Automatic Locality Exploitation in the Codelet Model

Chen Chen†, Yao Wu†, Joshua Suetterlein†, Long Zheng‡, Minyi Guo‡ and Guang R. Gao†
† *University of Delaware*
Newark, DE, 19716, USA
‡ *Shanghai Jiao Tong University*
Shanghai, 200240, China
Email: chenchen@capsl.udel.edu

***Abstract*—**
State-of-the-art codelet scheduling focuses on dynamic workload balance of codelets (similar to tasks). While this approach may achieve reasonable performance since computation resources are fully utilized, it may not attain optimal energy savings. In this paper, targeting at IBM Cyclops64 – a many-core system, we propose a novel polynomial time algorithm that finds out the optimal codelet scheduling in terms of maximum locality and minimum global memory accesses. Our algorithm leverages static information regarding locality among codelets to achieve better performance and energy efficiency. By using local buffers to pass data produced in one codelet to another, global memory accesses can be greatly reduced. The experimental results on our developed IBM Cyclops-64 emulator show that the codelet scheduling of our algorithm removes up to 59.7% of global memory accesses, achieves up to 68.1% of performance improvement, and reduces up to 40.7% of energy consumption comparing to the state-of-the-art codelet scheduling.

***Keywords*-scheduling; locality; codelet; fine-grain; execution model;**

I. Introduction

To continue to reach new levels of performance, HPC systems are growing extremely large and cumbersome. Current means for effectively utilizing these systems are quickly becoming antiquated. For this reason some are seeking alternate execution models parting from the unsatisfying MPI and OpenMP models which have dominated today's parallel paradigm. One such effort is the Codelet model which aims at providing scalable fine-grained execution for the upcoming exa-scale era. The Codelet model finds its inspiration in the dataflow and its descendants such as the hybrid dataflow/Von Neumann EARTH execution model[1]. We are hopeful that representing programs in there most basic dependencies will enable intelligent (codelet based) runtimes to adequately schedule and allocate resources in the ever growing systems of the future.

Several new and emerging architectures share a similar design trend, forgoing traditional caches for several levels of globally shared memory used for data transformations among cores. Each core has a local storage that can be accessed with lower latency and power consumption rather than accessing shared memory. This local storage can be used to buffer data which will be used in immediate future replacing caches. Some examples of such architecture designs are IBM CELL Broadband Engine[2], IBM Cyclops64 [3], and Intel UHPC Straw-man architecture [4].

Locality exploitation is very important for the aforementioned architectures since: (1) there is no hardware cache to automatically exploit locality, and (2) locality exploitation improves both performance and energy efficiency. Unfortunately, current codelet scheduling approaches mainly focus on balancing workloads and reducing scheduling overhead. Therefore, programmers have to manually exploit locality, which is detrimental to the programming productivity.

In this paper, we study the problem of automatically exploiting locality among codelets. Given both a static codelet graph and information regarding locality, our study targets finding the best parallel schedule capable of maximizing locality. The major contributions of this paper are as follows:

- We propose and study the codelet scheduling problem for maximum locality exploitation (Best Scheduling Problem). Through study of this problem we discover a solution which provides better performance and energy efficiency. To the best of our knowledge, we are the first to study this problem in codelet scheduling.
- We propose a polynomial-time algorithm that provides an optimal solution to the Best Scheduling Problem when there are enough computation resources. In addition, the algorithm provides the minimal computation resource required.
- For comparison, we studied two other algorithms, each of which has different trade-offs in algorithmic complexity, locality exploitation, program execution time, and energy efficiency. We analyze and show suitable algorithms which should be used in different circumstances.

We developed an emulation platform of the IBM Cyclops-64 many-core architecture to study various algorithms to solve the Best Scheduling Problem. Within the platform, we studied the three algorithms on various applications including matrix multiply, merge sort, and random generated codelet graphs with reasonable assumptions. The experimental results show that we can reduce up to 59.7% of global memory access via the locality exploitation. We also ob-

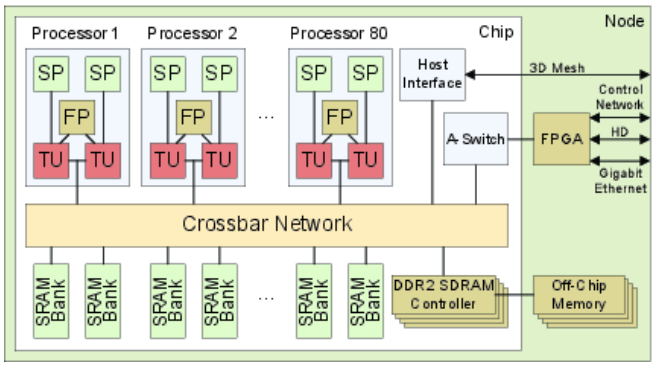

Figure 1. The Cyclops-64 processor chip block-diagram.

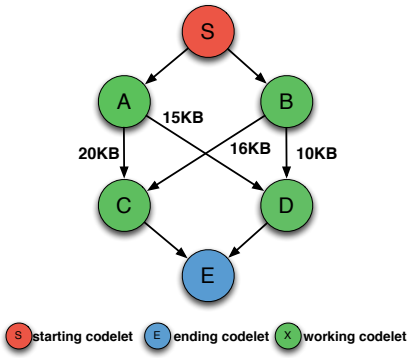

Figure 2. A motivating example of locality exploitation in the codelet model. The weight on an edge represents the amount of locality that can be exploited by scheduling the two ends of the edge on the same core. The best plan exploits 31KB locality by scheduling AD to one core and BC to another.

serve that our algorithms achieve up to 68.1% performance improvement and 40.7% energy reduction comparing to the state-of-the-art codelet scheduling.

The rest of the paper is organized as follows. Section II introduces the background architecture and execution model of our study in this paper. Section III introduces the methodology for solving the problem. Section IV introduces the algorithms in the application of our methodology. Section V reports our emulation platform and experimental results. Section VI discusses the related work. Section VII summarizes the paper.

II. Background

We use the IBM Cyclops-64 many-core architecture (C64) as our testbed. On C64, we study various algorithms that automatically exploit locality in the codelet model. We introduce the C64 architecture and the codelet model in Section II-A and II-B respectively.

A. The Cyclops-64 Architecture

Fig. 1 shows a block-diagram of a C64 node. Each node contains a 160-core chip, clocked at 500MHz. The chip is comprised of 80 processors, each of which contain two cores (called thread units or TU) and one floating point unit (FP). Each TU features a simple 64-bit in-order RISC architecture with 64 64-bit registers in its register file. Each FP is able to issue one fused multiply-add instruction (FMA) per cycle. Hence a C64 node has a theoretical peak performance of 80GFLOPS.

C64 has a 3-layer memory hierarchy with no data cache. First, each TU is assigned a 15KB scratchpad memory (SPM) with a memory bandwidth of 640GB/s. Moreover, all the TUs are able to access 2.4MB of on-chip shared SRAM with a memory bandwidth 320GB/s[1] Finally, the chip is equipped with 1GB of off-chip DRAM (16GB/s memory bandwidth). Both SRAM and DRAM are accessed through a 96-port crossbar switch.

[1]In practice, the amount of shared SRAM vs SPM is configurable at boot-time.

B. The Codelet Model

The codelet model is a fine-grain dataflow-inspired parallel execution model. In an application written for the codelet model, all code is partitioned into codelets. A codelet is a sequence of instructions that can be executed in a non-blocking fashion. A codelet extends traditional dataflow operation semantics beyond data-driven actors. A codelet is event-driven, firing once all data is available and all resource requirements are met.

All the codelets are linked together based on data dependencies to form a Codelet Graph (CDG) very similar to a dataflow graph [5]. A CDG is contained in a Threaded Procedure (TP). A TP is functionally invoked and contains all the necessary space for the data passed between codelets. Within a TP the CDG is static and acyclic. Our current work focuses on the static CDGs within a TP.

III. Methodology

This section introduces the locality exploitation problem and our solution. Firstly, we use an example to motivate the idea of locality exploitation in the codelet model in Section III-A. Next we formalize the locality exploitation problem as the Best Scheduling Problem in Section III-B. Finally, we introduce our methodology for solving the Best Scheduling Problem in Section III-C.

A. Motivating Example

To motivate the exploitation of locality in the codelet model, we present the following example. Fig. 2 shows 6 codelets and their dependencies. The starting and ending codelets do not affect the locality exploitation. The 4 codelets in middle are working codelets. The arrows from A and B to C and D indicate data dependencies between the codelets. A codelet is unable to begin execution until its dependencies are satisfied. The numbers on each arrow signifies the data generated by source of the arrow and

```
/*Define # of parents for codelets C & D*/
1:  dep_t depC=swarm_Dep_INITIALIZER(2,&C,...);
2:  dep_t depD=swarm_Dep_INITIALIZER(2,&D,...);

3:  CODELET_IMPL_BEGIN(A) //Begin of codelet A
4:      ... //A's work;
        /*Satisfy A's dependencies for C & D*/
5:      swarm_Dep_satisfyOnce(&depC);
6:      swarm_Dep_satisfyOnce(&depD);
7:  CODELET_IMPL_END; //End of codelet A

8:  CODELET_IMPL_BEGIN(B) //Begin of codelet B
9:      ... //B's work;
        /*Satisfy B's dependencies for C & D*/
10:     swarm_Dep_satisfyOnce(&depC);
11:     swarm_Dep_satisfyOnce(&depD);
12: CODELET_IMPL_END; //End of codelet B

13: CODELET_IMPL_BEGIN(C) //Begin of codelet C
14:     ... //C's work;
15: CODELET_IMPL_END; //End of codelet C

16: CODELET_IMPL_BEGIN(D) //Begin of codelet D
17:     ... //D's work;
18: CODELET_IMPL_END; //End of codelet D
```

Figure 3. A simplified SWARM codelet program corresponding to the codelet graph in Fig. 2 (starting and ending codelets are omitted). The SWARM runtime handles the codelet graph creation (line 1 and 2) and dependency satisfaction (line 5,6,10,and 11).

consumed by the sink. This number also indicates potential locality. For example, the arrow between A and C specifies that 20KB of data produced by A will be consumed by C. In general, A has to store this 20KB data into shared memory since it guarantees that C is able to access the data no matter where C is executed. However, if A and C are scheduled to the same core, A does not need to store the 20KB data into the shared memory. Instead, A may store the data into the core's local storage for future access of C. In such a way, we exploit the locality between A and C.

Fig. 3 shows a simplified codelet program written in SWARM [6]. The program corresponds to the codelet graph in Fig. 2 without the starting and ending codelets. In the program, line 1 and 2 specify that both codelets C and D have two parents, respectively. Line 5 and 6 satisfy the dependency from A to C and D once A's work is done. Similarly, line 10 and 11 satisfy the dependency from B to C and D. SWARM has a runtime to create the codelet graph, maintain the dependencies, and schedule the codelet whose dependencies have all been satisfied.

In general, there may be multiple choices to exploit locality. For example, we may schedule AC on one core and BD on the other. This scheduling plan exploits 30KB locality. However, the best plan for this example is to schedule AD on one core and BC on the other, which exploits 31KB locality. For a more complicated case that contains many codelets and dependencies, there may be exponential selections. It would be hard for a programmer to figure out the optimal schedule exploiting maximum locality. In this paper, we discuss a number of algorithms that automate selecting an optimal or nearly optimal schedule with respect to maximizing locality. The details will be discussed in Section IV.

B. Problem Statement

In this section, we formalize the problem of locality exploitation as the Best Scheduling Problem. We assume that the CDG is statically known. We also assume that the information of potential locality is known. By partitioning the CDG into several groups of codelets, we can generate a static schedule. Each group may be assigned to a single core. Then the adjacent codelets (the pair of codelets that are executed contiguously on the same core) may use local storage as a buffer to pass data. It reduces not only the latency of the memory access, but also saves energy as data is produced and consumed in place. Since the schedule is static, the execution order of the codelets assigned to the same group must be fixed. That is, the codelets in the same group are totally ordered in the CDG. The generated schedule should guarantee the maximum amount of potential locality is exploited. With this, we define the Best Scheduling Problem as follows:

(Best Scheduling Problem) Given a weighted CDG $G =< V, E, W >$ and a positive integer n, where V represents the codelets, E represents the dependencies, W represents the potential locality, and n represents the total number of cores, find a mapping

$$f : V \rightarrow \{1, \ldots, n\}$$

to satisfy the following requirement:

$$\begin{aligned} Maximize &: \{\sum W(v_1, v_2) | f(v_1) = f(v_2) \wedge v_1 \leftrightarrow v_2\} \\ Subject\ to &: \forall f(v_1) = f(v_2), v_1 \xrightarrow{P} v_2 \vee v_2 \xrightarrow{P} v_1 \end{aligned}$$

,where $v_1 \leftrightarrow v_2$ means that v_1 and v_2 are adjacent (executed one after the other) in the same group, and $v_1 \xrightarrow{P} v_2$ means that there exists a path in G from v_1 to v_2.

C. Solution

We propose three algorithms to solve the Best Scheduling Problem. The three algorithms have different trade-off in the algorithmic complexity, locality exploitation, program performance, energy efficiency, and required computation resources. The features of the three algorithms are as follows:

- **Min-cost flow based algorithm:** This algorithm converts the Best Scheduling Problem to a min-cost flow problem. It guarantees an optimal solution. The time complexity is $O(knmlog(n))$ where k is the number of cores, n is the number of codelets, and m is the number of dependencies in the codelet graph.
- **Max first algorithm:** This is a heuristic algorithm that provides a nearly optimal solution in practice. Its time

complexity is $O(nlog(n)+m)$ which is the lowest in the three algorithms.

- **Graph partitioning based algorithm:** This algorithm converts the Best Scheduling Problem to a graph partitioning algorithm. Its time complexity is $O(mlog(k))$ which is lower than the min-cost flow based algorithm.

The details of the three algorithms will be introduced in Section IV.

IV. Algorithm

In this section, we introduce three algorithms used to automatically exploit locality in the codelet model. We first formalize the problem in Section III-B. Then we explain the three algorithms in Section IV-A,IV-B ,and IV-C, respectively.

A. Min-cost Flow Based Algorithm

The Best Scheduling Problem can be converted to a min-cost flow problem. Given the weighted codelet graph, we can create a flow network that has two properties: (1) Each scheduling plan corresponds to a flow in the flow network, and vice versa; and (2) The sum of available weights in a scheduling plan and the cost of the corresponding flow are anticorrelated.

A min-cost flow algorithm finds the flow that has minimum cost among all possible flows. Applying the above two properties, we know that the corresponding scheduling plan is the one with maximum sum of available weights among all the plans. Therefore, the solution of the min-cost flow problem corresponds to the solution of the Best Scheduling Problem.

Algorithm 1 shows how to convert a given weighted codelet graph G into a flow network N and how to map the solution of the min-cost flow problem to the solution of the Best Scheduling Problem.

From the algorithm, we can see that N has two initial vertices src_1 and src_2 where src_1 connects to src_2 with capacity equals to the total number of cores. The capacity guarantees that the solution does not exceed the provided number of cores.

Each codelet v in G is represent as two vertices v_1 and v_2 in N. v_1 connects v_2 with capacity equals to 1. This capacity guarantees that the codelet v can be executed at most once. Moreover, the cost between v_1 and v_2 is equal to a big negative constant number $-M$. This cost guarantees that v will be executed since the min-cost flow must go through (v_1, v_2) to include $-M$ in its overall cost. Therefore, the capacity and the cost between v_1 and v_2 guarantee that the corresponding codelet v will be executed once and only once.

If codelets v depends on codelet u in G, then u_2 connects to v_1 with a capacity equal to 1 in N. If this capacity is reached in the min-cost flow, then v will execute right after u on the same core in the best scheduling plan. Since the locality between u and v is represented as $W(u,v)$ in G, the cost of (u_2, v_1) in N is the corresponding negative number $-W(u,v)$.

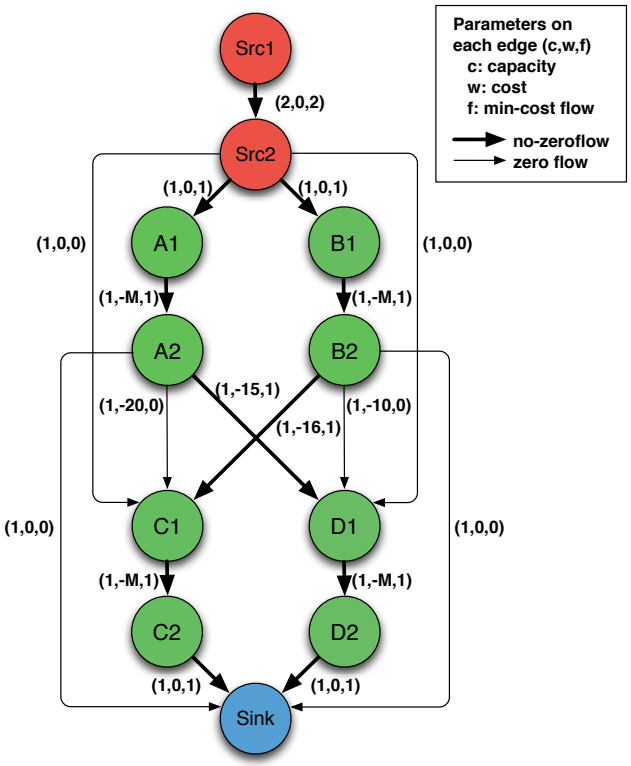

Figure 4. The flow network converted from the codelet graph in Figure 2. The resulting min-cost flow is consist of two paths: $Src_1 Src_2 A_1 A_2 D_1 D_2 Sink$ and $Src_1 Src_2 B_1 B_2 C_1 C_2 Sink$. The two paths correspond to the best scheduling plan that schedules codelet AD on one core and BC on the other.

Finally, src_2 connects to v_1, and v_2 connects to $sink$. Both the capacities equal to 1. If the former capacity is reached, then the corresponding v is the first codelet on some core. If the latter capacity is reached, then v is the last codelet on some core.

Fig. 4 shows the flow network converted from the codelet graph of Fig. 2. The min-cost flow is represented by thick arrows. The min-cost flow goes through two paths: $(src_1, src_2, A_1, A_2, D_1, D_2, sink)$ and $(src_1, src_2, B_1, B_2, C_1, C_2, sink)$. That means the corresponding best scheduling plan will schedule AD on one core and BC on the other.

B. Max First Algorithm

In this section, we introduce a heuristic algorithm (called max first algorithm) to provide nearly optimal solution for the Best Scheduling Problem. The main idea of the algorithm is to schedules the two codelets with maximum potential locality to some adjacent position on the same core at every step.

Algorithm 1: Using min-cost flow to solve the Best Scheduling Problem

input : Total number of cores n
A weighted codelet graph $G =< V, E, W >$

output: A vector par stores the parent of each codelet. Codelets $par[i]$ and i will be scheduled to the same core and executed one after the other. If codelet i is the first codelet scheduled to some core, then $par[i]$ equals -1.

Data: $N =< V, E, W, C >$ is the flow network that corresponds to G, where
W is the weight *i.e.* cost of each edge
C is the capacity of each edge in N
$-M$ is a big negative number to represent $-\infty$
F is the min-cost flow

PSEUDO CODE:

$N \leftarrow \emptyset$;
$N.V \leftarrow N.V \cup src_1 \cup src_2 \cup sink$;
$N.E \leftarrow N.E \cup (src_1, src_2)$;
$N.W(src_1, src_2) \leftarrow 0$;
$N.C(src_1, src_2) \leftarrow n$;
for *each* $v \in G.V$ **do**
  $N.V \leftarrow N.V \cup v_1 \cup v_2$;
  $N.E \leftarrow N.E \cup (v_1, v_2) \cup (src_2, v_1) \cup (v_2, sink)$;
  $N.W(v_1, v_2) \leftarrow -M$;
  $N.W(src_2, v_1) \leftarrow 0$;
  $N.W(v_2, sink) \leftarrow 0$;
for *each* $(u, v) \in G.E$ **do**
  $N.E \leftarrow N.E \cup (u_2, v_1)$;
  $N.W(u_2, v_1) \leftarrow -w(u, v)$;
for *each* $(u, v) \in N.E - (src_1, src_2)$ **do**
  $C(u, v) \leftarrow 1$;
$F \leftarrow$ `MinCostFlow(`N`)`;
for *each* $v \in G.V$ **do** $par[v] \leftarrow -1$;
for *each* $u, v \in G.V$ **do**
  if $F(u_2, v_1) == 1$ **then**
    $par[v] \leftarrow u$;

The algorithm is shown in Algorithm 2. Initially, all the edges are put into an edge pool. Then the algorithm picks the max edge (i.e., the one with highest weight) from the pool. The two ends of the edge will be scheduled to the same core in some adjacent position. The edges against the scheduling will be removed from the edge pool. The algorithm continues this process of picking and removing until the edge pool is empty.

The max first algorithm has lower time complexity than the min-cost flow based algorithm. If we use a heap as the data structure to store the edge pool, its time complexity is $O(nlog(n) + m)$ where n is the total number of codelets and m is the total number of dependencies.

Algorithm 2: Max first algorithm

input : A weighted codelet graph $G(V, E, W)$

output: A vector par stores the parent of each codelet. Codelets $par[i]$ and i will be scheduled to the same core and executed one after the other. If codelet i is the first codelet scheduled to some core, then $par[i]$ equals -1.

Data: P is an edge pool stored in a binary heap

PSEUDO CODE:

$P \leftarrow G.E$;
for *each* v *in* $G.V$ **do** $par[v] \leftarrow -1$;
while $P \neq \emptyset$ **do**
  $(u, v) \leftarrow$ `MaxElement(`P`)`;
  $P \leftarrow P - (u, v)$;
  $par[v] \leftarrow u$;
  for $(u, z) \in P$ **do**
    $P \leftarrow P - (u, z)$;
  for $(z, v) \in P$ **do**
    $P \leftarrow P - (z, v)$;

As a heuristic algorithm, the max first algorithm does not guarantee optimal solution. One example is shown in Fig. 2. The max first algorithm will exploit 30KB locality by scheduling AC on one core and BD on the other. However, the optimal solution can exploit 31KB locality by scheduling AD on one core and BC on the other. In practice, we found that the max first algorithm always finds nearly optimal solution (no more than 7.0% worse). The details will be explained in Section V.

C. Graph Partitioning Based Algorithm

We can also convert the Best Scheduling Problem to a graph partitioning problem. A graph partitioning algorithm [7] partitions the vertices of a weighted graph into multiple groups. It guarantees that the sum of inter-group weights (i.e., the weights of edges that go across groups) is minimum or nearly minimal. By applying the graph partitioning algorithm on a codelet graph, we may partition the codelets into n groups where n is the total number of cores. Then the codelets in the same group will be scheduled to the same core. The minimum sum of inter-group weights indicates that the schedule minimizes the waste of inter-core locality.

The benefit of the graph partition algorithm is that it can handle any given number of cores. However, it has two problems: (1) it may reduce parallelism of the codelet graph. This is because the two codelets with no dependency may be scheduled to the same core in its solution. So the scheduling plan may need to add extra dependencies to maintain total order of codelets on the same core. (2) The algorithm cannot guarantee optimal solution because it counts false locality. This is because the algorithm only minimize the waste of

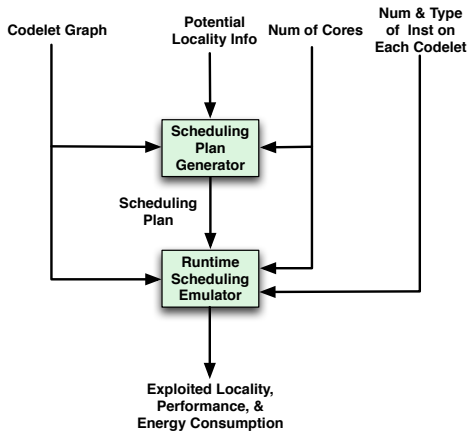

Figure 5. Experiment design.

inter-core locality. For two codelets scheduled to the same core, the false locality is counted even if the two codelets are not adjacent. In practice it is unsafe to exploit such locality because it requires data to overstay in the local storage, which reduces the available space of local storage for the execution of other codelets and may cause overflow.

V. Experiment

We evaluate the three algorithms on the C64 architecture model. Section V-A introduces our experimental design. Section V-B summarizes our major experimental observations. Finally, section V-C reports our experimental results.

A. Experimental Design

Fig. 5 shows the overview of our experiment design. We developed the following two modules for the experiments.

- **Scheduling plan generator:** This module uses our proposed algorithms in Section IV to automatically generate the codelet scheduling plan for locality exploitation. The inputs of the module are the static codelet graph, the potential locality information among the codelets, and the total number of cores. To fit the C64 architecture feature, we assume that each core will execute one codelet at a time. The output of the module are the scheduling plans generated by the three algorithms.
- **Runtime Scheduling Emulator:** This module emulates the codelet runtime that schedules the codelet on a C64 chip. The emulator could use either the default scheduling approach or the input scheduling plan. The default scheduling focuses on workload balancing but no locality exploitation, which matches the state-of-the-art codelet scheduling approaches. The input scheduling plan may exhibit various locality exploitation, depending on the algorithm that generates the plan. The other inputs of the module are the codelet graph, the total number of cores, and the numbers and types of instructions in each codelet. The output of the module are the exploited locality, performance, and energy consumption by using the different scheduling on the codelet graph, respectively.

We use the following six applications in our experiments:

- `mm` (matrix multiplication kernel): This benchmark is based on the previous study of matrix multiplication on C64 [8]. It computes $C = A \times B$ where A,B, and C are all 192×192 matrices that store double precision floating point numbers. C is further partitioned into many 6×6 tiles. Therefore, each codelet computes the multiplication of a 6×192 matrix and a 192×6 matrix to generate a 6×6 tile in C. Since there are 1024 tiles in C, the amount of the codelets are the same. Each codelet executes $6 \times 6 \times 192 = 6912$ float multiply-add and $192 \times 6 \times 2 = 2304$ load instructions (half on A and half on B). For two codelets that load the same part of A, the potential locality is $16Bytes \times 2304/2 = 18432Bytes$.
- `ms` (merge sort kernel): This benchmark computes a sorting of $10K$ integers via a 7-level merge process. Therefore, the codelet graph is a 7-level binary tree with 127 codelets. A codelet at level l ($0 \leq l \leq 6$) needs to execute $10K/2^l$ times of comparisons, loads, and stores, respectively. Half of the loads in a codelet is from one child and the other half is from the other child. Therefore, the potential locality between a parent codelet at level l and one of its child is $8Bytes \times 10K/2^{(l+1)}$.
- `rt_ci` (random tree with computation-intensive codelets): This is a randomly generated tree-structure codelet graph. The total number of codelets is 160 which matches the total number of cores on C64. Each codelet is computation-intensive. The amount of computation instructions is 6 times of the amount of memory access instructions. This ratio matches the ratio in `mm` because it is also a computation-intensive benchmark.
- `rt_mi` (random tree with memory-intensive codelets): This is also a randomly generated tree-structure codelet graph with 160 codelets. Each codelet is memory-intensive. The amount of computation instruction equals to the amount of memory access instructions. This ratio matches the ration in `ms` because it is also a memory-intensive benchmark.
- `rg_ci` (random graph with computation-intensive codelets): This benchmark is similar to `rt_ci`. However, the codelet graph is a randomly generated graph with 160 codelets and 320 dependency edges. In our observation, most codelet graphs have low average fanout (*e.g.*, around 2 for a codelet graph that represents a parallel *for* loop). That is why we set the average fanout to be 2.

- rg_mi (random graph with memory-intensive codelets): This benchmark is similar to rt_mi. However, the codelet graph is a randomly generated graph with 160 codelets and 320 dependency edges.

We assume that the data of all the applications are initially stored in the off-chip DRAM. For computation-intensive applications (*i.e.*, mm, rt_ci, and rg_ci), we assume that the application can fully hide the latency of memory accesses. Therefore, each memory access instruction only takes one cycle to issue. For memory-intensive applications (*i.e.*, ms, rt_mi, and rg_mi), we assume that the application is unable to hide the latency of memory accesses. Therefore, a load on the scratchpad memory will cause 2 cycles delay and a load on the off-chip DRAM will cause 57 cycles delay according to the C64 feature. Since a store on C64 does not have acknowledgement, it only takes one cycle to issue.

In the experiments, we tested 4 scheduling algorithms. They are described in Table I: Base, MCF, MF, and GP. Base focuses on workload balancing but no locality exploitation, which matches the state-of-the-art codelet scheduling approaches. In Base, the codelet runtime maintains a global codelet queue. Whenever a codelet has satisfied all of its dependencies, the codelet runtime will put it in the codelet queue. Then the runtime will look for an available core to execute the codelet. If there are multiple available cores, the runtime will arbitrarily pick one. The other three algorithms have different tradeoffs on locality exploitation, algorithm complexity, etc. Without loss of generality, we assume that the scheduling takes trivial overhead. That is because the execution time of a codelet is normally much larger than the overhead of the scheduling.

B. Major Observations

The major observations of our experimental results are summarized as follows:

- MCF always exhibits best locality exploitation. It reduces up to 59.7% of global memory accesses. MF is the second best (within 7.0% of difference comparing to MCF).
- The applications using MCF outperform the same applications using the other scheduling algorithms. MCF achieves up to 68.1% of performance improvement comparing to Base. MF is the second best (within 9.1% of difference comparing to MCF).
- MCF exhibits best energy reduction on both overall and dynamic energy consumptions. It reduces up to 40.7% overall energy and 59.2% dynamic energy comparing to Base. MF is the second best (within 8.5% of difference on overall energy and 3.6% on dynamic energy comparing to MCF).

Table I
DESCRIPTION OF THE VARIOUS ALGORITHMS USED TO SCHEDULE CODELETS ON C64. DESCRIPTION OF EACH ALGORITHM IS IN THE RIGHT HAND SIDE COLUMN.

Name	Description
Base	Basic scheduling without locality exploitation
MCF	Min-cost flow based algorithm (see Section IV-A)
MF	Max-first algorithm (see Section IV-B)
GP	Graph partitioning based algorithm (see Section IV-C)

C. Experimental Result

In this section, we report and analyze the experimental results of the evaluation on locality exploitation, performance, and energy efficiency of the four scheduling algorithms.

Locality exploitation

Fig. 6 shows the best locality exploitation of three algorithms (MCF, MF, and GP) applied on the six applications. We do not show the result of Base because it does not exploit locality. In the figure, the x-axis represents the six applications. The y-axis represents the locality exploitation, that is, the percentage of global memory accesses that have been reduced via buffer in local storages. We have the following observations from Fig. 6:

- MCF exhibits better locality exploitation than the other two algorithms. This is because MCF guarantees optimal solution that maximizes the locality exploitation. The other two algorithms may not reach optimal solution because they are heuristic algorithms.
- MF provides nearly optimal solutions. The difference of locality exploitation is within 7.0% between MF and MCF. The worst case of MF happens on rg_ci and rg_mi because the codelet graphs are too complicated. The other four applications have simpler codelet graphs (either tree structure or uniformed weight). So MF finds optimal (or very close to optimal) solutions for them.
- GP finds worst solutions. This is because GP suffers from false locality between two codelets that are scheduled to the same core but not adjacent. The reason was explained earlier in Section IV.

Performance

Fig. 7 shows the performance evaluation of the four algorithms on various applications. The x-axis represents the various applications. The y-axis features the normalized execution time of each application by using the four scheduling algorithms, respectively. To make the comparison fair, all the algorithms use the same amount of cores. We set the amount to be equivalent to the requirement of MF because it is the only algorithm that does not support arbitrary number of cores. We have the following observations from Fig. 7:

- MCF outperforms the other three algorithms. It achieves up to 68.1% of performance improvement comparing to Base on rg_mi. For memory-intensive applications, the latency of global memory accesses on DRAM is

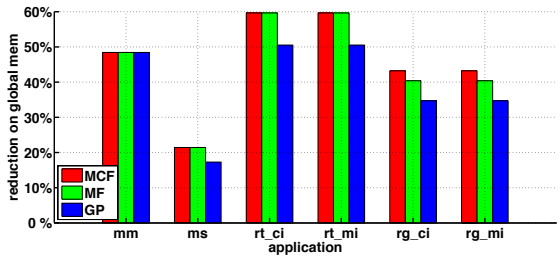

Figure 6. Best locality exploitation on various applications by using the three algorithms from Section IV. X-axis represents the various applications. Y-axis features the percentage of global memory accesses that have been reduced by each algorithm. Higher is better. `MCF` exhibits best locality exploitation. It reduces up to 59.7% of global memory accesses on application `rt_ci` and `rt_mi`. `MF` provides a good quality solutions (within 7.0% of difference comparing to `MCF`). `GP` is the worst.

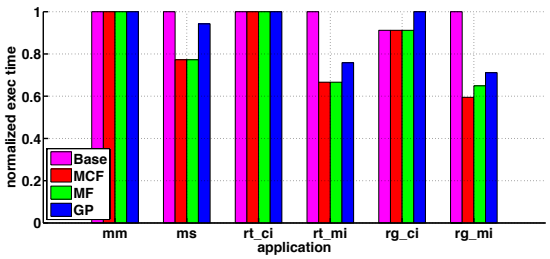

Figure 7. Performance evaluation of the four algorithms on various applications. X-axis represents the various applications. Y-axis features the normalized execution time of the applications by using the four scheduling algorithms. Lower is better. `MCF` outperforms the other algorithms (up to 68.1% of performance improvement comparing to `Base`). `MF` is the second best (no more than 9.1% slower comparing to `MCF`). `Base` is the worst in most of the cases but it outperforms `GP` on `rg_ci`.

much longer than that of local storage accesses. So the locality exploitation greatly reduces execution time of each codelet. Therefore, the performance of the application is improved. However, the performance of computation-intensive applications is not affected by the locality exploitation. The reason is that such applications may fully hide latency of memory accesses. So both DRAM access and local storage access take same execution time.

- `MF` is the second best. This is because its locality exploitation is worse than `MCF`. In special cases, better locality exploitation may not guarantee better performance. However, in our experiments we haven't observed such a special case.
- `GP` performs worse than `MCF` and `MF`. There are two reasons: (1) `GP` exhibits worst locality exploitation in the three algorithms; and (2) `GP` may introduce extra dependency edges to the codelet graph, which may reduce the parallelism of the application. However, `GP` still outperforms `Base` for memory-intensive applications because the locality exploitation reduces the latency of memory accesses. For computation-intensive applications, `GP` may slowdown the performance because it may reduce parallelism of the application. One example is that `GP` is slower than `Base` on `rg_ci`.

Energy efficiency

The overall energy consumption of an application consists of static and dynamic energy consumptions. The static energy consumption is determined by the execution time. On C64 it is 64.11W as explained in [9]. The dynamic energy is determined by the number and type of the executed instructions. Table II shows the energy consumption of various instructions on C64 that was earlier tested in [9]. `lddram`, `ldsram`, and `ldspm` are double-word load instructions on DRAM memory, SRAM memory, and SPM, respectively. Similarly, `stddram`, `stdsram`, and `stdspm` are the corresponding store instructions. `mov` is the access

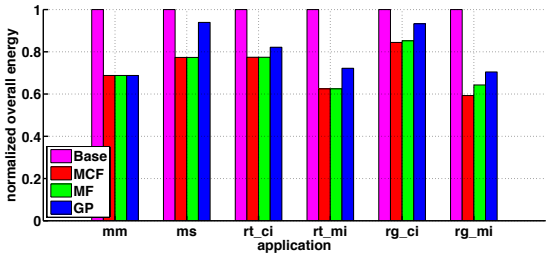

Figure 8. Overall energy consumption on various applications by using the four algorithms. X-axis represents the various applications. Y-axis features the normalized overall energy consumption of each algorithm applied to various applications. Lower is better. `MCF` exhibits best energy efficiency. It reduces up to 40.7% of overall energy comparing to `Base`. `MF` is the second best (within 8.5% difference comparing to `MCF`).

on a double-word register. `fmad` is the multiple and add computation on double precision floating point numbers. Integer and logical operations consume similar energy. We use `add` to represent them. Since [9] does not provide the energy consumption of `lddspm` and `stdspm`, we use the following formula to estimate them: `lddspm` (or `stdspm`) = (`lddsram` (or `stdsram`) - `mov`) $\times$ *ratio* + `mov`. We set *ratio* as $1/3$ because the energy consumption of SPM accesses is closer to register accesses than SRAM accesses.

Fig. 8 and Fig. 9 show the normalized overall and dynamic

Table II
ENERGY CONSUMPTION PER INSTRUCTION.

Instruction	Energy (pJ/Operation)
lddram	48924.10
stddram	51488.99
lddsram	964.65
stdsram	548.31
mov	105.48
lddspm	535.065
stdspm	326.895
fmad	245.27
add	127.65

energy consumption of the four algorithms on various applications, respectively. We have the following observations from the two figures.

- `MCF` is the most energy efficient algorithm. It reduces up to 40.7% of overall energy and 59.2% of dynamic energy compared to `Base`. That is because `MCF` exhibits both the best locality exploitation and the best performance.
- `MF` is the second best due to its relatively good solution for both locality exploitation and performance.
- `GP` is worse than the above two due to its poor solution for both locality exploitation and performance.

VI. Related Work

The Codelet execution model finds its roots in the classical dataflow model originally proposed by Dennis[5]. Dataflow has undergone several iterations which have served as inspirations for the Codelet model including dynamic dataflow, macro dataflow[10], and the EARTH system[1]. The most closely related is the EARTH system as it incorporates dual level parallelism in the form of data-driven fibers and functionally invoked threaded procedures. The primary difference between EARTH and the Codelet model is the operational semantics of fibers versus codelets as fibers are data-driven and codelets are event-driven.

Dataflow based execution models have received more interest as multi and many-core architectures have become more popular. These include several codelet based efforts such as SWARM[6], FreshBreeze[11], and the ParalleX execution model[12] which can directly benefit from our proposed techniques. In addition, we believe our approach may extend beyond codelet based execution models to more general dataflow related technologies including Intel's Threading Building Blocks' augmented flow graphs[13], pragma based StarSs[14], and the OpenMP-based OpenStream[15].

In the data-flow processor design[16][17][18], the similar problem also exists, called program allocation. This posed the problem of maximize concurrency while minimizing contention for processing resources. In addressing this problem static scheduling was proposed[19]. This work utilizes execution time and communication costs to determine an appropriate schedule. Our methods differ since codelets are much coarser than the originally proposed dataflow actors and their execution time can not be guaranteed.

Currently there are several approaches to addressing locality at vary levels. A number of HPCS languages permit locality abstractions permitting users to express relations between data and processing elements. Examples of these abstractions include X10's places[20] and Chapel's locales[21]. For distributed systems, PGAS languages have included data distribution features such as UPC[22] and CAF[23]. In shared memory systems OpenMP is most popular, however it has no inherent mechanism for locality. As NUMA effects are becoming more prevalent within a single node efforts to introduce notions of locality have been proposed[24]. Besides, multiple efforts to introduce locality aware work stealing have been explored in such works as SLAW[25], CATS[26]. Lastly, locality optimization is used in compiler optimization. [27] and [28] show the way to reuse the cache and register with help of compiler, resulting in significant performance improvement.

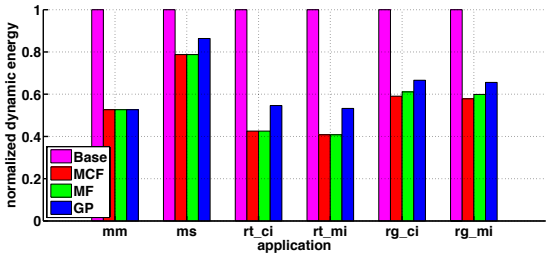

Figure 9. Dynamic energy consumption on various applications by using the four algorithms. X-axis represents the various applications. Y-axis features the normalized dynamic energy consumption of each algorithm applying on various applications. Lower is better. `MCF` exhibits best energy efficiency. It reduces up to 59.2% of dynamic energy comparing to `Base`. `MF` is the second best (within 3.6% difference comparing to `MCF`).

VII. Conclusion

In this paper, we propose and study the codelet scheduling problem for maximizing locality exploitation among codelets. The solution provides a codelet scheduling that improves performance and saves energy by reducing the total amount of global memory access. To solve this scheduling problem, we propose one polynomial time algorithm together with two other heuristic algorithms, each of which has different trade-offs in algorithmic complexity, locality exploitation, program execution time, and computation time. We analyze and show suitable algorithms which should be used in different circumstances. We evaluate our algorithms on an emulator based on the IBM Cyclops-64 many-core architecture. The experimental result shows that our algorithms reduce up to 59.7% of global memory access via the locality exploitation. We also observe that our algorithms give up to 68.1% performance improvement and 40.7% energy reduction comparing to the state-of-the-art codelet scheduling approach.

Acknowledgment

This research was supported by the NSF through grants CCF-0833122, CCF-0925863, CCF-0937907, and OCI-0904534. Moreover, this work was supported by European FP7 project TERAFLUX, id. 249013. We thank Prof. R. Govindarajan for his suggestions on the experimental design.

References

[1] K. B. Theobald, "EARTH: an efficient architecture for running threads," Ph.D. dissertation, McGill University, Montreal, Que., Canada, Canada, May 1999, AAINQ50269.

[2] "IBM Cell Broadband Engine." [Online]. Available: http://www-01.ibm.com/chips/techlib/techlib.nsf/products /Cell_Broadband_Engine

[3] J. Cuvillo, W. Zhu, Z. Hu, and G. R. Gao, "Tiny threads: A thread virtual machine for the cyclops64 cellular architecture," in *Fifth Workshop on Massively Parallel Processing (WMPP 2005)*, 2005, p. 265.

[4] R. Knauerhase, R. Cledat, and J. Teller, "For extreme parallelism, your os is sooooo last-millennium," in *Proceedings of the 4th USENIX conference on Hot Topics in Parallelism*, ser. HotPar'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 3–3.

[5] J. Dennis, "First version of a data flow procedure language," in *Programming Symposium*, ser. Lecture Notes in Computer Science, B. Robinet, Ed. Springer Berlin Heidelberg, 1974, vol. 19, pp. 362–376.

[6] C. Lauderdale and R. Khan, "Towards a codelet-based runtime for exascale computing: position paper," in *Proceedings of the 2nd International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, ser. EXADAPT '12. New York, NY, USA: ACM, 2012, pp. 21–26.

[7] G. Karypis and V. Kumar, "Multilevel algorithms for multi-constraint graph partitioning," in *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, ser. Supercomputing '98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 1–13.

[8] E. Garcia, I. E. Venetis, R. Khan, and G. R. Gao, "Optimized dense matrix multiplication on a many-core architecture," in *Proceedings of the 16th international Euro-Par conference on Parallel processing: Part II*, ser. Euro-Par'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 316–327.

[9] E. Garcia, D. Orozco, and G. R. Gao, "Energy efficient tiling on a many-core architecture," in *Proceedings of the 4th workshop on programmability issues for heterogeneous multicores (MULTIPROG 2011)*, Heraklion, Greece, Jan. 2011.

[10] J. Silc, B. Robic, and T. Ungerer, "Asynchrony in parallel computing: From dataflow to multithreading," *Journal of Parallel and Distributed Computing Practices*, vol. 1, pp. 1–33, 1998.

[11] J. B. Dennis, "Fresh breeze: A multiprocessor chip architecture guided by modular programming principles," *ACM SIGARCH Computer Architecture News*, vol. 31, pp. 7–15, 2003.

[12] G. Gao, T. Sterling, R. Stevens, M. Hereld, and W. Zhu, "Parallex: A study of a new parallel computation model," in *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, Mar. 2007, pp. 1 –6.

[13] C. Pheatt, "Intel Threading Building Blocks," *J. Comput. Sci. Coll.*, vol. 23, no. 4, pp. 298–298, Apr. 2008.

[14] J. Planas, R. M. Badia, E. Ayguad, and J. Labarta, "Hierarchical task-based programming with starss," *International Journal of High Performance Computing Applications*, vol. 23, no. 3, pp. 284–299, 2009.

[15] A. Pop and A. Cohen, "Openstream: Expressiveness and data-flow compilation of openmp streaming programs," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, pp. 53:1–53:25, Jan. 2013.

[16] J. Dennis, "Data flow supercomputers," *Computer*, vol. 13, no. 11, pp. 48 –56, nov. 1980.

[17] J. R. Gurd, C. C. Kirkham, and I. Watson, "The manchester prototype dataflow computer," *Commun. ACM*, vol. 28, no. 1, pp. 34–52, Jan. 1985.

[18] G. M. Papadopoulos and D. E. Culler, "Monsoon: an explicit token-store architecture," *SIGARCH Comput. Archit. News*, vol. 18, no. 3a, pp. 82–91, May 1990.

[19] B. Lee, A. Hurson, and T. Feng, "A vertically layered allocation scheme for data flow systems," *Journal of Parallel and Distributed Computing*, vol. 11, no. 3, pp. 175 – 187, 1991.

[20] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," *SIGPLAN Not.*, vol. 40, no. 10, pp. 519–538, Oct. 2005.

[21] B. Chamberlain, D. Callahan, and H. Zima, "Parallel programmability and the chapel language," *International Journal of High Performance Computing Applications*, vol. 21, no. 3, pp. 291–312, 2007.

[22] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks, and K. Warren, *Introduction to UPC and language specification*. Center for Computing Sciences, Institute for Defense Analyses, 1999.

[23] R. W. Numrich and J. Reid, "Co-array fortran for parallel programming," *SIGPLAN Fortran Forum*, vol. 17, no. 2, pp. 1–31, Aug. 1998.

[24] L. Huang, H. Jin, L. Yi, and B. Chapman, "Enabling locality-aware computations in openmp," *Scientific Programming*, vol. 18, no. 3, pp. 169–181, 2010.

[25] Y. Guo, J. Zhao, V. Cave, and V. Sarkar, "Slaw: A scalable locality-aware adaptive work-stealing scheduler," in *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, Apr. 2010, pp. 1–12.

[26] Q. Chen, M. Guo, and Z. Huang, "Cats: cache aware task-stealing based on online profiling in multi-socket multi-core architectures," in *Proceedings of the 26th ACM international conference on Supercomputing*, ser. ICS '12. New York, NY, USA: ACM, 2012, pp. 163–172.

[27] G. Gao, R. Olsen, V. Sarkar, and R. Thekkath, "Collective loop fusion for array contraction," *Languages and Compilers for Parallel Computing*, pp. 281–295, 1993.

[28] R. Govindarajan, H. Yang, J. Amaral, C. Zhang, and G. Gao, "Minimum register instruction sequence problem: revisiting optimal code generation for dags," in *Parallel and Distributed Processing Symposium., Proceedings 15th International*, Apr. 2001, p. 8 pp.