better results. The extra time can be a problem for time-critical systems. Therefore we further propose an improved algorithm with lookup table. The table is constructed in the background, and stored in the memory for fast lookup (Figure 3). Here we define the table consisting of all the mapping regions which $1 \leq |N| \leq n_{Available}$. Obviously higher number of available nodes means larger table. For each entry in the table, the fundamental algorithm is carried out with certain number of search spaces. Therefore the table contains the information of optimized mapping regions for all possible sizes.

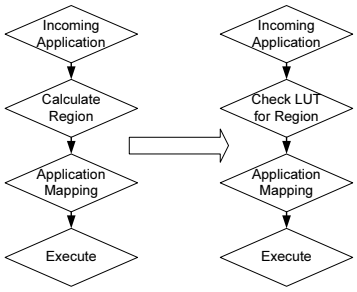

Fig. 3: The process of conventional mapping (left) and mapping based on lookup table (right).

As aforementioned, there are several key issues for implementing lookup tables. On the one hand, the size of the table should be small enough in order to be efficiently stored in the memory or preferably the cache. Otherwise the time overhead of retrieving the table from the disk could be unworthy. On the other hand, the time spent for constructing the table should be relatively low compared with actual application execution, i.e. if it takes too long time, direct computation might be a better solution. We evaluated both issues by using the same application traces as before, the results are illustrated in Figures 4, 5 and 6.

As is shown in Figure 4, the size of the table is relatively small for the given environment. The table becomes larger as system utilization decreases since there are more available nodes for calculation and storage. For example with 90% utilization, the average entries of the table are around 20, and the size of the table is about 1KB. Each entry of the table stores the node information of an optimized mapping region for a certain size, i.e. the table has 10 entries for 10 available nodes, storing information of optimized mapping regions containing 1 to 10 nodes. Notice that even at 50% utilization, the storage overhead of the table is approximately 4KB, small enough

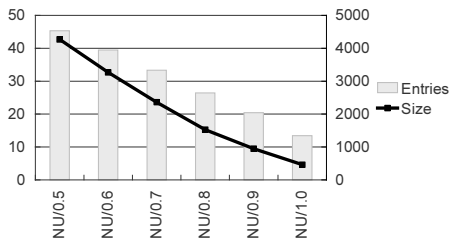

Fig. 4: Average number of entries (left Y-axis) of the lookup table with different Node Utilizations ($NU/*$), and the average size of the lookup table shown as bytes (right Y-axis, assuming 32-bit integer).

to fit into the cache. The size of the table is linearly related with the number of available nodes.

In terms of table construction time, Figure 5 shows that the time depends on both the number of search space and system utilization. For high-utilized systems, the construction time is relatively low due to reduced number of available nodes, e.g. all the table construction times for over 80% utilized systems are below $20ms$ provided that the number of search space is lower than 16. The time increases significantly as system utilization decreases and the number of search space grows due to extra calculation. However as aforementioned the system utilization of large-scale systems is typically over 80%. Moreover the AIL gap of different number of search spaces is not that high for low utilized systems compared with high utilized systems: for instance in Table 1, the gap of AIL for $SS8$ and $SS32$ is 1.51% with 50% utilization, while the gap increases to 4.41% with 90% utilization. This implies that a smaller number of search spaces might be more suitable for low utilized system, while a larger number of search spaces is required for highly utilized system.

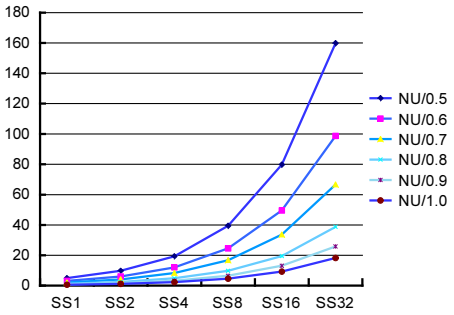

Fig. 5: Average construction time (ms, Y-axis) for lookup tables, in terms of different number of Search Spaces ($SS*$) and Node Utilizations ($NU/*$).

Figure 6 illustrated the time cost of each table construction with 90% utilization and search depth of 32. Notice that the first five data are not shown in the figure (101 to $517ms$ due to the system is empty in the beginning, requiring more computation). Overall most tables are constructed within few milliseconds, e.g. 45% of the lookup tables are calculated under $10ms$, where 75% are finished below $20ms$. The application execution time is usually much longer than the magnitude of millisecond, especially for large-scale multi-task applications. Therefore the time overhead for calculating the lookup table should be acceptable for most cases. Overall the experiments indicate that the lookup table used here is small enough for the cache and the calculation overhead is practical.

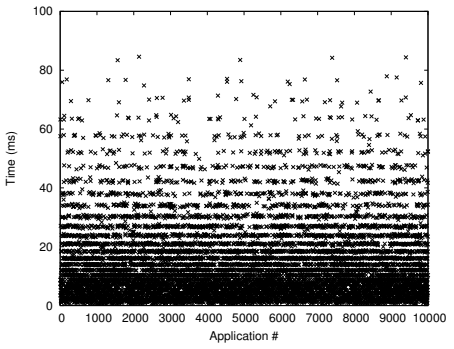

Fig. 6: Detailed time spent for each lookup table construction for 10,000 applications.

Figure 7 illustrated the detailed AIL results for different algorithms mapping 10,000 applications under 90% system utilization. Obviously the proposed algorithm produced relatively good and stable results. We notice NN generated comparable results with $SS1$, while the curve of INC is comparable with $SS2$. However higher number of search spaces provide significantly better results in nearly all cases. The differences among $SS8$, $SS16$ and $SS32$ are noticeable especially for applications number over 7000, indicating that a larger number of search space is still preferable in many cases.

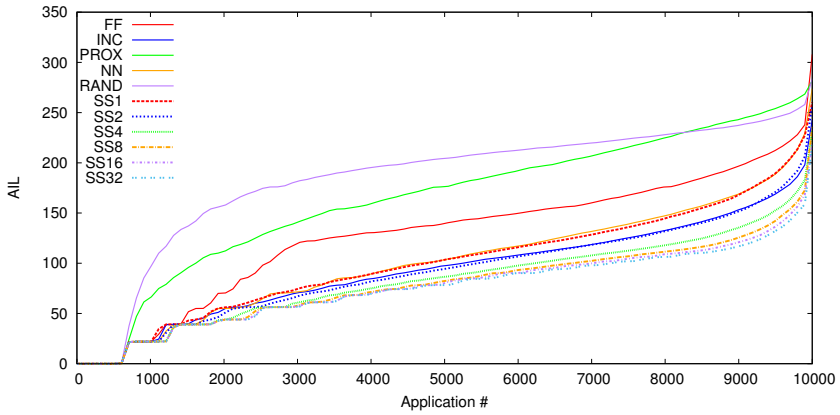

Fig. 7: Comparison of different mapping algorithms with 90% system utilization. The curves show exact sorted AIL values for 10,000 applications.

4 Application Evaluation

In this section, we evaluate the performance of different mapping algorithms with real applications.

4.1 System Configuration

The experimental environment for applications is based on a full system simulator GEMS/Simics [15] [16]. A multiprocessor system with 64 (8×8 mesh) nodes is simulated, where each node is equipped with a Sun UltraSPARC VIII+ core running at 2GHz. Several workloads are selected from [20] and [1], including matrix decomposition ($Cholesky$ and LU), video coding ($H264$) and FFT. All the mapping algorithms are evaluated in the same system state, in which the aforementioned 10,000 applications were executed with the corresponding algorithm with 90% system utilization. We measure performance metrics in terms of application execution time and time spent for the mapping algorithm. Here the proposed algorithm ($LUTMap$ in figures) is set to 32 search spaces, and we only compare results from FF, INC, $PROX$ and $RAND$ for simplicity. The normalized results are illustrated in Figures 8a and 8b.

4.2 Result Analysis

The experimental results from Figure 8a show that, in terms of average application execution time, the proposed algorithm $LUTMap$ is the best compared with other algorithms. We notice that compared with INC and FF, on average the four applications run 10.2% and 15.6% faster respectively. The two other mapping algorithms perform much worse, the $PROX$ is 18.2% slower for these applications compared with $LUTMap$, while not surprisingly $RAND$ is the worst algorithm here. This can be explained from the search depth of the proposed algorithm. While the compactness and continuity of the mapping region are considered in INC, it suffered from limited number of search spaces. The essential idea of INC is to include the nearest nodes that minimizes average latency of the region without a global view. It is noteworthy that in Table 1 the $NAIL$ for $SS32$ has improved 23.3% over INC with 90% utilization, while the improvement is much less for the four applications. We also note that the improvement of execution time depends on the communication graph of applications, albeit not significant here.

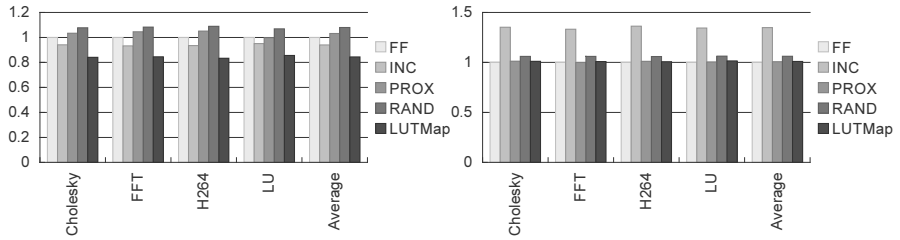

Fig. 8: Normalized application execution time (a) and calculation time of mapping regions (b) for different mapping algorithms.

In terms of computation time of the mapping regions, Figure 8b reveals that the proposed algorithm achieved nearly identical results as FF and $PROX$. This is primarily due to the fact that selecting an optimized mapping region in $LUTMap$ requires only a table lookup operation and the table is small enough to fit into the memory. Therefore the operation can be completed instantly as is in FF and $PROX$. While in $RAND$ an extra operation is needed for computing a random number, and selecting neighbouring nodes costs additional time for generating the mapping region in INC (on average 35% slower than FF).

5 Conclusion

We proposed a dynamic application mapping algorithm in this paper. Parallel and distributed processing are trends for modern applications. Systems integrate more and more processor cores to increase the capability of processing multiple applications. However the mapping of applications in such a system is critical for

performance and efficiency. We explored the intra-application delay of these systems. A fundamental algorithm is proposed to optimize the delay with improved mapping region selection. The algorithm is proved to be effective in terms of average intra-application delay, however the time cost of calculating additional regions can be a problem. We then investigated a scheme based on lookup tables. The tables are computed dynamically based on the status of the system. Our results show that both the overhead of size and computation time of the lookup table were acceptable for most cases. Experiments were conducted based on real applications with a cycle-accurate simulator. It is shown that checking the lookup table is as fast as other simple mapping algorithms, while the execution time of four applications was improved by 15.6% compared with the first fit algorithm.

References

1. Bienia, C., Kumar, S., Singh, J.P., Li, K.: The parsec benchmark suite: Characterization and architectural implications. In: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques. pp. 72–81. PACT '08, ACM, New York, NY, USA (2008)
2. Chen, Y.J., Yang, C.L., Chang, Y.S.: An architectural co-synthesis algorithm for energy-aware network-on-chip design. Journal of Systems Architecture 55(5?6), 299 – 309 (2009)
3. Chou, C.L., Ogras, U., Marculescu, R.: Energy- and performance-aware incremental mapping for networks on chip with multiple voltage levels. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on 27(10), 1866–1879 (Oct 2008)
4. Dally, W., Towles, B.: Principles and Practices of Interconnection Networks. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2003)
5. Demaine, E.D., Fekete, S.P., Rote, G., Schweer, N., Schymura, D., Zelke, M.: Integer point sets minimizing average pairwise distance: What is the optimal shape of a town? Computational Geometry 44(2), 82 – 94 (2011), special issue of selected papers from the 21st Annual Canadian Conference on Computational Geometry
6. Fattah, M., Rahmani, A.M., Xu, T., Kanduri, A., Liljeberg, P., Plosila, J., Tenhunen, H.: Mixed-criticality run-time task mapping for noc-based many-core systems. In: Parallel, Distributed and Network-Based Processing (PDP), 2014 22nd Euromicro International Conference on. pp. 458–465 (Feb 2014)
7. Fleig, T., Mattes, O., Karl, W.: Evaluation of adaptive memory management techniques on the tilera tile-gx platform. In: Architecture of Computing Systems (ARCS), 2014 27th International Conference on. pp. 1–8 (Feb 2014)
8. Ghosh, A., Paul, S., Bhunia, S.: Energy-efficient application mapping in fpga through computation in embedded memory blocks. In: VLSI Design (VLSID), 2012 25th International Conference on. pp. 424–429 (Jan 2012)
9. Hu, J., Marculescu, R.: Energy-aware communication and task scheduling for network-on-chip architectures under real-time constraints. In: Proceedings of the Conference on Design, Automation and Test in Europe - Volume 1. pp. 10234–. DATE '04, IEEE Computer Society, Washington, DC, USA (2004)
10. Hyde, R.: The Art of Assembly Language. No Starch Press, San Francisco, CA, USA, 2nd edn. (2010)

11. LaCouvee, D.: Fact or fiction: Android apps only use one cpu core (December 2015), http://www.androidauthority.com/fact-or-fiction-android-apps-only-use-one-cpu-core-610352/
12. Lei, T., Kumar, S.: A two-step genetic algorithm for mapping task graphs to a network on chip architecture. In: Digital System Design, 2003. Proceedings. Euromicro Symposium on. pp. 180–187 (Sept 2003)
13. Leung, V.J., Sabin, G., Sadayappan, P.: Parallel job scheduling policies to improve fairness: A case study. In: 39th International Conference on Parallel Processing, ICPP Workshops 2010, San Diego, California, USA, 13-16 September 2010. pp. 346–353 (2010)
14. Leutenegger, S.T., Vernon, M.K.: The performance of multiprogrammed multiprocessor scheduling algorithms. SIGMETRICS Perform. Eval. Rev. 18(1), 226–236 (Apr 1990)
15. Magnusson, P., Christensson, M., Eskilson, J., Forsgren, D., Hallberg, G., Hogberg, J., Larsson, F., Moestedt, A., Werner, B.: Simics: A full system simulation platform. Computer 35(2), 50–58 (2002)
16. Martin, M.M., Sorin, D.J., Beckmann, B.M., Marty, M.R., Xu, M., Alameldeen, A.R., Moore, K.E., Hill, M.D., Wood, D.A.: Multifacet's general execution-driven multiprocessor simulator (gems) toolset. Computer Architecture News (September 2005)
17. Mediatek: Helio x20 (December 2015), http://mediatek-helio.com/x20/
18. de Souza Carvalho, E., Calazans, N., Moraes, F.: Dynamic task mapping for mpsocs. Design Test of Computers, IEEE 27(5), 26–35 (Sept 2010)
19. TGG: Task graph generator (July 2014), http://taskgraphgen.sourceforge.net/
20. Woo, S.C., Ohara, M., Torrie, E., Singh, J.P., Gupta, A.: The splash-2 programs: Characterization and methodological considerations. In: Proceedings of the 22nd International Symposium on Computer Architecture. pp. 24–36 (June 1995)
21. Xu, T., Toivonen, J., Pahikkala, T., Leppanen, V.: Bdmap: A heuristic application mapping algorithm for the big data era. In: Ubiquitous Intelligence and Computing, 2014 IEEE 11th Intl Conf on and IEEE 11th Intl Conf on and Autonomic and Trusted Computing, and IEEE 14th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UTC-ATC-ScalCom). pp. 821–828 (Dec 2014)
22. Xu, T.C., Leppänen, V.: Algorithms and Architectures for Parallel Processing: 15th International Conference, ICA3PP 2015, Zhangjiajie, China, November 18-20, 2015, Proceedings, Part I, chap. DBFS: Dual Best-First Search Mapping Algorithm for Shared-Cache Multicore Processors, pp. 185–198. Springer International Publishing, Cham (2015)
23. Xu, T.C., Liljeberg, P., Plosila, J., Tenhunen, H.: Exploration of heuristic scheduling algorithms for 3d multicore processors. In: Proceedings of the 15th International Workshop on Software and Compilers for Embedded Systems. pp. 22–31. SCOPES '12, ACM, New York, NY, USA (2012)
24. Xu, T., Leppnen, V.: Cache- and communication-aware application mapping for shared-cache multicore processors. In: Pinho, L.M.P., Karl, W., Cohen, A., Brinkschulte, U. (eds.) Architecture of Computing Systems ? ARCS 2015, Lecture Notes in Computer Science, vol. 9017, pp. 55–67. Springer International Publishing (2015)
25. Xu, T., Liljeberg, P., Tenhunen, H.: A minimal average accessing time scheduler for multicore processors. In: Xiang, Y., Cuzzocrea, A., Hobbs, M., Zhou, W. (eds.) Algorithms and Architectures for Parallel Processing, Lecture Notes in Computer Science, vol. 7017, pp. 287–299. Springer Berlin Heidelberg (2011)