A framework for design space exploration for HPC architectures

Authors Name/s per 1st Affiliation (Author)
line 1 (of Affiliation): dept. name of organization
line 2: name of organization, acronyms acceptable
line 3: City, Country
line 4: Email: name@xyz.com

Authors Name/s per 2nd Affiliation (Author)
line 1 (of Affiliation): dept. name of organization
line 2: name of organization, acronyms acceptable
line 3: City, Country
line 4: Email: name@xyz.com

***Abstract*—In this article we provide a framework for exploiting parallelism onto heterogeneous HPC architectures. Given a HPC cluster with varying compute units, communication constraints and topology, our framework can be utilized for partitioning applications exhibiting task and data parallelism resulting in increased throughput. Our framework can also be used by designers at an early design stage to explore the type of compute units needed and the topology that would be suited for a given application, thereby reducing costs and power requirements. Moreover, software programmers and compiler writers can utilize our framework to gauge the potential parallelism in their programs. The challenge lies in the fact that heterogeneous compute clusters consist of processing elements exhibiting different compute speeds, vector lengths, and communication bandwidths, which all need to be considered when partitioning the application and associated data. We tackle this problem using a staged graph partitioning framework. Our experiments show an order of magnitude speedup for applications. Furthermore our framework finishes within seconds even when simulating 100's of processing elements, which makes our architecture suitable for exploring parallelism potential at compile time.**

***Keywords*-Graph partitioning, vectorization, data parallelism, heterogeneous architectures, clusters.**

```
//Task and data-parallel
for (int i=0;i<M; ++i){
 for (int j=0;j<N; ++j){
  1: A[i][j] = ((((i)*((j)+2.0))+2.0)/(N))
  2: B[i][j] = ((((i)*((j)+3.0))+3.0)/(N))
 }
}
for (int k=0;k<TSTEPS;++k){
 for (int i=1; i<M; ++i)
  for (int j=1; j<N; ++j)
   3: B[i][j] = 0.2*(A[i][j]+A[i][j-1]
              +A[i][j+1]+A[i-1][j])

 //Data-parallel
 for (int i=1; i<M; ++i)
  for (int j=1; j<N; ++j)
   4: A[i][j] = B[i][j]
}
```

Figure 1. Example 2-dimensional Jacobi application

I. Introduction and motivating example

Today's HPC clusters consists of a large number of heterogeneous processing elements such as CPUs, GPUs, DSPs, FPGAs, etc. Given an application exhibiting potential for parallelism the question remains: how does one determine the type of architecture best suited to extract this parallelism? Or given an architecture, how does one determine how to exploit the potential parallelism in the applications.

Consider the code snippet in Figure 1 that carries out the main stencil computation using the Jacobi algorithm. Jacobi is an important stencil computation, which is used for solving large systems of linear equations, especially for heat transfer problems and numerical fluid mechanics. We have chosen this as our motivating example, because there have been attempts to parallelize Jacobi using MPI [1] and CUDA [2] with success, making it an important problem to solve on a heterogeneous compute cluster mixing MPI and CUDA programming techniques and with the potential of making it faster still. The CUDA programming techniques exploit the *Single Instruction Multiple Data* (SIMD) potential in the Jacobi algorithm [2] by modeling parallelism as vector computations suitable for a GPU. The MPI approach on the other hand exploits *Multiple Instruction Multiple Data* (MIMD) potential by modeling the parallelism across different CPU nodes. Both techniques result in 3-4 times speedup compared to single CPU implementations. The challenge when *re-designing* and *tuning* such parallel applications to exploit vector units *or* MPI alone is well documented [1], [2]. The complexity of re-designing for a mixture of two grows exponentially. The growth in complexity of design space is due to a number of factors, some of which we enumerate below:

- The vector lengths of the underlying processing elements differ. Intel processors have 256 bit vector instructions, while the GPU range varies.
- The actual data-type results in different utilization of vector units. For example, a `double` type requires twice as many vector registers to carry out processing as compared to an `int` type.
- The size of the vector length and the number of vector

units required needs to be determined: simply dividing the data-parallel vector units onto the largest available vector processing elements does not necessarily result in good application throughput or latency. In an ideal scenario for very large vector computations, the vector units can be utilized completely and the rest of the data-parallelism can be exploited in parallel on a CPU unit iteratively in a loop.
- The bottleneck of the communication fabric plays an important role in the partitioning problem. Note that in a heterogeneous compute cluster the communication latencies and bandwidths themselves vary.
- Allocation of data-stores being utilized by the different processing elements needs to be handled.
- Applications written in different ways result in different parallelism potential.
- Finally, the scheduling problem is known to be NP-hard [3]. Thus, we need a good heuristic solution, which finishes quickly and gives good results.

There are a number of other applications where parallelism plays and important role. For example, binomial option pricing, k-means calculations, Gauss-Seidel stencil computations, etc, are well suited to be optimized across heterogeneous HPC architectures. In general we have found it is much more essential to exploit data-parallelism as compared to task-parallelism to achieve speedups. But, exploiting both types of parallelism is essential in the general case.

In this paper we propose a framework, which can be used by topology designers, and application writers to quickly carry out a design space exploration to determine the type of underlying topology best suited for a given application. Moreover, compiler writers can also use our framework when vectorizing to partition large vector units onto parallel GPU/CPU units.

II. Related work

A significant amount of research literature exists for extracting parallelism from programs [4], [5], [6], [7], [8]. The polyhedral optimization model [4] concentrates on extracting parallelism from loops, which is a form of data-parallelism. The polyhedral optimization community has concentrated on optimizing for CPUs and GPUs separately, but to our knowledge has not explored the combination of the two. Carpenter et.al [8] have again explored ideas for partitioning onto heterogeneous architectures, but at a much smaller scale and again ignoring data-allocation costs and depending upon the polyhedral model for vectorization. The StreamIt [9] community has also explored parallelization techniques, but they have only targeted homogeneous RAW [10] architecture. There are the classical algorithms such as critical path scheduling [11] and list scheduling [12], which have been used for scheduling task parallel process onto homogeneous architectures. Declustering [7], is another technique, which misses the opportunity to mix SIMD and task-parallel optimizations together.

Cluster based partitioning techniques [13], [14], [15] only consider independent tasks without communication. The proposed heuristics for partitioning data-parallel applications onto clusters [16], [17] do not consider vectorization potential available on the compute clusters and only concentrate on partitioning task parallel processes.

III. Preliminaries

First of all, in this section, we present a formal description of the problem along with the notations used.

A. Notations

We refer to our application graph, as a *Static Task Graph* (STG) defined formally as: $G_t(V_t, E_t)$, where V_t is the set of all tasks in the application graph and E_t represents the communication between these tasks. The system resources are represented by a weighted undirected graph $G_r(V_r, E_r)$ where V_r represents a set of processing elements which can have different processing capabilities. Each vertex in the task graph, $t_i \in V_t$ is referred to as tasks and each vertex in the resource graph, $r_i \in V_r$ is referred to as processing elements (PEs). We use N_T to denote the total number of tasks in the task graph and N_R to denote the total number of processing elements. By our definition of previous notations, it follows that $N_T = |V_t|$ and $N_R = |V_r|$.

B. Problem Definition

Given a graph $G_t(V_t, E_t)$, each vertex in the task graph, $t_i \in V_t$ has a set of associated requirements represented by T^i_{cj} where $j = 0...n$ with n being the number of requirements for the task graph. These requirements represent the computational requirement of the kernel. Consider the task graph in Figure 2 for the running Jacobi example. The `boxed` statements give the number of instructions and the vector count requirements. The communication edges ($(t_i, t_j) \in E_t$) between data-stores (`boxed` statements without the constraints in Figure 2) and the execution statements represent the amount of data that needs to be transfered from the store and its utilization at the statement level. The task graphs are generated directly from the program by our compiler. More information about the generation of the task graphs is provided later in Section IV-A.

Similarly we have a resource graph $G_r(V_r, E_r)$ where each vertex denotes a processing element. It consists of a set of vertices $V_r = \{r_1, r_2, ..., r_n\}$ and a set of edges $E_r = \{(r_i, r_j) | r_i, r_j \in V_r\}$. Each processing element $r_i \in V_r$ has a set of constraints, that represent the computational capabilities of the processing elements, represented by R^i_{cj} where $j = 0...n$ with n being the number of constratins for the resource graph. For some resource graph as shown in Figure **??**, which has x nodes and 2 constraints. Constraint 1 represents the frequency of the PE or how many

scalar instructions the PE can perform in one second (the MIPS count). Constraint 2 denotes the maximum number of parallel vector operations it can perform (the vector length). Each edge (r_i, r_j) represents the latency/bandwidth between two PEs r_i and r_j.

The problem at hand is to effectively map said task graph G_t onto given resource graph G_r. This problem is reminiscent of a problem of bin packing which is known to be NP-Hard [3]. This immediately implicates that we have to look for a heuristic solutions.

To decide if a mapping is effective enough or not, we need a heuristic that defines how good a mapping of tasks onto resources is. This is denoted by our cost function

We are looking for a non-injective and non-surjective mapping.

IV. Our framework

In this section we describe our heuristic algorithm. There are three important concepts that need description. First, we describe how we extract fine grained parallelism from the application into the task graph. Next, we describe how the topology clusters are formed from the resource graphs accounting for communication and heterogeneity of the topology. Finally, we describe how the mapping is performed.

A. Generating the task graph

The task graph is built from the application. The compiler extracts task and data parallelism form the application for partitioning the application onto the architecture. The compiler looks at every statement in the program to form the task graph. The task graph is formed in the following manner:

- The assembly instruction count for every statement is obtained first. Currently, we look at the LLVM (Low level virtual machine) [18] instruction count. For example, the LLVM code generated for the assignment statement `4` in Figure 1 is shown in Figure 3. The LLVM instruction count gives an approximation of the instruction count of the underlying hardware, while remaining independent of the hardware itself.
- Every loop is fissed thereby forming multiple statements. The intuition behind fissing the loops is two fold: (a) task parallelism can be exploited by running independent loop statements separately on different machines (see Figure 2) and (b) the graph partitioner would give us feedback on the vector size of the loop, which can then be fused back if allocated to the same resource.
- The vector counts for each statement is determined using dependence analysis and using the polyhedral model [4]. The largest vector size requirement is given as the second constraint in the task graph.
- Finally, the polyhedral model is also used to find the iteration count of the statements and to determine the total amount of data (in bits) required to process by that statement. For example, statement `4` (the last `boxed` statement in Figure 2) requires `7.8085KB`, while the first two statements `1` and `2` require 64-bits more due to the difference in the loop iteration count (see Figure 1).
- The edges in Figure 2 with 0 weights are dependence arcs. For example, in the Jacobi example, statements `1` and `2` can be carried out in parallel, while statements `3` and `4` have a dependence on these two statements and hence, cannot be split.

1) Tiling vectors: As we can see from Figure 2, three of the statements in Jacobi example require almost a million vector instructions to be carried out in parallel. The vector counts can increase quite quickly for large examples (notice that the current 1 million vector count is just for a 1000 $\times$ 1000 Jacobi matrix). In order to properly utilize the underlying vector hardware these vector lengths need to be split into smaller vector lengths. The resultant vector lengths depend upon the underlying hardware. Splitting the vector constraints is termed tiling in the compiler community. There are many ways to tile a vector. For example, given a single processing element with a small vector size: a vector might be tiled to fit the underlying hardware vector size and then run in a loop (an approach taken by the gcc compiler). If a number of processing elements with differing vector sizes are available as is the case of HPC architectures determining the optimal vector tiles is a challenge and can be solved with *Simulated annealing* (SA) [15] or meta *Genetic algorithm* (GA) [15] heuristics. In this article we do not solve the problem of determining the tile sizes, instead our framework allows the application designers to plug and play with different vector tiles in-order to determine the tile size that suits their architecture. We randomly generate different tile sizes for experimentation.

An example tiling with 250K separate, 2 $\times$ 2 tiles for statement `1` is shown in Figure 2 with its resultant LLVM code. It is important to mention that vectors are only single dimensional. In Figure 2, we are type-casting a 2D matrix of size 2 $\times$ 2 into a single dimensional 4 element vector. Such vectorization is also called loop-collapsed vectorization, i.e., instead of just vectorizing the inner most loop in Figure 1 statement `1`, we are collapsing the outer and inner loop into a single vector instruction to improve performance. Such loop collapse is not necessary and can be considered a super optimization. But, our framework allows the application designers to explore such possibilities.

B. Generating the resource graph

The resource graph represents the cluster of compute nodes on which the task graph will be executed. A sample resource graph is shown in fig 4 at level 0. The resource graph that is shown is truly hetrogeneous in both computations and communication. The resource graph is described below:

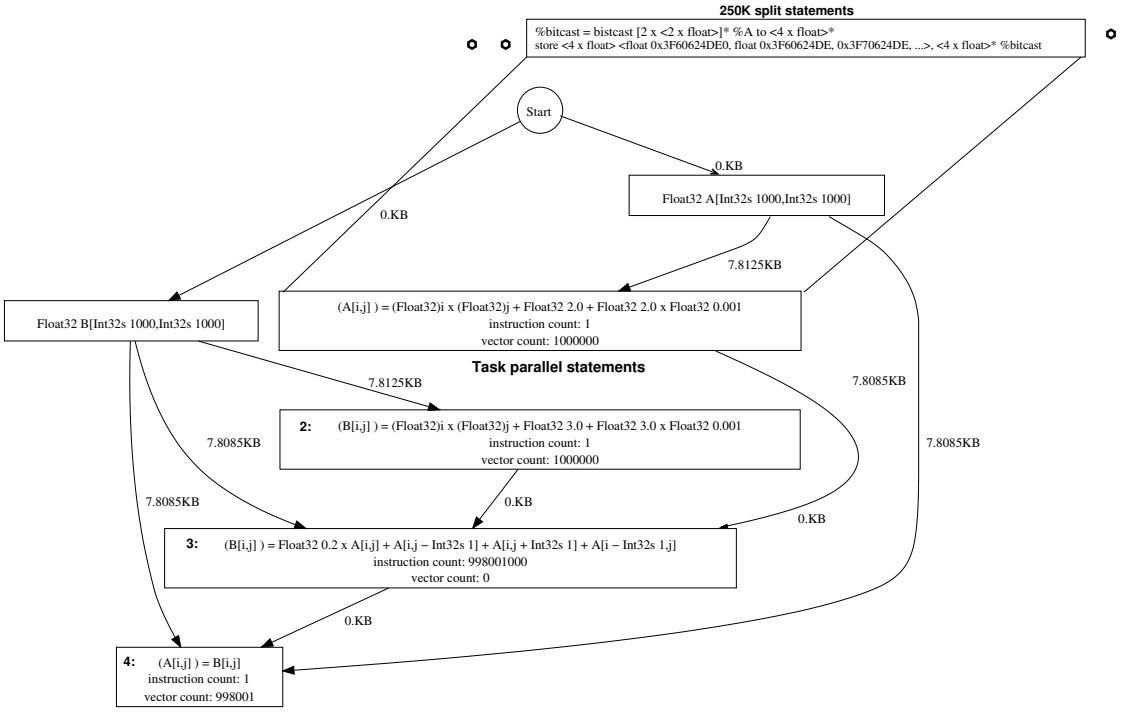

Figure 2. The task graph for the Jacobi example

```
% scevgep = getelementptr [1000 x <1000 x float>]* %A, i64 0, i64 %3
% scevgep10 = bitcast <1000 x float>* %scevgep to i8*
% uglygep = getelementptr i8* %scevgep10, i64 4
% scevgep11 = getelementptr [1000 x <1000 x float>]* %B, i64 0, i64 %3
% scevgep1112 = bitcast <1000 x float>* %scevgep11 to i8*
% uglygep13 = getelementptr i8* %scevgep1112, i64 4
call void @llvm.memcpy.p0i8.p0i8.i64(i8* %uglygep,i8* %uglygep13,i64 3996,i32 4,i1 false)
```

Figure 3. LLVM code for assignment statement 4 from Figure 1

- The compute nodes are assumed to belong under two categories of processing units, mainly CPUs and GPUs. The scalar instructions of a Processing Element (**PE**) that represents a CPU is much higher that that of a GPU. At the same time parallel vector operations that a **PE** representing a GPU is much higher compared to that of a CPU. In Fig 4 R^1 is a 'CPU' and R^5 is a 'GPU'.
- In the resource graph shown in Fig 4 at level 0, the interconnect between these PEs are considered to be a two dimensional mesh. In this topology the PEs are connected in a grid with individual communication links between them. The bandwidth of these links are considered to be non uniform. Similarly, it is possible to represent various resource graphs with different topologies.

Mapping the task graph on to such a hetrogeneous is NP hard. The problem mainly lies in the size of the resouce graph and the sheer number of ways in which the task graph can be mapped on to the resource graph. In our approach we tackle the problem in several phases. The various stages that are performed is shown here:

- Firstly, we form virtual representations of the resource graph by clustering the nodes. These virtual nodes are formed by min cutting communication volume and load balancing the virtual nodes capabilities. The formation of these virtual nodes creates homogeneous partitions from hetrogeneous PEs.
- Secondly, instead of doing this in a single step, we construct this in several level by clustering half the nodes from the previous level. We end up with a structure consisting of several levels, where the $Number\ of\ levels = log_2(Number\ of\ Nodes)$.
- The communication bandwidth between the clustered nodes is determined by,
  $\sum(\min forsamedestn(\max bwR^i, R^j)))$
  The max bandiwdths between any R^i is determined by floyd warshall algorithm.
- The capabity of each of the PEs that are clustered together are aggregated to form the larger virtual node.
- In each level PEs with high communication bandwidth and balanced capabilities are clustered together. In doing this in a bottom up approach i.e. clustering instead of partitioning, we allow the nodes to form without ignoring communication links between the nodes. This also avoids the formation of dangling nodes in the graph.

In fig 4 we show our clustering approach on a 4x2 mesh. At each level suitable nodes are clustered together to form a larger node. We see that the PE R^3 and PE R^8 are combined together when moving from level 0 to level 1. Overall, we reach three levels and at the top most level all the nodes from a single virtual node.

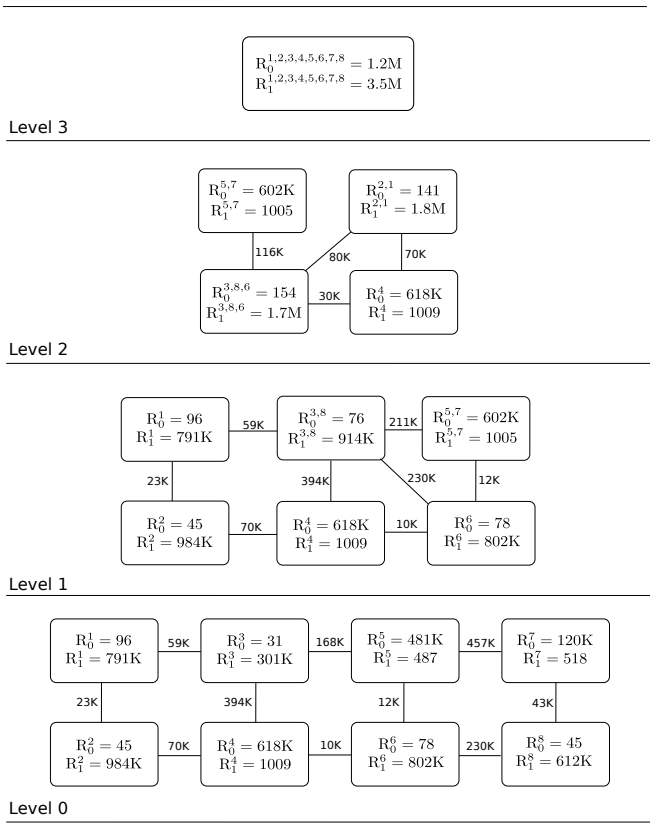

Figure 4. Clustering of a resource graph

C. Application Partitioning

To partition the application we start with the top most level in the resource graph and divide it according to the required virtual nodes at that level. We then move on to the next level and partition the application on to the virtual nodes that the current node belongs to. In this way we follow the structure of the resource graph until we reach level, by which reach the actual mapping of the task graph on to the resource graph.

Doing it in this top down manner carries two important advantages. Firstly, the complexity involved is much less compared to doing it in a single instance and thus preventing an explosive state space search. This makes our heuristic complete much faster than doing it in a brute force manner. Secondly, by dividing the mapping on to several levels we can achieve much better load balancing by considering only fewer nodes to map than if we were to do it directly on to the resource graph.

V. Implementation

We follow a bottom up approach when clustering onto the virtual nodes from the resource graph and top down approach when In order to implement our heuristic we use METIS [] graph partitioning tool.

bullshit about metis.

The resource graph is generated by assuming the capabilities lie in a fixed range. Using this we define a range from which we assign the respective PE's capabilities. In generating them synthetically we avoid being biased by a single architecture and we can evaluate our system for different characteristics. The interconnect is considered to be two dimensional mesh with varying bandwidths. Thus, making both the computaions and communication be truly hetrogeneous.

Once we have the resource graph, we perform the clustering as describe in section IV-B. We use metis to do the clustering of the nodes based on communication volume min cut and load balance constraints. The entire process is described below:

- The generated resource is graph is represented in the Metis graph format. We represent the PEs capabilities as constraints of the nodes and the link's bandiwdths as communication volume of the edges.
- We then construct our clustered structre by halving the nodes at each level. Metis partitions the graph by load balancing the constraints and min cut communication. So the number of partitions that it might provide might be less than that being requested.
- When constructing the new virtual node, the constraints of the previous are aggregated and the communication is calculated by summing the minimum bandwidths of links connecting the nodes present in the different partitions.
- We repeat the clustering for each level, until we reach a stage in which all the PEs form a single partition.

In partitioning the task graph, we need to balance the constraints on to the available partitions. Metis offers the ability to load balance multiple constraints on to different partitions based on the metric 'tp weight'. We calculate the ratios between the capabilities of different partitions and represent as this metric in order to load balance on to the available partitions. The steps through which task graph follows are,

- We start at the top most level in the resource graph and request for the number of partitions equal to that of the child nodes at the next level.
- The capabilities of the resource graph are represented as ratios for each of the constraint as the 'tp weight' metric when partitioning the task graph using Metis.
- As we go down on to each level we partition the task graph into smaller fine grained parts until we reach the lowest level.
- Finally, the entire task graph is partitioned into sub graphs where each of them are mapped onto a specific PE in the resource graph.

A. Load Balancing Multiple Constraints

Metis load balances constraints by minimizing the imbalance with that of a ideal value. It tries to acheive this by modifying the partitioning such that each consrtaint lies within certain imbalance from the ideal value. Since reaching a 'optimal' solution is difficult here, it matches each of the constraint in order until it finds a solution where all the constraints are matched within a certain tolerance level. Unfortunately, this causes the contraint to be matched with more and more imbalance as we move further.

In order to handle this, we chose an addition to our heuristic that checks whether we achieve the least imbalance in compute power in our partitions. This is done by clustering the nodes at each level based on all the possible permutations of the resource graph. After each clustering for a given permutation the imbalance in compute power is calculated as,

$$\begin{gathered} desired_compute_power = total_compute_power/no_of_nodes; \\ foreachpartition, for 1 -> n \\ cp_n = sum((computepowerofnode(R*R..Rn)) - desired_compute_p \\ chosenpartition = min(cp_n) \end{gathered} \tag{1}$$

We then pick the partition that was yielded with the minimum deviation in the compute power.

VI. Experiments and results

We show the speedup obtained by our graph partitioning technique compared to a single PE allocation. Next, we compare our algorithm against two well known heuristic techniques Cross-Entropy [16] and Simulated annealing [19]. We chose these techniques, because they have already been successfully used for partitioning graphs onto heterogeneous architectures [16]. Moreover, these two are the only techniques that perform reduced search space exploration and are able to produce decent results when auto-tuning compilers.

VII. Conclusion

The conclusion goes here. this is more of the conclusion

References

[1] D. an Mey, "MPI Case Study, Jacobi-Solver," in *Parallel Programming in Computational Engineering and Science (PPCES)*, 26 March 2010.

[2] Zhang, Zhihui and Miao, Qinghai and Wang, Ying, "CUDA-Based Jacobi's Iterative Method," in *Proceedings of the 2009 International Forum on Computer Science-Technology and Applications - Volume 01*, ser. IFCSTA '09. IEEE Computer Society, 2009, pp. 259–262. [Online]. Available: http://dx.doi.org/10.1109/IFCSTA.2009.68

[3] V. Sarkar, "Partitioning and scheduling parallel algorithms for execution on multiprocessors," Ph.D. dissertation, Stanford university, 1989.

[4] M. Griebl, C. Lengauer, and S. Wetzel, "Code Generation in the Polytope Model," in *In IEEE PACT*. IEEE Computer Society Press, 1998, pp. 106–111.

[5] J. Donald and M. Martonosi, "An Efficient, Practical Parallelization Methodology for Multicore Architecture Simulation," *IEEE Computer. Architecture. Letter.*, vol. 5, pp. 14–, July 2006.

[6] M. I. Gordon, W. Thies, and S. Amarasinghe, "Exploiting coarse-grained task, data, and pipeline parallelism in stream programs," *SIGOPS Oper. Syst. Rev.*, vol. 40, pp. 151–162, October 2006. [Online]. Available: http://doi.acm.org/10.1145/1168917.1168877

[7] G. C. Sih and E. A. Lee, "Declustering: A new multiprocessor scheduling technique," *IEEE Trans. Parallel Distrib. Syst.*, vol. 4, pp. 625–637, June 1993. [Online]. Available: http://portal.acm.org/citation.cfm?id=628910.629186

[8] P. M. Carpenter, A. Ramirez, and E. Ayguade, "Mapping stream programs onto heterogeneous multiprocessor systems," in *Proceedings of the 2009 international conference on Compilers, architecture, and synthesis for embedded systems*, ser. CASES '09. New York, NY, USA: ACM, 2009, pp. 57–66. [Online]. Available: http://doi.acm.org/10.1145/1629395.1629406

[9] W. Thies, M. Karczmarek, and S. P. Amarasinghe, "StreamIt: A Language for Streaming Applications," in *CC '02: 11th International Conference on Compiler Construction*, London, UK, 2002, pp. 179–196.

[10] E. Waingold, M. Taylor, V. Sarkar, V. Lee, W. Lee, J. Kim, M. Frank, P. Finch, S. Devabhaktumi, R. Barua, J. Babb, S. Amarsinghe, and A. Agarwal, "Baring it all to software: The raw machine," MIT, Cambridge, MA, USA, Tech. Rep., 1997.

[11] W. Kohler, "A Preliminary Evaluation of the Critical Path Method for Scheduling Tasks on Multiprocessor Systems," *Computers, IEEE Transactions on*, vol. C-24, no. 12, pp. 1235 – 1238, dec. 1975.

[12] T. L. Adam, K. M. Chandy, and J. R. Dickson, "A comparison of list schedules for parallel processing systems," *Commun. ACM*, vol. 17, no. 12, pp. 685–690, Dec. 1974. [Online]. Available: http://doi.acm.org/10.1145/361604.361619

[13] M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, and R. F. Freund, "Dynamic mapping of a class of independent tasks onto heterogeneous computing systems," *J. Parallel Distrib. Comput.*, vol. 59, no. 2, pp. 107–131, Nov. 1999. [Online]. Available: http://dx.doi.org/10.1006/jpdc.1999.1581

[14] A. Doğan and F. özgüner, "Genetic algorithm based scheduling of meta-tasks with stochastic execution times in heterogeneous computing systemst1," *Cluster Computing*, vol. 7, no. 2, pp. 177–190, Apr. 2004. [Online]. Available: http://dx.doi.org/10.1023/B:CLUS.0000018566.13071.cb

[15] T. D. Braun, H. J. Siegel, N. Beck, L. L. Bölöni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, B. Yao, D. Hensgen, and R. F. Freund, "A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems," *J. Parallel Distrib. Comput.*, vol. 61, no. 6, pp. 810–837, Jun. 2001. [Online]. Available: http://dx.doi.org/10.1006/jpdc.2000.1714

[16] S. Sanyal and S. Das, "Match : Mapping data-parallel tasks on a heterogeneous computing platform using the cross-entropy heuristic," in *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, april 2005, p. 64b.

[17] Kumar, S. and Jantsch, A. and Soininen, J.-P. and Forsell, M. and Millberg, M. and Oberg, J. and Tiensyrja, K. and Hemani, A., "A network on chip architecture and design methodology," in *VLSI, 2002. Proceedings. IEEE Computer Society Annual Symposium on*, 2002, pp. 105 –112.

[18] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[19] H. Orsila, T. Kangas, E. Salminen, and T. Hamalainen, "Parameterizing simulated annealing for distributing task graphs on multiprocessor socs," in *System-on-Chip, 2006. International Symposium on*, nov. 2006, pp. 1 –4.