# MOMA: Mapping of Memory-intensive Software-pipelined Applications for Systems with Multiple Memory Controllers

Janmartin Jahn, Santiago Pagani, Jian-Jia Chen, Jörg Henkel
Karlsruhe Institute for Technology (KIT), Germany
{jahn, pagani, chen, henkel}@kit.edu

***Abstract*—In many-core systems, the efficient deployment of computational and other resources is key in order to achieve a high throughput. Current state-of-the-art task mapping schemes balance the computational load among cores while avoiding congestions within the communication links. The problem is that a large number of cores running many memory-intensive tasks may congest memory controllers because their number and bandwidth is constrained. To avoid a high throughput degradation that could result from congested memory controllers, the mapping of tasks must be sensitized to the limited bandwidth of off-chip memory. Designing efficient and effective algorithms to optimize the throughput by jointly considering the load of memory controllers, computation, and communication is very challenging.**

**In this paper, we address this problem by distributing cores among applications and then heuristically map tasks such that the load of the memory controllers is sufficiently balanced. Our heuristic also minimizes the effect of decreased throughput resulting from mapping communicating tasks to cores that belong to different controllers.**

**Our experiments encourage us in that we can reduce the saturation of memory controllers and significantly increase the system throughput compared to employing several state-of-the-art task mapping schemes.**

## I. Introduction

Many-core systems have emerged as a powerful means to achieve performance increases through parallel processing beyond the capabilities of single-core systems and are projected to integrate hundreds or even thousands of cores on a single chip [1]. A key challenge is the *efficient* employment of cores that is largely impacted by the task mapping [2]–[4]. State-of-the-art task mapping schemes aim at balancing the computational load among cores while avoiding contentions in the communication links between them (see [5]–[7]). Due to the rapidly increasing number of cores, bandwidth limitations of *memory controllers* (i.e. controllers located on-chip that enable access to off-chip memory) may have a significant impact on the system throughput when running memory-intensive tasks [8].

This limitation is mostly due to the fact that both the number of pins to connect the chip with off-chip memory as well as the bandwidth of each individual pin is limited [9]. Hence, the number of memory controllers and their individual bandwidth are also limited. In systems with a large number of cores, each memory controller has to serve (too) many requests [8]. As an example, Intel's newest Xeon Phi™ 5110P integrates 16 memory controllers for 61 cores and there, each memory controller serves the accesses of approx. 4 cores [10]. However, as the number of memory controllers is limited by the pin count constraints [8], each memory controller may need to serve the accesses of 64 cores in a system with 1024 cores. Thus, it has to serve 16 times the requests.

In case one memory controller serves many memory-intensive tasks, it may operate in *saturation*, i.e. it may be requested to access more data than it can provide. This reduces the bandwidth that is available for the individual tasks, and hence their throughput will degrade. Such a saturation of memory controllers has recently been identified as the major cause for deteriorated throughput [9]. Furthermore, when tasks on different memory controllers require communications, the corresponding data have to be copied from one memory controller to the other, inducing a performance penalty. Section IV shows an example of how task mapping to balance the computational load among cores can cause memory controllers to operate in saturation and how that may degrade the system throughput.

However, it is challenging to map memory-intensive tasks *jointly* based on computation, communication, and off-chip memory accesses because the throughput depends on the saturation of memory controllers, and vice versa. In this paper, we present MOMA to *efficiently* tackle this challenge for memory-intensive software-pipelined applications. Software pipelines are a well-established paradigm for parallel programming in systems with distributed, private memories and is most prominently employed for stream-processing applications. To summarize, our novel contributions are:

- Our MOMA algorithm *jointly* optimizes the mapping of software pipelines based upon considering computation, communication, and the load of memory controllers.
- We provide simulated experiments, which demonstrate that MOMA achieves a significant improvement over state-of-the-art mapping schemes in systems that comprise a large number of cores.

The rest of this paper is organized as follows: Section II defines terms and sets definitions, Section III discusses related work, and Section IV illustrates a motivational example. In Section V, our system model and problem definition is presented, while Section VI details our MOMA algorithm. Experimental results showing the significant advantages of our novel technique are presented in Section VII while Section VIII concludes this paper.

## II. Terms and Definitions

For the rest of this paper, following terms and definitions hold:

1) In this paper, *task mapping* refers to assigning a task to a core.
2) A *software pipeline* is a parallel application that comprises multiple *stages* that repeatedly perform *iterations* (i.e. computations) on a stream of input data. The output of one stage forms the input of its direct successor, and there is no further communication. Each stage is a software task.
3) We refer to the time required for the computations of each iteration as the *computational requirements* of a stage, assuming the responsible memory controller is not saturated.
4) The *bandwidth requirements* of a stage denote the amount of off-chip memory that it accesses, in MB/s, assuming that its throughput is not limited by the bandwidth constraints of its memory controller. The *memory requirements* of a stage denote this per iteration, in MB.
5) The *throughput* of a software pipeline denotes the number of iterations the pipeline finishes per second. The *system throughput* is the average throughput of all applications. We use this metric since software pipelines often run perpetually such that metrics as makespan are not applicable.
6) *Fusing* consecutive stages means to map them to the same core similar to fusing filters in StreamIt [11]. This reduces the degree of parallelism of the pipeline and the number of cores it uses. No on-chip communication between fused stages is necessary. Stages can be fused (and fused stages can be split) at runtime.

000-0-0000-0000-0/13/$26.00  Published in the 32nd IEEE/ACM International Conference on Computer-Aided Design (ICCAD), San Jose, CA, USA, November 2013.

7) A *memory island* consists of one memory controller and a number of cores. All memory accesses of the cores of one memory island are served by the same memory controller. tasks that are mapped to one memory island can share memory and may pass pointers to data, while tasks mapped to different memory islands transfer data via message passing (MPI, e.g. [12]).
8) The *load* of a memory controller denotes the accesses it serves per second, in MB/s.
9) The *bandwidth constraint* of a memory controller expresses the maximum load, in MB/s.

## III. Related Work

We group the related work into task mapping schemes, design-time placement strategies for memory controllers, and into request optimization strategies.

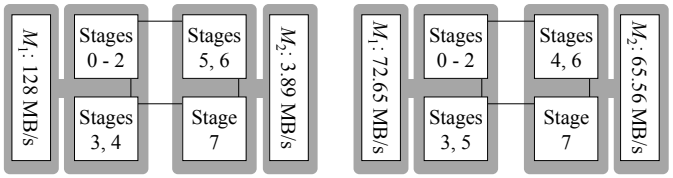
State-of-the-art task mapping schemes for many-core systems, e.g. [2]–[6], [13]–[24], concentrate on balancing the computational load among cores while avoiding bottlenecks in the communication infrastructure, assuming either shared or distributed (private) memories. This allows to achieve good results in systems where memory controllers cannot be saturated because they serve a small number of cores (as it is the case in many multi-core systems, such as in traditional symmetric multiprocessors (SMP), or in Intel's Xeon Phi™5110P [10]), or because the individual cores are too slow to saturate the memory controllers (e.g. the P54C cores of Intel's Single-Chip Cloud Computer [25]). The task mapping schemes [3], [4], [24] are similar to MOMA they target many-core systems with distributed memories and complex parallel applications. However, their focus is on balancing computation and communication among cores, while MOMA takes computation, communication, and bandwidth requirements into account. Hence, we compare MOMA to these schemes.

StreamIt-based software pipelines [11] can be mapped to embedded multi-core systems considering CPU load and on-chip scratch-pad memory [7]. However, this work aims at maximizing the throughput by reducing DMA traffic to off-chip DRAM and does not face a saturation of these units due a small numbers of cores. In contrast to this, we tackle scenarios with a large number of cores where this must be taken into account.

Design-time placement strategies: based on offline estimations of the workload, design-time strategies place the memory controllers to balance the accesses among them, e.g. [8]. Assuming static task mappings and that memory accesses are predictable at design-time, the physical location of memory controllers, and thus the assignment of cores (and consequently of tasks) to memory controllers is optimized. However, such strategies cannot provide good results in scenarios where the applications as well as their individual memory requirements are unpredictable at design time, e.g. because they depend on user interactions or on the input data.

Request optimization strategies: Memory controllers can schedule requests in a way that improves their throughput [9] by grouping requests. However, these strategies cannot balance the requests across multiple memory controllers, and thus such approaches cannot account for scenarios where some memory controllers are saturated due to unfavorable task mappings, even though the total bandwidth requirements of tasks does not exceed the combined bandwidth constraints of all memory controllers.

To summarize, state-of-the-art task mapping schemes aim at systems with a small number of cores where memory controllers cannot be saturated. However, this is an issue in systems with a large number of cores and thus, these scenarios must be addressed. Existing approaches to avoid the saturation of memory controllers require that the memory accesses are predictable at design-time, or aim at optimizing the performance of individual memory controllers without balancing the load among memory controllers. Orthogonal to this, we propose to incorporate bandwidth requirements into task mapping so that the load is balanced among memory controllers.




(a) Throughput: 10.71 iterations/second (b) Throughput: 19.1 iterations/second

Fig. 1: Simple system with two memory islands / four cores. Balancing computational requirements (a) saturates memory controller $M_1$ and leads to a significantly reduced throughput as compared to (b).

## IV. Motivation

We motivate by discussing an example of task mapping (for the purpose of balancing computations among cores) that puts the memory controllers in saturated operation and causes a severe throughput degradation. For illustration, let us consider a system with two memory islands of 2 cores each, and each memory controller with bandwidth constraint of 128 MB/s[1]. We map a software-pipelined application ("Object Tracking") with 8 stages. Table I lists their *computational requirements* and *memory requirements*.

| Stage | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| Computational requirements [ms] | 24.3 | 11.7 | 13.2 | 22.9 | 26.2 | 23.8 | 26.9 | 49.3 |
| Memory requirements [MB] | 0.18 | 0.23 | 0.46 | 5.43 | 5.65 | 0.17 | 0.17 | 0.02 |

TABLE I: "Object Tracking": Computational demands and memory requirements of the 8 stages.

To balance the *computational requirements*, one may map Stages 0 to 2 to Core 0, stages 3-4 to Core 1, stages 5-6 to Core 2, and stage 7 to Core 3. If the memory controllers of both memory islands were not saturated, a throughput of 20.04 iterations/second could be achieved. However, the mapping results in a load of 11.95 MB/iteration and 0.36 MB/iteration for the memory controllers, respectively. Due to their bandwidth constraint of 128 MB/s, one memory controller is saturated at approx. 10.71 iterations/second, which limits the application's throughput to this level. In order to account for the bandwidth constraint of memory controllers, Stages 0 to 2 may be mapped to Core 0, 3 and 5 to Core 1, 4 and 6 to Core 2, and stage 7 to Core 3, which results in a total memory requirement of 6.78 MB/iteration and 6.12 MB/iteration for the memory controllers, respectively. This mapping allows to achieve a throughput of approx. 19.1 iterations/second (limited by the computational load on Core 2), which causes a load of 72.65 MB/s and 65.56 MB/s for the memory controllers, which is below their bandwidth constraint and thus, they are not saturated. This corresponds to an increased throughput by approx. 76%. Figure 1 shows these mappings and the resulting throughputs.

Consequently, it is crucial that task mapping *jointly* balances the computational requirements and the load of memory controllers. Otherwise, a saturation of memory controllers and a significantly reduced throughput can be the result. This problem worsens with a growing number of cores per memory island.

[1]We chose a low bandwidth constraint for this example to illustrate the problem in a simplified way with four cores. However, the problem arises equally in systems with fast memory controllers where each memory controller has to serve a multitude of cores.

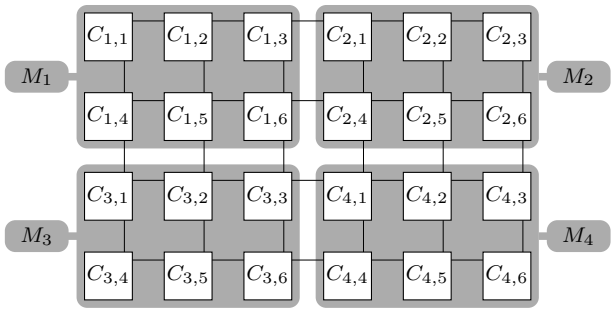



Fig. 2: Example of a target architecture showing four memory islands, each island $I_i$ containing 6 cores $C_{i,j}$ and one memory controller $M_i$. Cores are connected via a network-on-chip.

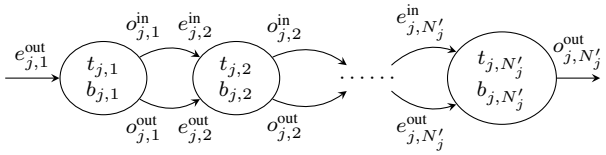



Fig. 3: The model of software pipelines.

## V. System Model and Problem Definition

In the following, we detail our system model and define the mapping problem formally.

### A. Hardware Model

We consider that there are $V$ memory controllers $\mathbf{M} = \{M_1, M_2, \ldots, M_V\}$, and each controller serves the $Q$ cores of the corresponding memory island $I_i \in \mathbf{I} = \{I_1, I_2, \ldots, I_V\}$, such that there are $Q \cdot V$ cores in total in the system. Every core $C_{i,j}$ is identified by pair $(i, j)$, where $i = 1, 2, \ldots, V$ represents the memory islands to which it belongs, and $j = 1, 2, \ldots, Q$ the index of the core inside its memory island. Figure 2 shows an example of the architecture.

We consider that each memory controller $M_i$ has a bandwidth constraint of $B_i$ and a *remaining* bandwidth constraint of $B'_i$, such that when no stages are mapped then $B'_i = B_i$ holds. When mapping stage $S^j_h$ from pipeline $P_j$ to memory island $I_i$, the value of $B'_i$ is updated by subtracting the bandwidth requirement $b_{j,h}$.

Similarly to $B'_i$, $Q'_i$ denotes the number of *free cores* (i.e. cores where no pipeline stage has been mapped to) of memory island $I_i$. When $n$ stages of a pipeline are mapped to island $I_i$, then $n$ is subtracted from $Q'_i$. Obviously, $Q'_i \geq 0$ holds, because only up to $Q$ fused stages can be mapped to an island.

### B. Pipeline Model

In the context of this paper, every application $j$ forms a software pipeline $P_j$ with $N'_j$ stages. Stage $i$ of pipeline $P_j$ is denoted as $S^j_i$. Per iteration, each stage $S^j_i$ has a *computational requirement* of $t_{j,i}$ (unit: seconds) and a *bandwidth requirement* of $b_{j,i}$ (unit: MB/s). When stage $S^j_{i-1}$ is mapped to the same memory island as $S^j_i$, it takes $e^{\text{in}}_{j,i}$ time for stage $S^j_i$ to receive the data from stage $S^j_{i-1}$. In contrary, when stage $S^j_{i-1}$ is mapped to a different memory island than $S^j_i$, it takes $e^{\text{out}}_{j,i}$ time for stage $S^j_i$ to receive the input data from stage $S^j_{i-1}$. Similarly, $o^{\text{in}}_{j,i}$ represents the time to send the output data to stage $S^j_{i+1}$ when both stages are mapped to the same memory island, and $o^{\text{out}}_{j,i}$ corresponds to the time required for sending the output data when $S^j_i$ and $S^j_{i+1}$ are mapped to different memory islands. Figure 3 illustrates the model.

Such a model is based on the assumption that $o^{\text{in}}_{j,i-1}$ and $e^{\text{in}}_{j,i}$ correspond to exchanging pointers, given that memory can be accessed in a shared manner within a memory island. Likewise, when $S^j_{i-1}$ and $S^j_i$ communicate across memory islands, $o^{\text{out}}_{j,i-1}$ and $e^{\text{out}}_{j,i}$ correspond to transferring the data between memory controllers, which requires considerably more time.

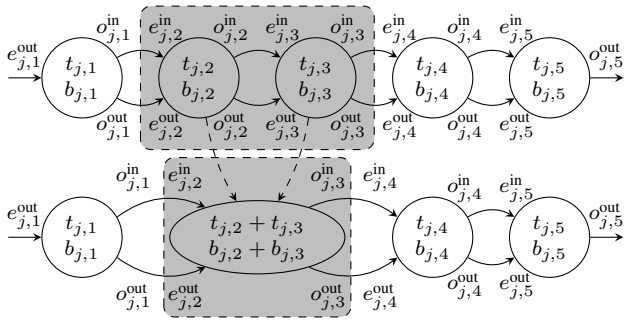



Fig. 4: Example of fusing pipeline stages.

Consecutive stages can be *fused* to form a single stage, as shown in Figure 4. When $S^j_{i-1}$ and $S^j_i$ are fused, the resulting computational requirement and bandwidth requirement, after fusion, are $t_{j,i-1} + t_{j,i}$ and $b_{j,i-1} + b_{j,i}$, respectively. Moreover, $o^{\text{in}}_{j,i-1}$, $o^{\text{out}}_{j,i-1}$, $e^{\text{in}}_{j,i}$ and $e^{\text{out}}_{j,i}$ are not considered, because $S^j_{i-1}$ and $S^j_i$ then form a single stage and thus no data exchange over the on-chip network is necessary.

For simplicity, the resulting number of stages after fusion for pipeline $P_j$ is denoted as $N_j$, such that $N_j \leq N'_j$. Furthermore, the *maximum response time* for a given mapping of pipeline $P_j$ is denoted $T_j$, and is computed by Equation (1). Finally, cores cannot be shared among fused stages, and thus $\sum_{j=1}^{K} N_j \leq Q \cdot V$ holds.

$$T_j = \max_{i=1,2,\ldots,N_j} \left\{ e^*_{j,i} + t_{j,i} + o^*_{j,i} \right\}$$

$$e^*_{j,i} = \begin{cases} e^{\text{out}}_{j,i} & \text{if } i = 1 \\ e^{\text{in}}_{j,i} & \text{if } S^j_{i-1}, S^j_i \in I_k \\ e^{\text{out}}_{j,i} & \text{if } S^j_{i-1} \in I_h \text{ and } \left\{ S^j_i \right\} \in I_k \end{cases} \tag{1}$$

where

$$o^*_{j,i} = \begin{cases} o^{\text{out}}_{j,i} & \text{if } i = N_j \\ o^{\text{in}}_{j,i} & \text{if } S^j_i, S^j_{i+1} \in I_k \\ o^{\text{out}}_{j,i} & \text{if } S^j_i \in I_k \text{ and } S^j_{i+1} \in I_\ell \end{cases}$$

We obtain the memory requirements for each stage by offline analysis. However, the bandwidth requirements $b_{j,i}$ of each stage cannot be accurately estimated in advance, given that the *throughput* of every application depends on the final mapping. Nevertheless, the bandwidth requirements are needed to avoid saturating the memory controllers. Therefore, we compute an *upper bound* for the bandwidth requirement $b_{j,i}$ for stage $S^j_i$, by dividing the memory requirement by the *maximum response time* $T_j$ for pipeline $P_j$, assuming that its throughput is not limited by bandwidth constraints and that no communication across memory islands is necessary.

### C. Problem Definition

For a system with $V$ memory controllers, where each memory controller manages $Q$ cores, the objective is to map $K$ software pipelines $\mathbf{P} = \{P_1, P_2, \ldots, P_K\}$ with weights $\mathbf{W} = \{w_1, w_2, \ldots, w_K\}$ (weights can be interpreted as user-defined priorities), such that the weighted system throughput is maximized using up to $Q \cdot V$ cores, i.e.,

$$\text{Maximize} \left\{ \sum_{j=1}^{K} \frac{w_j}{T_j} \right\} \qquad \text{such that } \sum_{j=1}^{K} N_j \leq Q \cdot V. \tag{2}$$

**Algorithm 1** MOMA Algorithm

**Input:** The information of the system and the pipelines to map;
**Output:** A mapping to maximize the overall system performance;
1: Execute Phase 1;
2: **repeat**
3: Execute Phase 2;
4: Execute Phase 3;
5: **until** All stages of all pipelines are mapped onto cores
6: **return** The mapping of all stages from all pipelines;

Intuitively, the stages mapped to a memory island $I_i$ suffer from degraded performance when the remaining bandwidth constraint $B'_i$ becomes negative, i.e. $B'_i < 0$. Consequently, in addition to find a solution for the goal and constraint of Equation (2), the MOMA algorithm aims at avoiding this by balancing the load among memory controllers. Finding a perfectly balanced memory bandwidth assignment is a *NP-complete* problem in a strong sense, which can be easily reduced from the 3-PARTITION problem [26]. The design of our MOMA algorithm is aimed to derive heuristic approximated solutions in polynomial-time.

## VI. MOMA Algorithm

In this section, we describe our three-phase heuristic, MOMA, presented in Algorithm 1, that finds a solution for Equation (2), i.e., it solves the defined problem. As an overview of MOMA, an initial solution is derived (Phase 1), and all stages have been mapped with the objective of minimizing a saturation of memory controllers (Phase 2), and minimizing the degradation of throughput that results from mapping communicating stages to multiple memory islands while not saturating the controllers (Phase 3).

### A. Phase 1: Initial solution

In this phase, an initial solution based on the algorithm presented by [24] is obtained, which solves two problems: (a) it distributes the cores among the applications, and then, given the number of cores for each application, (b) it fuses their stages so that their throughput is maximized. For the completeness of this paper, the algorithm proposed in [24] is summarized in Appendix A. The initial solution neglects the bandwidth constraint of the memory controllers and the overhead when stages communicate that are mapped to different memory islands. It should be noted that our approach is not tied to this algorithm; other algorithms to obtain these initial fusions could be used as well.

The *objective* of this phase is to consider the fusion of the stages individually for all pipelines, which will not be modified from this point on. Thus, for the rest of this paper, each pipeline $P_j$ will consist of $N_j$ fused stages, and constraint $\sum_{j=1}^{K} N_k \leq Q \cdot V$ is satisfied. Given that the optimal solution that considers all constraints might result in different fusions and cores per application, no further modifications to these fusions implies that no optimal solution is achievable, but the complexity of the problem is considerably reduced. Phase 1 thus gives a starting point for phases 2 and 3, namely the fusions and number of cores for each application. Then, as explained in Section V-B, the upper bound for the bandwidth requirements $b_{j,i}$ for all stages can be computed based on the memory requirements for each stage and the *maximal response time* of each pipeline.

Before proceeding to phase 2, three ordered tables are built to be used in phases 2 and 3.

*a)* **Memory Controller Information Table (MCI):** Each row in this table holds the memory island identifier $I_i$, the number of free cores $Q'_i$ of each memory island $I_i$ and the remaining memory bandwidth constraint $B'_i$ of the corresponding memory controller $M_i$, for all $i = 1, 2, \ldots, V$. The table is ordered in a decreasing manner with respect to $B'_i$. As explained in Section V-A, the values of $B'_i$ and $Q'_i$ are updated when stages are mapped to $I_i$, after which the corresponding row is reordered (the entire table does not need to be re-sorted, because only this row changes). When memory island $I_i$ has no more free cores, i.e. $Q'_i = 0$, the memory controller $M_i$ is removed from the table. An example for this table is presented in Table IIa.

| $\mathbf{I}$ | $\mathbf{B'}$ | $\mathbf{Q'}$ |
|---|---|---|
| $I_1$ | $B'_1$ | $Q'_1$ |
| $I_2$ | $B'_2$ | $Q'_2$ |
| $I_3$ | $B'_3$ | $Q'_3$ |
| $\vdots$ | $\vdots$ | $\vdots$ |
| $I_V$ | $B'_V$ | $Q'_V$ |

(a) Table *MCI*

| $\mathbf{P}$ | $\sum \mathbf{b}$ |
|---|---|
| $P_1$ | $\sum_{i=1}^{N_1} b_{1,i}$ |
| $P_2$ | $\sum_{i=1}^{N_2} b_{2,i}$ |
| $P_3$ | $\sum_{i=1}^{N_3} b_{3,i}$ |
| $\vdots$ | $\vdots$ |
| $P_K$ | $\sum_{i=1}^{N_K} b_{K,i}$ |

(b) Table *RB*

| $\mathbf{P}$ | $\mathbf{S}$ | $\mathbf{b}$ |
|---|---|---|
| $P_1$ | $S_1^1$ | $b_{1,1}$ |
| $P_1$ | $S_2^1$ | $b_{1,2}$ |
| $P_2$ | $S_1^2$ | $b_{2,1}$ |
| $\vdots$ | $\vdots$ | $\vdots$ |
| $P_K$ | $S_{N_K}^K$ | $b_{K,N_K}$ |

(c) Table *SSB*

TABLE II: Examples for Tables *MCI* (with $B'_1 \geq B'_2 \geq \cdots \geq B'_V$), *RB* (before phase 2, and $\sum_{i=1}^{N_1} b_{1,i} \geq \sum_{i=1}^{N_2} b_{2,i} \geq \cdots \geq \sum_{i=1}^{N_K} b_{K,i}$) and *SSB* (with $N_1 = 2$ and $b_{1,1} \geq b_{1,2} \geq b_{2,1} \geq \cdots \geq b_{K,N_K}$).

*b)* **Remaining Bandwidth requirement per-pipeline Table (RB):** Each row of RB contains the total bandwidth requirements of the non-mapped stages of each pipeline $P_j$, for all $j = 1, 2, \ldots, K$. The table is sorted with respect to the total bandwidth requirements in a decreasing order. When a stage is mapped to a core, its bandwidth requirements are subtracted from the corresponding row in the table, and this row is reordered (the table does not need to be re-sorted, because only this row changes). When all stages of pipeline $P_j$ have been mapped, the corresponding row is removed from the table. An example for this table is presented in Table IIb.

*c)* **Single Stage Bandwidth Requirements Table (SSB):** This table contains the memory bandwidth requirements of every non-mapped stage of each pipeline $P_j$, for all $j = 1, 2, \ldots, K$. The table is ordered in a decreasing order with respect to the bandwidth requirement of each stage. When a stage is mapped to a core, the corresponding row is removed from the table. No reordering is necessary. An example for this table is presented in Table IIc.

Phases 2 and 3 mostly focus on the first rows of each table, hence, for simplicity in presentation, the *parameters that represent the first row of each table* are denoted as $M_{\text{MCI}}^{\text{first}}$, $B'^{\text{first}}_{\text{MCI}}$, $Q'^{\text{first}}_{\text{MCI}}$, $P_{\text{RB}}^{\text{first}}$, $\sum b_{\text{RB}}^{\text{first}}$, $P_{\text{SSB}}^{\text{first}}$, $S_{\text{SSB}}^{\text{first}}$ and $b_{\text{SSB}}^{\text{first}}$.

### B. Phase 2: Mapping stages with high bandwidth requirements

For the fusions of *Phase 1*, there may be stages whose bandwidth requirements exceed the maximum remaining bandwidth constraint among all memory controllers, i.e., $\exists b_{j,h} > B'_i$ for all $i = 1, 2, \ldots, V$, $j = 1, 2, \ldots, K$ and $h = 1, 2, \ldots, N_j$. This means that mapping such stages to *any* memory island will saturate its memory controller as there will be at least one controller with $B'_i < 0$.

The *objective* of this phase is to balance the saturation caused by such stages among all memory controllers by using a *"largest bandwidth requirements first"* strategy in order to minimize the impact of the saturation of memory controllers. Otherwise, $-B'_i$ may be unnecessarily high, which could lead to a severely degraded throughput for all stages mapped to the corresponding memory island $I_i$.

Therefore, this phase starts by checking if the stage with the highest bandwidth requirement exceeds the remaining bandwidth of the memory controller with the highest remaining bandwidth, i.e., whether the value of $b_{\text{SSB}}^{\text{first}}$ is larger than the value of $B'^{\text{first}}_{\text{MCI}}$ (given that both tables are ordered, it is not necessary to check for all $i$, $j$ and

**Algorithm 2** MOMA: Phase 2

**Input:** Tables *MCI*, *RB* and *SSB*;
**Output:** Mapping of current *high bandwidth requirement* stages;
1: **while** $b^{\text{first}}_{\text{SSB}} > B'^{\text{first}}_{\text{MCI}}$ **do**
2: Map single stage $S^{\text{first}}_{\text{SSB}}$ to the island of controller $M^{\text{first}}_{\text{MCI}}$;
3: Update and re-order table *MCI*;
4: Update and re-order table *RB*;
5: Remove first row of table *SSB*;
6: Update parameters that represent the first row of each table;
7: **end while**
8: **return** Information about the mapped stages;

$h$). If this condition holds, this stage is mapped to the corresponding memory island, i.e., stage $S^{\text{first}}_{\text{SSB}}$ is mapped to $I^{\text{first}}_{\text{MCI}}$.

Once the stage is mapped, we subtract the value of $b^{\text{first}}_{\text{SSB}}$ from $B'^{\text{first}}_{\text{MCI}}$, decrease $Q'^{\text{first}}_{\text{MCI}}$ by 1 and reorder the first row of table *MCI*. Similarly, we also subtract $b^{\text{first}}_{\text{SSB}}$ from $\sum b^{\text{first}}_{\text{RB}}$ and reorder the first row of table *RB*. Finally, we remove the first row of table *SSB*, and update all parameters that represent the first row of each table.

If there are no more stages in table *SSB*, the mapping is completed. If there are still stages in table *SSB*, we repeat the process until $b^{\text{first}}_{\text{SSB}}$ is smaller than $B'^{\text{first}}_{\text{MCI}}$, and then proceed to Phase 3. A pseudo-code for Phase 2 is presented in Algorithm 2.

### C. Phase 3: Mapping pipelines with highest bandwidth requirements

This phase aims at mapping consecutive stages to a memory island, such that the *used bandwidth* of its memory controllers is maximized without exceeding the bandwidth constraint. Stages are mapped considering both bandwidth requirements and the communication latency when communicating between memory islands.

We focus on the pipeline with the highest bandwidth requirement for non-mapped stages and the highest remaining bandwidth constraint, i.e., pipeline $P^{\text{first}}_{\text{RB}}$ and controller $M^{\text{first}}_{\text{MCI}}$, respectively. For simplicity in presentation, we consider $P_j = P^{\text{first}}_{\text{RB}}$ and denote $N_j^{\text{non-mapped}}$ as the number of non-mapped stages of pipeline $P_j$.

We evaluate all combinations of mapping $1, 2, \ldots, \min\left\{N_j^{\text{non-mapped}}, Q'^{\text{first}}_{\text{MCI}}\right\}$ consecutive stages of pipeline $P_j$ to the memory island of controller $M^{\text{first}}_{\text{MCI}}$. After Phase 2, at least one combination that requires less bandwidth than $B'^{\text{first}}_{\text{MCI}}$ exists, e.g., any single stage $b_{j,h}$ for $h = 1, 2, \ldots, N_j^{\text{non-mapped}}$. Furthermore, we only consider the combinations of consecutive stages of pipeline $P_j$ with less or equal bandwidth requirements than $B'^{\text{first}}_{\text{MCI}}$. Therefore, when checking the combinations of mapping stages $S^j_h$ to $S^j_\ell$, if $\sum_{n=h}^{\ell} b_{j,n} > B'^{\text{first}}_{\text{MCI}}$, there is no need in considering the rest of the combinations that start on $S^j_h$.

In order to consider $B'_i$ as well as $e^{out}_{j,s}$ and $o^{out}_{j,s}$ when mapping stages to memory controllers, we introduce a window that contains different possible combinations with similar bandwidth requirements, and choose the combination that achieves the highest throughput for its application (affected by $e^{out}_{j,s}$ and $o^{out}_{j,s}$). For all possible combinations in the window $\left[B'^{\text{first}}_{\text{MCI}} - \frac{B'^{\text{first}}_{\text{MCI}}}{Y}, B'^{\text{first}}_{\text{MCI}}\right]$, where $Y$ is any integer (a design parameter) larger or equal than 1 that sets the size of the window. In other words, e.g., with $Y = 5$, we first consider only the combinations that require a bandwidth between 80% and 100% of $B'^{\text{first}}_{\text{MCI}}$. If there is no combination inside this window, we move the window to $\left[B'^{\text{first}}_{\text{MCI}} - 2\frac{B'^{\text{first}}_{\text{MCI}}}{Y}, B'^{\text{first}}_{\text{MCI}} - \frac{B'^{\text{first}}_{\text{MCI}}}{Y}\right]$, e.g., 60% and 80% of $B'^{\text{first}}_{\text{MCI}}$ with $Y = 5$. We repeat this up to $Y$ times, until we find at least one combination inside the window. Among all the combinations inside the evaluated window, we choose the one with the minimum *partial maximum response time* for consecutive stages $S^j_h$ to $S^j_\ell$, according to Equation (3), and map the selected stages to the memory island of controller $M^{\text{first}}_{\text{MCI}}$.

$$T'_j(h,\ell) = \begin{cases} e^{\text{out}}_{j,h} + t_{j,h} + o^{\text{out}}_{j,h} & \text{if } h = \ell \\ \max\left\{ \begin{array}{l} e^{\text{out}}_{j,h} + t_{j,h} + o^{\text{in}}_{j,h}, \\ \max\limits_{h<n<\ell} \left\{e^{\text{in}}_{j,n} + t_{j,n} + o^{\text{in}}_{j,n}\right\}, \\ e^{\text{in}}_{j,\ell} + t_{j,\ell} + o^{\text{out}}_{j,\ell} \end{array} \right\} & \text{if } h < \ell \end{cases} \quad (3)$$

Once stages $S^j_h$ to $S^j_\ell$ are mapped, we update table *MCI* by subtracting $\sum_{n=h}^{\ell} b_{j,n}$ from $B'^{\text{first}}_{\text{MCI}}$ and reorder the first row. For updating tables *RB* and *SSB*, some further considerations need to be taken. In case that $h = 1$ or $l = N_j^{\text{non-mapped}}$, we proceed in a similar fashion as done for Phase 2: we subtract $\sum_{n=h}^{\ell} b_{j,n}$ from $\sum b^{\text{first}}_{\text{RB}}$ and reorder the first row of table *RB*; and we remove all the rows that correspond to the mapped stages from table *SSB*.

However, when $h > 1$ and $l < N_j^{\text{non-mapped}}$, it implies that some stages are mapped to the memory island of controller $M^{\text{first}}_{\text{MCI}}$, while both their preceeding stages as well as their succeeding stages are mapped to different islands. When this happens, in order to correctly compute the minimum *partial maximum response time* for new iterations of Phase 3, we need to split pipeline $P_j$ into two sub-pipelines, one with stages $S^j_1$ to $S^j_{h-1}$ and another with stages $S^j_{\ell+1}$ to $S^j_{N_j}$. Failing to do this would give incorrect results for Equation (3) in new iterations of Phase 3. With the sub-pipelines, we can proceed to update table *RB* by removing the old pipeline and inserting the new sub-pipelines in the corresponding order. Similarly, we remove all stages from the previous pipeline from table *SSB* and insert the stages of the sub-pipelines, also in the corresponding order.

Once all three tables are updated, we also update all parameters that represent the first row of each table. If there are no more stages in table *SSB*, the mapping is completed. If there are still stages in table *SSB*, the algorithm continues to execute by returning to Phase 2, since after updating $B'^{\text{first}}_{\text{MCI}}$, now there may exist a single stage with a $b^{\text{first}}_{\text{SSB}}$ value larger than $B'^{\text{first}}_{\text{MCI}}$. A pseudo-code for Phase 3 is presented in Algorithm 3.

### D. Algorithmic Complexity

For notational simplicity, we denote $N_{\max} = \max\limits_{1 \le j \le K} N_j$. The time complexity for Phase 1 is for computing the initial solution and, using the algorithm from [24], is $O\left(\max\left\{QVN^2_{\max}, Q^2V^2K\right\}\right)$. The time complexity for Phase 2 and Phase 3 are $O\left(N_{\max}K\right)$ and $O\left(\max\left\{N^2_{\max}, Q^2\right\}\right)$, respectively. Given that Phase 2 and 3 can be executed up to $N_{\max}K$ times, the total time complexity for MOMA is $O\left(\max\left\{QVN^2_{\max}, Q^2V^2K, N^3_{\max}K, N_{\max}Q^2K\right\}\right)$.

## VII. Experimental Results

This section presents our experimental setup and discusses our experiments with respect to the achieved system throughput, the saturation of memory controllers, and the overhead. We compare our approach to DistRM [4] and AIAC [3] to show that a significant improvement over state-of-the-art runtime task mapping can be achieved by balancing the load of the memory controllers. Our proposed MOMA approach is orthogonal to design-time optimization strategies such as [8] and also orthogonal to request optimization strategies, e.g. [9] and could be combined with those approaches to complement them on system level.

### A. System Details and Implementation

**Adaption of OCM [24]:** To adapt OCM [24] for comparison with our MOMA approach from Section VI, cores are distributed to the applications and the stages are fused accordingly. Then, stage $S^1_1$ of pipeline $P_1$ is mapped to core $C_{1,1}$, stage $S^1_2$ of application $P_1$ is mapped to core $C_{1,2}$, ..., and stage $S^K_{N_K}$ of application $P_K$ is mapped to core $C_{V,Q}$.

**Algorithm 3** MOMA: Phase 3

**Input:** Tables *MCI*, *RB* and *SSB*;
**Output:** Mapping of stages for highest bandwidth pipeline;

$P_j \leftarrow P_{\text{RB}}^{\text{first}}$;
**for** $h = 1, 2, \ldots, \max\left\{N_j^{\text{non-mapped}}, Q'^{\text{first}}_{\text{MCI}}\right\}$ **do**
  **for** $\ell = h, h+1, \ldots, \max\left\{N_j^{\text{non-mapped}}, Q'^{\text{first}}_{\text{MCI}}\right\}$ **do**
    **if** $\sum_{n=h}^{\ell} b_{j,n} \leq B'^{\text{first}}_{\text{MCI}}$ **then**
      **for** $y = 1, 2, \ldots Y$ **do**
        **if** $1 - \frac{y}{Y} \leq \frac{\sum_{n=h}^{\ell} b_{j,n}}{B'^{\text{first}}_{\text{MCI}}} \leq 1 - \frac{y-1}{Y}$ **then**
          Window$[y] \leftarrow$ Append combination $\left[S_h^j, S_\ell^j\right]$;
        **end if**
      **end for**
    **end if**
  **end for**
**end for**
**for all** $y = 1, 2, \ldots Y$ **do**
  **if** Window$[y]$ is not empty **then**
    **for all** Elements in Window$[y]$ **do**
      Compute $T'_j(h, \ell)$ according to Equation (3);
    **end for**
    $\left[S_h^j, S_\ell^j\right] \leftarrow$ Element with minimum $T'_j(h, \ell)$;
    Map stages $\left[S_h^j, S_\ell^j\right]$ into controller $M_{\text{MCI}}^{\text{first}}$;
    Update and re-order table *MCI*;
    Update and re-order table *RB*;
    Remove rows of mapped stages from table *SSB*;
    Update parameters that represent the first row of each table;
    **return** Information about the mapped stages;
  **end if**
**end for**

**Implementation of the Applications:** We use several software-pipelined applications, as shown in Table III, since they are well-suited to form software pipelines. We manually parallelized them (using C/C++) and use MPI [12] for communication between stages.

**Obtaining Stage Parameters:** The parameters of each stage are obtained through offline analysis. To obtain $t_{j,i}$, we collect the difference of the CPU cycle counter before and after the computation of a stage. For this, each stage is run separately on a single Intel Core CPU at 1.2 GHz (Family 6, Model 23, Stepping 10), while the input data has been pre-computed and is stored in memory (the memory controller is not saturated). $b_{j,i}$ is measured by generating an extensive memory trace in a second profile run by using guard pages that allow to trace memory accesses accurately. Only first-time accesses to each memory page are considered to account for cache effects. This corresponds to an infinite cache size and thus can be considered as very conservative. In a real system, the memory accesses can be assumed to be higher than our measurements, thus the load of the memory controllers and thus the benefit of our MOMA algorithm may be higher. To obtain $o_{j,i}^{in}$, $o_{j,i}^{out}$, $e_{j,i}^{in}$, and $e_{j,i}^{out}$, the stages run on Intel's Single-Chip Cloud Computer [25] and transfer the output data of each stage using Intel's RCCE library. The experiments to obtain the parameters are repeated and their results are averaged across multiple runs.

Different sets of input data for "automotive", "h264ref', and "lame" allow to obtain varying computational requirements, communication demands, and memory accesses. Each set of input data is associated with one instance of the application. The input scenes for "automotive" are characterized by low, medium, and high traffic. The input video sequences for "h264ref" have resolutions of 352x240, 576x432, 640x480, 720x405, and 1080x720 pixels. Different command line options for "lame" create CBR and VBR output with bitrates of 8, 128, and 320.

| Name | Stages | Source | Input data |
|---|---|---|---|
| automotive | 21 | Algorithms from [27] | 3 different input scenes |
| h264ref | 4 | SPEC CPU 2006 [28] | 352x240, 576x432, 640x480, 720x405, 1080x720 resolution |
| lame | 4 | MiBench [29] | CBR/VBR bitrates 8, 128, 320 |
| PGP | 5 | MiBench [29] | input from benchmark suite |
| sphinx3 | 22 | SPEC CPU 2006 [28] | input from benchmark suite |

TABLE III: Benchmark applications.

### B. Experimental Setup and Scenario

**Setup:** We use a high-level many-core system simulator to simulate systems with an arbitrary number of cores and memory controllers. It simulates a network-on-chip similar to the one of Intel's Single-Chip Cloud Computer (SCC) [25], i.e. it transfers flits of 16 Bytes at 533 MHz, assumes a 16 KiB send/receive buffer per core, and uses a x/y routing strategy. Each core execute traces collected as described in Section VII-A, where each stage repeats a pattern of (a) receiving input data ($e_{j,i}^{in}$ or $e_{j,i}^{out}$), (b) computation and accessing memory ($t_{j,i}$ and $b_{j,i}$), (c) sending output data ($o_{j,i}^{in}$ or $o_{j,i}^{out}$) . The memory controllers are configured for certain bandwidth constraints and perform FIFO scheduling for requests. Our simulator is written in C++ and runs on a system with 4 Six-Core AMD Opteron 8431 processors at 2.4 GHz, running a 2.6.37 Ubuntu Linux.

**Scenario:** The applications detailed in Table III are spawned repeatedly until the total number of stages (not considering fusions) exceeds the number of cores by a factor of $\geq 3$. This number is chosen arbitrarily in order to achieve considerable system load. Our experiments are performed for three different systems with 128, 512, and 1024 cores. All systems contain 16 memory islands (as Intel's Xeon Phi 5110P [10]), thus each memory controller serves 8, 32, and 64 cores, respectively. Each memory controller has a bandwidth constraint of 128 GiB/s, similar to the bandwidth constraint of many state-of-the-art GDDR5 controllers. For the system with 128 cores, the cores cannot saturate the memory controllers due to their limited performance. This is similar to Intel's Single-Chip Cloud Computer [25], where the comparably slow individual cores cannot saturate a memory controller [30].

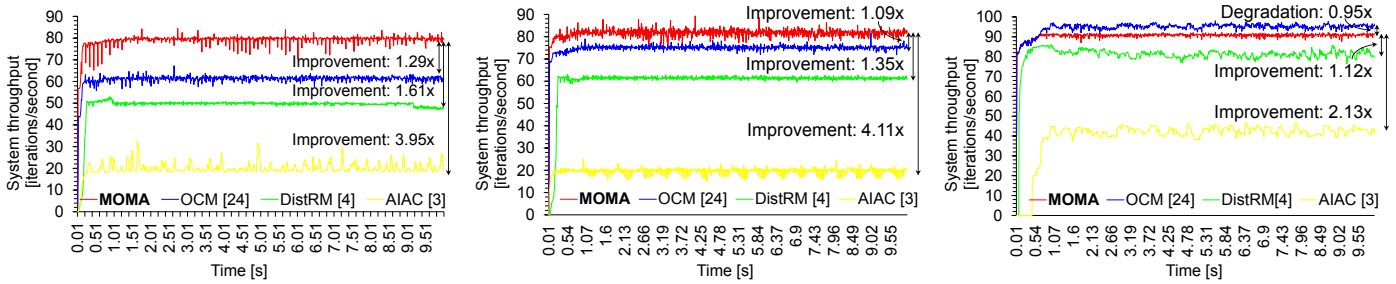
### C. Throughput and Memory Controller Load

Figure 5 (a) compares the throughput of our proposed MOMA scheme with OCM [24] (i.e. our Phase 1, as described above), DistRM [4], and AIAC [3], for 1024 cores over a period of 10 seconds. The throughput can be increased by approx. 1.29x, 1.61x, and 3.95x over these schemes, respectively, by reducing the saturation of memory controllers. Figure 5 (b) shows a similar result for 512 cores, however, the improvement over OCM [24] and DistRM [4] is smaller. The reason is that for a system size of 512 cores, each memory controller serves 32 cores (as compared to 64 cores for 1024 core systems), the memory controllers are less often saturated. Finally, in a 128-core system, the throughput achieved by MOMA is less than when using OCM [24] and the improvement over the other DistRM [4] and AIAC [3] is smaller, as depicted in Figure 5 (c), as the memory controllers are not saturated because their bandwidth constraint exceeds the bandwidth requirements.

Table IV summarizes the relative improvements achieved when

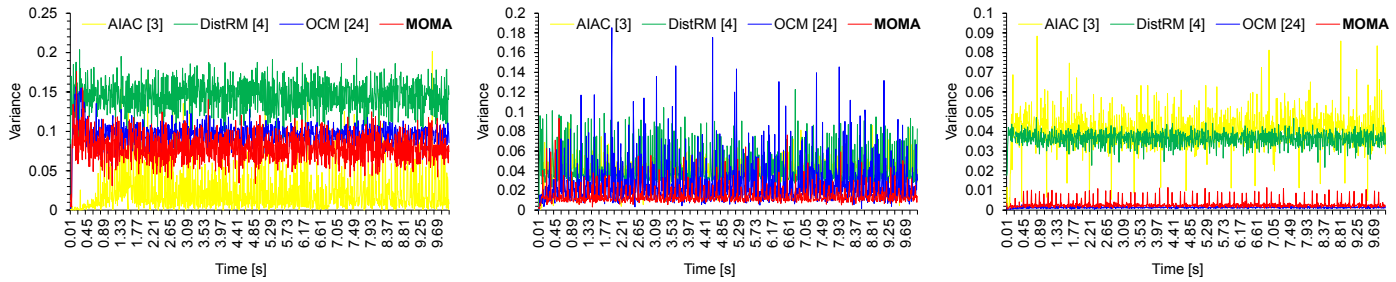
| # Cores | OCM [24] | DistRM [4] | AIAC [3] |
|---|---|---|---|
| 128 | -4.7% | 11.6% | 113.2% |
| 512 | 8.9% | 35.4% | 311.3% |
| 1024 | 29.3% | 61.5% | 295.0% |

TABLE IV: Relative throughput of MOMA compared to OCM [24], DistRM [4], and AIAC [3]. Negative values correspond to lower throughputs.




(a) 1024 cores (b) 512 cores (c) 128 cores

Fig. 5: Comparison of the system throughput achieved by MOMA and several task mapping schemes in systems with 1024, 512, and 128 cores and 16 memory controllers.




(a) 1024 cores (b) 512 cores (c) 128 cores

Fig. 6: Variance between the load of the 16 memory controllers that results from using MOMA and several task mapping schemes, for 1024, 512, and 128 cores over a period of 10 seconds.

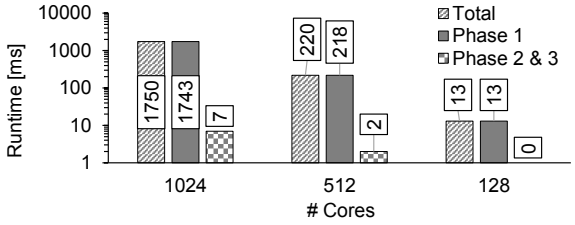



Fig. 7: Runtime overhead of the MOMA algorithm for 1024, 512, and 128 cores (16 memory islands), which is dominated by Phase 1.

using MOMA compared to the state-of-the-art schemes. From these observations, we conclude that MOMA effectively balances the bandwidth requirements to avoid a saturation of memory controllers.

Figure 6 illustrates the variance of the load of the memory controllers over a period of 10 seconds for systems with 1024, 512, and 128 cores. For 1024 cores, AIAC [3] has the lowest variance between memory controllers because it achieves a low throughput and thus, the load of the memory controllers is low. As compared to OCM [24] and to DistRM [4], MOMA achieves a significantly lower variance, and thus a better balance, of the load of memory controllers. The load balance is responsible for the increased throughputs for systems with 1024 and 512 cores. As illustrated in Figure 5 (c) and 6 (c), the throughput achieved when using MOMA is lower than when using OCM [24]. The reason is that the imbalanced load of the memory controllers that results from OCM [24] does not saturate a memory controller. As a result, the balancing performed by MOMA worsens the throughput slightly. Consequently, MOMA is limited to systems with a large number of cores and scenarios of memory-intensive applications. In such scenarios, it can greatly improve the system throughput over the state of the art.

### D. Overhead

Figure 7 shows the runtime overhead of our MOMA algorithm for 1024, 512, and 128 cores. The runtime is purely dominated by Phase 1, which is the initial solution as obtained by the algorithm taken from [24] (other algorithms to obtain the initial fusions could also be used for Phase 1). The runtimes of Phases 2 and 3, which refine this initial solution to balance the bandwidth requirements among the memory controllers, is very small and thus, we consider the runtime overhead of our algorithm well tolerable.

## VIII. Conclusion

We have presented MOMA, a task mapping scheme for memory-intensive software-pipelined applications. MOMA computes mappings not only based upon the computation and on-chip communication requirements of tasks, but also *jointly* based upon their off-chip memory accesses. This is particularly important in systems that comprise a large number of cores since the number and bandwidth of memory controllers is limited and thus, each memory controller must serve many cores. Experiments show that MOMA improves the system throughput significantly by avoiding a saturation of memory controllers that would result when applying state-of-the-art task mapping schemes. Our approach is therefore a necessary contribution to the scalability of upcoming many core systems.

## IX. Acknowledgements

This work was partly supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre “Invasive Computing” (SFB/TR 89). This work is supported in parts by the Baden Württemberg MWK Juniorprofessoren-Programm. This work was supported in parts by Intel Corp. and Intel Labs Braunschweig.

## REFERENCES

[1] S. Borkar, "Thousand Core Chips: A Technology Perspective," in *Proc. ACM/IEEE Design Automation Conf. (DAC)*, 2007.

[2] J. Jahn and J. Henkel, "Pipelets: Self-Organizing Software Pipelines for Many Core Architectures," in *Proc. ACM/IEEE Des., Autom. and Test in Europe (DATE)*, 2013.

[3] J. M. Bahi *et al.*, "Dynamic load balancing and efficient load estimators for asynchronous iterative algorithms," *IEEE Trans. Parallel Distrib. Syst.*, vol. 16, pp. 289–299, April 2005.

[4] S. Kobbe *et al.*, "DistRM: Distributed Resource Management for On-Chip Many-Core Systems," in *Proc. IEEE/ACM/IFIP Int. Conf. on Hardware/Software Codes. and Syst. Synth (CODES+ISSS)*, ser. CODES+ISSS '11, 2011.

[5] E. Carvalho *et al.*, "Heuristics for Dynamic Task Mapping in NoC-based Heterogeneous MPSoCs," in *IEEE/IFIP Int. Workshop on Rapid System Prototyping (RSP)*, 2007.

[6] K. Klues *et al.*, "Processes and Resource Management in a Scalable Many-Core OS," in *USENIX Workshop on Hot Topics in Parallelism (HotPar)*, 2010.

[7] H. Lee *et al.*, "Dynamic scheduling of stream programs on embedded multi-core processors," in *Int. Symp. on Hardw./Softw. Codes. and Syst. Synth. (CODES+ISSS)*, 2012.

[8] D. Abts *et al.*, "Achieving Predictable Performance through Better Memory Controller Placement in Many-Core CMPs," in *Proc. Int. Symposium on Computer Architecture (ISCA)*, 2009.

[9] Y. Kim *et al.*, "ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers," in *Proc. Int. Conf. on High-Performance Computer Architecture (HPCA)*, 2010.

[10] http://www.intel.de/content/dam/www/public/us/en/documents/product-briefs/xeon-phi-datasheet.pdf.

[11] William Thies and others, "StreamIt: A Language for Streaming Applications," in *Proc. Int. Conf. on Compiler Const. (ICCC)*, 2001.

[12] http://www.open-mpi.org/software/ompi/v1.6/.

[13] C. Lee *et al.*, "A Task Remapping Technique for Reliable Multi-core Embedded Systems," in *Proc. Int. Conf. on Hardware/Software Code. and Syst. Synth. (CODES+ISSS)*, 2010.

[14] J. Stender *et al.*, "Mobility-based runtime load balancing in multi-agent systems," in *Proc. Int. Conf. on Software Engineering and Knowledge Engineering (SEKE)*, 2006.

[15] C.-K. Luk *et al.*, "Qilin: Exploiting Parallelism on Heterogeneous Multiprocessors with Adaptive Mapping," in *Proc. IEEE/ACM Int. Symp. on Microarch. (MICRO)*, 2009.

[16] M. A. Faruque *et al.*, "ADAM: Run-time Agent-based Distributed Application Mapping for On-chip Communication," in *Proc. ACM/IEEE Design Automation Conf. (DAC)*, 2008, pp. 760–765.

[17] C. Chou *et al.*, "Incremental Run-time Application Mapping for Homogeneous NoCs With Multiple Voltage Levels," in *Proc. Int. Conf. on Hardware/Software Codes. and System Synth. (CODES+ISSS)*, 2007.

[18] C.-L. Chou *et al.*, "Contention-aware Application Mapping for Network-on-Chip Communication Architectures," in *IEEE Int. Conf. on Computer Design (ICCD)*, 2008.

[19] M. Rajagopalan *et al.*, "Thread scheduling for Multi-Core Platforms," in *USENIX HotOS*, 2007.

[20] A. Snavely and D. M. Tullsen, "Symbiotic Jobscheduling for a Simultaneous Multithreaded Processor," in *ASPLOS*, 2000.

[21] V. Nollet *et al.*, "Centralized Run-time Resource Management in a Network-on-Chip Containing Reconfigurable Hardware Tiles," in *Proc. ACM/IEEE Design, Automation and Test in Europe (DATE)*, 2005.

[22] T. Li *et al.*, "Efficient operating system scheduling for performance-asymmetric multi-core architectures," in *Proc. of the ACM/IEEE Conf. on Supercomputing (ICS)*, 2007.

[23] K. Lakshmanan *et al.*, "Partitioned Fixed-Priority Preemptive Scheduling for Multi-core Processors," in *ECRTS*, 2009.

[24] J. Jahn *et al.*, "Optimizations for Configuring and Mapping Software Pipelines in Many Core Systems," in *Proc. ACM/IEEE Design Automation Conf. (DAC)*, 2013.

[25] J. Howard *et al.*, "A 48-Core IA-32 Message-Passing Processor with DVFS in 45nm CMOS," in *Proc. IEEE Int. Solid-Stat Circuits Conference (ISSCC)*, 2010.

[26] M. R. Garey and D. S. Johnson, *Computers and intractability: A guide to the theory of NP-completeness*. W. H. Freeman and Co., 1979.

[27] P. Azad *et al.*, *Computer Vision - Principles and Practice*. Elektor Electronics, 2008.

[28] J. L. Henning, "SPEC CPU2006 Benchmark Descriptions," *SIGARCH Comput. Archit. News*, vol. 34, no. 4, Sep. 2006.

[29] M. R. Guthaus *et al.*, "MiBench: A Free, Commercially Representative Embedded Benchmark Suite," in *Proc. IEEE Workshop on Workload Characterization (WWC-4)*, 2001.

[30] N. Melot *et al.*, "Investigation of Main Memory Bandwidth on Intel Single-Chip Cloud Computer," in *Proc. Intel MARC Symposium*, 2011.


## APPENDIX

### A. Distributing cores and fusing stages [24]

In the following, we describe the algorithms to distribute cores among applications and, given a number of cores for each application, to fuse the stages so that the throughput of each application is maximized, as proposed by [24]. These algorithms, however, do not consider the bandwidth constraint of memory controllers, nor the overhead when communicating stages are mapped to different memory islands. For our model, this means that $B_i \to \infty$ for all $i = 1, 2, \ldots, V$, and that $e^{\text{out}}_{j,h} = e^{\text{in}}_{j,h}$ and $o^{\text{out}}_{j,h} = o^{\text{in}}_{j,h}$ for all $h = 1, 2, \ldots, N'_j$ stages in all $j = 1, 2, \ldots, K$ pipelines. Under such simplifications, the problem can be solved optimally by two dynamic programming algorithms, where the first algorithm $\mathcal{A}_1$ considers all possible fusions for each pipeline, and the second algorithm $\mathcal{A}_2$ decides how many cores are assigned to each application such that the overall weighted system performance is maximized.

The first algorithm $\mathcal{A}_1$ builds table $R_j(i, n)$, which stores the minimum *maximal response time* for a *sub-pipeline* that considers stages $S^j_1$ to $S^j_i$ for pipeline $P_j$, using at most $n$ cores. The initial conditions for $R_j(i, 0)$ and $R_j(i, 1)$ are

$$\begin{aligned} R_j(i,0) &= \infty && \forall i = 1 \ldots N'_j \\ R_j(i,1) &= e^{\text{in}}_{j,1} + o^{\text{in}}_{j,i} + \sum_{h=1}^{i} t_{j,h} && \forall i = 1 \ldots N'_j. \end{aligned} \tag{4}$$

Moreover, by defining function $\text{minmax}RF_j(i, n)$ as

$$\text{minmax}RF_j(i,n) = \min_{n-1 \le \ell < i} \left\{ \max \left\{ R_j(\ell, n-1), e^{\text{in}}_{j,\ell+1} + o^{\text{in}}_{j,i} + \sum_{h=\ell+1}^{i} t_{j,h} \right\} \right\}, \tag{5}$$

the recursive function $R_j(i, n)$ with $n \ge 2$ is

$$R_j(i,n) = \begin{cases} R_j(i, n-1) & i < n \\ \min\{R_j(i, n-1), \text{minmax}RF_j(i,n)\} & i \ge n. \end{cases} \tag{6}$$

The algorithm is executed for all $j = 1, 2, \ldots, K$ pipelines for $i = 1, 2, \ldots N'_j$ and $n = 1, 2, \ldots \max\{N'_j, Q \cdot V\}$. Each entry $R_j(N'_j, n)$, contains the minimum *maximal response time* of pipeline $P_j$ using at most $n$ cores.

Based on $R_j(N'_j, \max\{N'_j, Q \cdot V\})$, the second algorithm $\mathcal{A}_2$ decides how many cores are allocated to each application such that the overall weighted system performance is maximized. The algorithm builds a table $G(j, n)$ to store the maximum weighted system performance for the first $j$ pipelines on at most $n$ cores. When there is no feasible solution, i.e. $j > n$, entries $G(j, n)$ are set to $-\infty$. The initial conditions for $G(1, n)$ are

$$G(1,n) = \frac{w_1}{R_1(N'_1, n)} \qquad \forall n = 1, 2, \ldots, Q \cdot V. \tag{7}$$

Moreover, the recursive function for $G(j, n)$ with $j \ge 2$ is

$$G(j,n) = \begin{cases} -\infty & j > n \\ \max\limits_{j-1 \le \ell < n} \left\{ G(j-1, \ell) + \frac{w_j}{R_j(N'_j, n-\ell)} \right\} & j \le n \end{cases} \tag{8}$$

The algorithm is executed for all $j = 1, 2, \ldots, K$ and $n = 1, 2, \ldots, Q \cdot V$. Entry $G(K, Q \cdot V)$ contains the maximum weighted system performance for all pipelines using up to $Q \cdot V$ cores.