Pipelets: Self-Organizing Software Pipelines for Many-Core Architectures

Janmartin Jahn and Jörg Henkel
Chair for Embedded Systems (CES), Karlsruhe Institute of Technology (KIT), Germany
{jahn,henkel}@kit.edu

***Abstract*—We present the novel concept of Pipelets: self-organizing stages of software pipelines that monitor their computational demands and communication patterns and interact to optimize the performance of the application they belong to. They enable dynamic task remapping and exploit application-specific properties. Our experiments show that they improve performance by up to 31.2% compared to state-of-the-art when resource demands of applications alter at runtime as is the case for many complex applications.**

I. Introduction and Novel Contributions

Current and future computing systems comprise a rapidly increasing number of cores on a single chip to reap performance benefits from parallel processing. This trend, however, makes it difficult to efficiently employ all cores and the communication infrastructure between them (from now on, we refer to both as *resources*) as the performance is largely affected by *how* tasks are mapped to cores [17], [21]: When task mapping is decided upon at compile-time, it can hardly efficiently cope with situations where the usage of resources is hard to predict in the first place (as argued e.g. in [10], [11], [19]). Thus, *dynamic task mapping* maps tasks at runtime based on observed resource usage [6], [8]. Such approaches may provide good performance when the resource demands of tasks remain mainly unaltered at runtime. However, if they do significantly alter and if that cannot be predicted, performance penalties are most likely a consequence (Section II discusses an example). To account for such scenarios, *remapping* tasks, i.e. transferring the execution of a task from one core to another (e.g. [15]), is a viable solution to balance the resource demands at runtime [5], [19]. However, the problem of finding (and adapting) task mappings is NP-complete, which makes (potentially frequent) computations of mappings at runtime challenging. Consequently, a centralized controlling instance that decides about all task remappings may suffer from high computational overhead. Additionally, the communication overhead for monitoring the *system state* (i.e. the usage of resources and the resource demands of tasks) may become infeasible in systems with many cores.
A possible, scalable alternative are distributed approaches that focus on local decisions rather than on the entire system. However, it must be carefully chosen which observations are used, and balancing computation might impair communication (and vice versa) in systems that run multiple complex applications with potentially heavily-communicating tasks (i.e. the time required for their communication is significant).

To adress this issue, we present *Pipelets* as self-organizing stages of software pipelines. Software pipelines are a well-established means to achieve parallelism for a large class of applications (e.g. for stream-processing applications). They comprise stages that repeatedly compute *iterations*, where the output of one stage forms the input of its direct successor.

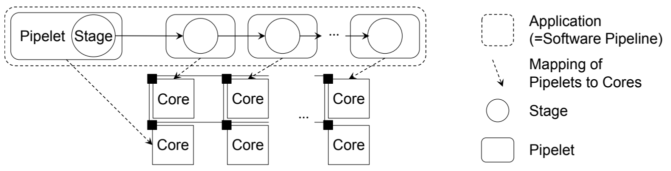
Pipelet
Stage
Core
Core
Core
Core
Core
Core
...
...
Application (=Software Pipeline)
Mapping of Pipelets to Cores
Stage
Pipelet

Fig. 1. Relationship of Pipelets to Applications, Tasks and Stages

Each stage is a task that may be mapped to an individual core. Pipelets interact in order to achieve self-organization, which is a powerful paradigm for managing complex systems [12] for the following reasons: (1) it allows for distributed decisions, (2) it shifts the responsibility of observing the relevant system state to the involved Pipelets, allowing them to exploit application knowledge, and (3) it therefore limits the computational and communication overhead The concept of self-organization is successfully employed in many instances, [14] presents a comprehensive survey.
Definition of Pipelets: A Pipelet is a task forming one stage of a software pipeline (see Figure 1) with the properties:

- It can remap between cores at runtime using existing task remapping mechanisms (Section IV-C and IV-D).
- It interacts with other Pipelets (Section IV-E).
- Pipelets aim at optimizing their application's performance when an established task mapping becomes inefficient (Section IV-C).

Pipelets can be created manually or by compilers extracting software pipelines from existing sequential programs, e.g. [7], [20], [23]. For the rest of this paper, we assume that all cores are connected via a Network-on-Chip.

Our novel contributions are:

1) We show how the new concept of Pipelets can adapt task mappings at runtime.
2) We demonstrate how Pipelets exploit application knowledge to estimate the impact of task remapping.
3) We show how Pipelets achieve scalability through self-organization.

II. Motivation

The following example of a pipelined visual object tracking application (8 stages) shows how altering resource demands caused by adding multiple tracked objects to the input scene may degrade the performance of a system and how task

$T_{i=0,1,2}$: Point of time

Task	T_0	T_1	T_2
A	24.3	35.3	35.3
B	11.7	54.3	54.3
C	13.2	31.3	31.3
D	22.9	20.8	20.8
E	26.2	11.7	11.7
F	23.8	13.2	13.2
G	26.1	23.1	23.1
H	49.3	21.2	21.2

(a) Iteration runtime [ms]

	T_0	T_1	T_2
Core 0	99%	100%	96%
Core 1	98%	27%	92%
Core 2	100%	30%	85%
Core 3	99%	18%	100%
Avg.	99%	44%	93%

(b) CPU utilization

T_0	T_1	T_2
20.04	8.27	17.70

(c) Application throughput [1/s]

Table 1. Characteristics of an exemplary scenario

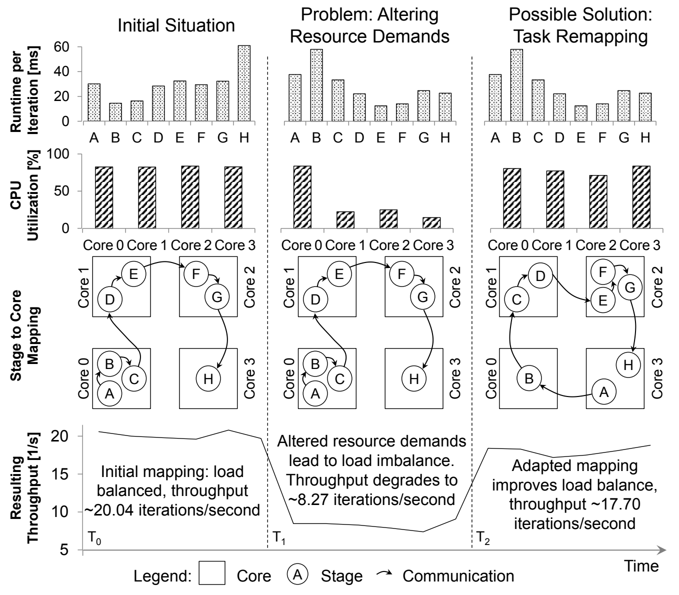

Fig. 2. Altering resource demands and resulting performance

remapping can improve this: Figure 2 shows this in a 4-core system. At T_1, the altering resource demands lead to a computational bottleneck on *'Core 0'* as the runtime stage B increases largely. Table 1 contains their detailed requirements and performance. The bottleneck arises because the initial mapping (that a provided good load balance at T_0) maps stages A, B and C to this core, and cores 1, 2 and 3 are now under-utilized. Therefore, the established mapping becomes inefficient. This problem may arise in turn for altered communication demands that may lead to congestions in the communication infrastructure. A possible solution is to remap the stages as done at T_2. This re-balances the computational load and increases the throughput to 17.7 iterations/second (which is slower than at T_0 as the total resource demands increased from 197 ms to 210 ms). In the following, we discuss why the state of the art does not provide efficient means to address these and similar scenarios.

III. Related Work

The related work can be grouped into *dynamic task mapping* (finding mapping based on resource usage) and *dynamic task remapping* (adapting them for altering resource demands).

Dynamic task mapping: contention-aware application mapping is proposed by [9] using offline profiling to find a number of near-optimal static mappings to choose from at runtime. The resource demands of tasks are assumed to remain unaltered at runtime. [10] presents distributed application mapping for Network-on-Chip architectures using task remapping in order to serve mapping requests. It aims at reducing the computational efforts to calculate task mapping and of the communication overhead required for collecting the system state. It does not aim at balancing the load when resource demands of tasks alter at runtime.

Dynamic task remapping: AIAC [4] balances the load of grid computing systems at runtime by exchanging workload of tasks with their nearest neighbor until the load is balanced. This can be applied to many-core systems with distributed memories by exchanging workload through remapping tasks between cores. However, AIAC does not target heavily communicating tasks and thus does not consider inter-task communication, leading to inferior throughput when applied to such applications. [17] and [21] present runtime load balancing for shared-memory architectures and are thus not well-suited for architectures with distributed memories. [11] remaps workload to nearest neighbors, but targets at optimizing thermal characteristics and not the system performance. [22] uses load balancing coordinators to carry out task remapping, but explicitly states that it cannot fully answer which tasks should be remapped.

To summarize, the state of the art does not sufficiently address the scenario in many-core systems when resource demands alter significantly at runtime: in this scenario, dynamic task mapping can hardly provide a balanced load, and existing task remapping techniques do not take inter-task communication into account or they assume shared memories.

IV. Pipelets

This sections details how Pipelets adapt task mappings using the concept of self-organization to avoid scalability issues of centralized approaches. Figure 3 shows a schematic view. The means of self-organization are:

- *Bottleneck Relief* (Section IV-C) to improve the throughput when a bottleneck is detected, and
- *Contraction* (Section IV-D) to reduce the communication distance (i.e. number of hops) between them to reduce the bandwidth requirements.

Pipelets *interact* (Section IV-E) as they do not dispose of information about the (global) system state. In the following, we detail how Pipelets achieve self-organization.

A. Overview and Problem Definition

Pipelets *exploit* properties of software pipelines to achieve self-organization. To achieve self-organization, a Pipelet must be able to (a) find out if it limits the throughput of the application it belongs to, and to (b) remap in a way that improves the throughput by balancing the load. Therefore, a Pipelet must be able to estimate the impact of task remapping so it can choose a remapping that improves the throughput of its application without impairing the throughput of other applications. Both (a) and (b) can be accomplished when Pipelets exploit the following properties of software pipelines:

(1) Each stage repeats *iterations* that wait for and receive input data, compute, and then pass the output to their successor as soon as this is ready to receive it.
(2) The slowest stage of a pipeline limits its throughput (i.e. it causes a bottleneck).
(3) Preceding and succeeding stages of a software pipeline communicate once per iteration, passing the output data from the *predecessor* as the input data of its direct *successor*. There is no further communication.
(4) The peak memory requirement of pipeline stages is often during computation because many buffers are freed after passing the output to the successor.

Pipelets exploit these properties in the following way: The strict temporal execution pattern of (1) enables Pipelets to measure the different time phases (by repeatedly querying the system time) that comprise their iteration: Figure 3 illustrates how we denote T_{WR} and T_{Recv} as the time required for waiting for and receiving the input data, while T_C denotes the time consumed by computation. Likewise, T_{WS} and T_{Send} denote the times for waiting for the successor and sending

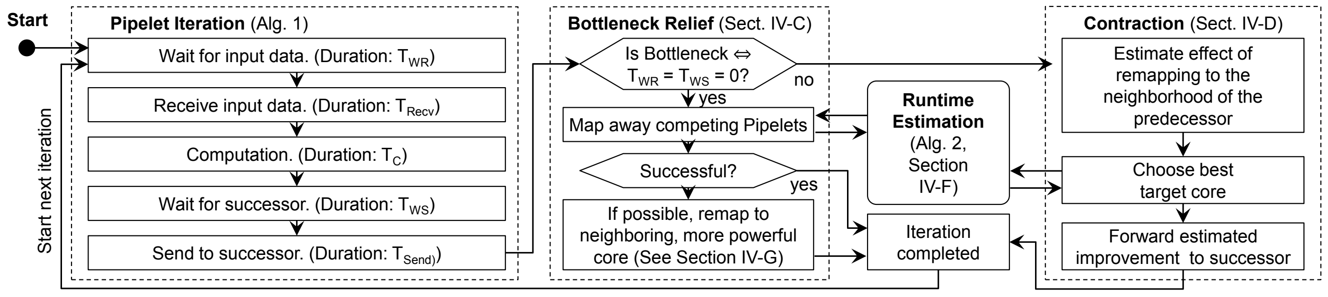
Start
Pipelet Iteration (Alg. 1)
Wait for input data. (Duration: T_WR)
Receive input data. (Duration: T_Recv)
Computation. (Duration: T_C)
Wait for successor. (Duration: T_WS)
Send to successor. (Duration: T_Send)
Start next iteration
Bottleneck Relief (Sect. IV-C)
Is Bottleneck ⇔ T_WR = T_WS = 0?
no
yes
Map away competing Pipelets
Successful?
yes
If possible, remap to neighboring, more powerful core (See Section IV-G)
Runtime Estimation (Alg. 2, Section IV-F)
Iteration completed
Contraction (Sect. IV-D)
Estimate effect of remapping to the neighborhood of the predecessor
Choose best target core
Forward estimated improvement to successor

Fig. 3. Overview of Pipelet self-organization

data to it. Algorithm 1 illustrates the main loop (including measurements of its time phases) of a Pipelet.

As the slowest stage of an application limits its throughput (2), other stages need to wait. Consequently, for an application A where $\mathbb{P}_A$ denotes the set of its Pipelets, we define λ_A as the time spent for each iteration:

$$\forall p, q \in \mathbb{P}_A : T^p_{WR} + T^p_{Recv} + T^p_C + T^p_{WS} + T^p_{Send} = T^q_{WR} + T^q_{Recv} + T^q_C + T^q_{WS} + T^q_{Send} = \lambda_A \qquad (1)$$

The one-to-one communication restriction of (3) reduces the potential impact of the remapping a Pipelet can have on the throughput of others. Only its predecessor, successor, and Pipelets that are mapped to the target core may be impacted.

Additionally, we define the *slack* of a Pipelet p as the time it needs to wait for its predecessor and successor:

$$slack_p = T^p_{WR} + T^p_{WS} \qquad (2)$$

The Pipelet that causes a bottleneck has a *slack* of 0. The throughput F_A of an application A is defined as:

$$F_A = \frac{1}{\lambda_A} \qquad (3)$$

To increase F_A, the Pipelet b_A causing the bottleneck of A tries to reduce T^b_{Recv}, T^b_C, or T^b_{Send} to decrease λ_A. The rest of this paper discusses in which way this can be achieved.
(4) implies that the task context (program code, stack, registers and heap) of a Pipelet is smallest directly after finishing an iteration (as it may deallocate temporary buffers required for computation). To obtain low remapping latencies, Pipelets remap after a completed iteration.

Figure 3 shows how Pipelets perform their main loop, i.e. an iteration and the steps required for self-organization: After the output data has been sent to the successor, a Pipelet evaluates its *slack* to see if it causes a bottleneck. If so, it tries to resolve it as detailed in Section IV-C.

B. Components

We employ two distinct components: Pipelets and *core guards*. On each core, a core guard task is responsible to (a) virtualize communication between Pipelets, (b) provide CPU type and load, bandwidth usage, and a list of mapped Pipelets, and for (c) helping Pipelets remap to its core. *Virtualized communication* between Pipelets enables Pipelet-to-Pipelet communication in an MPI-like manner even when multiple Pipelets are mapped to the same (receiving) core and without knowing a recipient's physical location.

Figure 4 shows how Pipelets communicate via core guards, implemented within the user-level and kernel-level on Intel's Single-Chip Cloud Computer (SCC) that runs a single-core

Algorithm 1 Pipelet main loop with time measurements

```
1: while application running do
2:    // the first Pipelet as no predecessor
3:    if Has Predecessor then
4:       $t_0$ = GetTime()
5:       WaitForData( $Predecessor$ )
6:       $t_1$ = GetTime()
7:       $T_{WR} = t_1 - t_0$
8:       $inputData$ = ReceiveData( $Predecessor$ )
9:       $t_2$ = GetTime()
10:      $T_{Recv} = t_2 - t_1$
11:   end if
12:   $t_2$ = GetTime()
13:   // Compute performs the computation of the Pipelet
14:   $outputData$ = Compute( $inputData$ )
15:   $t_3$ = GetTime()
16:   $T_C = t_3 - t_2$
17:   // the last Pipelet has no successor
18:   if Has Successor then
19:      WaitForRecipient( $Successor$ )
20:      $t_4$ = GetTime()
21:      $T_{WS} = t_4 - t_3$
22:      SendData( $Successor$, $outputData$ )
23:      $t_5$ = GetTime()
24:      $T_{Send} = t_5 - t_4$
25:   end if
26:   Interact with other Pipelets (Section IV-E)
27: end while
```

Linux (Ubuntu 10.4 LTS) on each individual core. Core guards (and Pipelets) can be implemented on most architectures, operating systems, and are not tied to the SCC.

C. Bottleneck Relief

If a Pipelet p of application A causes a bottleneck, it may increase the throughput F_A by decreasing T_{Recv}, T_C or T_{Send}, thus decreasing λ_A for all Pipelets $p \in \mathbb{P}_A$. Therefore, it asks other Pipelets that are mapped to its core to remap in order to free resources. These Pipelets are in turn responsible to find possible target cores. If remapping other Pipelets would decrease (any of) their application's throughputs beyond the gain for A, this option is discarded and b itself tries to remap to a different core in its *neighborhood*, which is a set of cores

User-Level: Pipelets communicate via *core guards* and are agnostic to the actual physical location of their successor.
Kernel-Level: *Core guards* maintain mapping information and establish connections if required or deliver data locally for Pipelets mapped to the same core.

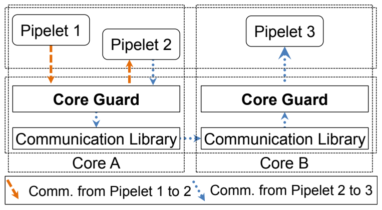

Fig. 4. Pipelets communicate via so-called core guards

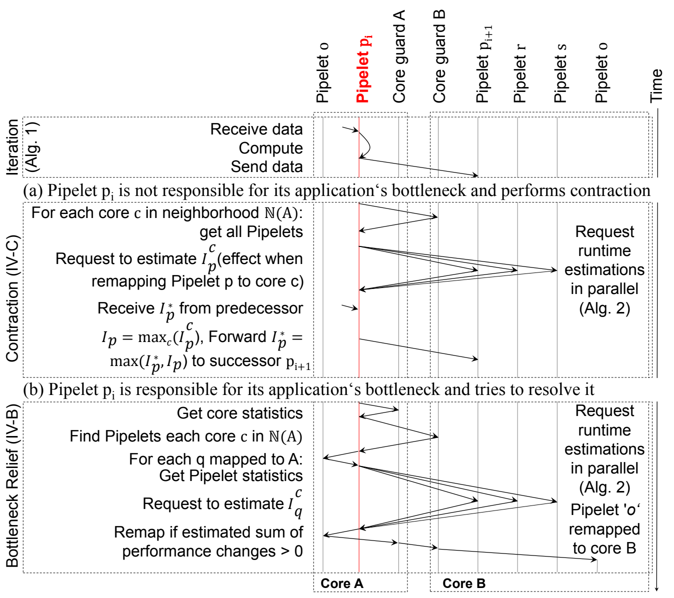
Pipelet o
Pipelet p_i
Core guard A
Core guard B
Pipelet p_{i+1}
Pipelet r
Pipelet s
Pipelet o
Time
Iteration (Alg. 1)
Receive data
Compute
Send data
(a) Pipelet p_i is not responsible for its application's bottleneck and performs contraction
Contraction (IV-C)
For each core c in neighborhood $\mathbb{N}(A)$:
get all Pipelets
Request to estimate I_p^c(effect when
remapping Pipelet p to core c)
Receive I_p^* from predecessor
$I_p = \max_c(I_p^c)$, Forward $I_p^* =$
$\max(I_p^*, I_p)$ to successor p_{i+1}
Request runtime estimations in parallel (Alg. 2)
(b) Pipelet p_i is responsible for its application's bottleneck and tries to resolve it
Bottleneck Relief (IV-B)
Get core statistics
Find Pipelets each core c in $\mathbb{N}(A)$
For each q mapped to A:
Get Pipelet statistics
Request to estimate I_q^c
Remap if estimated sum of
performance changes > 0
Request runtime estimations in parallel (Alg. 2)
Pipelet 'o' remapped to core B
Core A
Core B

Fig. 5. Interaction between Pipelets in several scenarios

physically closest to it (the size of the neighborhood is a design parameter). Small neighborhood sizes might increase the number of remappings (as it limits the search space), while large neighborhoods increase the overhead. In our experiments, we find a neighborhood size of 12 cores provides best results. If a Pipelet is not responsible for a bottleneck, it tries to *contract* as explained below in Section IV-D.

D. Contraction

Pipelets that do not cause a bottleneck *contract*, i.e. they remap to the spatial proximity (neighborhood) of their predecessor to reduce communication volumes if this does not impair throughput. This is is desirable because it may potentially increase the performance of other Pipelets that require higher bandwidths. Contraction is performed as follows: every iteration, each Pipelet p_i evaluates $\mathbb{D}_{p_i} = D_{p_i,p_{i-1}} + D_{p_i,p_{i+1}}$, which denotes the sum of the distance (e.g. the hop count, i.e. the number of hops between two cores.) to its predecessor and to its successor. It receives I_{p*} that denotes the improvement (reduction) in $\mathbb{D}$ for one of its predecessors $p*$ (the first stage does not receive a value), and estimates the impact on its throughput for every possible remapping to a core in its neighborhood that would decrease $\mathbb{D}_p$ without impacting any application's throughput. If it finds that its decrease in $\mathbb{D}_p$ exceeds I_p*, it updates this value and sets $p*$ to p. Next, it forwards I_{p*} to its successor. The Pipelet without a successor informs the Pipelet $p*$ with the largest positive improvement $I_{p*} > 0$ to perform the corresponding remapping. No contraction is carried out in this iteration if a Pipelet remaps to relieve a bottleneck.

E. Interactions

Pipelets interact to achieve self-organization. Figure 5 shows a sequence diagram of the interactions for a Pipelet p_i (p_{i+1} is its successor, while o, r, and s are other Pipelets that may but need not belong to the same application).

Pipelets interact in three cases: 1) p_i interacts with p_{i+1} to send the output data. Secondly, if p_i is not causing a bottleneck, it interacts with other Pipelets in its neighborhood (in this example o, r and s) to perform contraction. Therefore, it requests runtime estimates for possible contraction remappings. Thirdly, if p_i causes a bottleneck, it interacts with the other Pipelets in its neighborhood to estimate which of the possible remappings to relieve the bottleneck (as described in Section IV-C) offers the best improvement. Pipelets interact with *core guards*, a helper task detailed below, to receive core information (such as CPU load and bandwidth availability and a list of mapped Pipelets).

To summarize: interactions take place between Pipelets in spatial (i.e. neighborhood) and temporal (i.e. predecessor/successor relationship) proximity. This limitation induces local, sub-optimal decisions, which trades scalability for a loss of optimality. While Pipelets cannot provide *globally optimal* mapping at runtime, the experimental results show that Pipelets achieve a near-optimal remapping.

Algorithm 2 Runtime estimation

Input:

$Dir : boolean$	Direction: to or away from core
$\mathbb{R} = \{CPU, BW_{in}, BW_{out}\}$	Resource change tuple

Definitions:

H_t^p	Heterogeneity factor for Pipelet p on core type t
t_{new}	Type of new (evaluated) core
t_{old}	Type of old (current) core

Output:

I_p	Performance improvement for p (can be negative)

```
1: if dir == away from core then
2:     $\mathbb{R} = -1 * \mathbb{R}$   // Invert requirements (free resources)
3: end if
4: $\widetilde{T_C} = T_C * (1 + \mathbb{R}_{CPU}) * H_{t_{new}}^p / H_{t_{old}}^p$
5: $\widetilde{T_{Recv}} = T_{Recv} * (1 + \mathbb{R}_{BW_{in}})$
6: $\widetilde{T_{Send}} = T_{Send} * (1 + \widetilde{\mathbb{R}_{BW_{out}}})$
7: return $I_p = T_{Recv} - \widetilde{T_{Recv}} + T_C - \widetilde{T_C} + T_{Send} - \widetilde{T_{Send}}$
```

F. Runtime Estimation

Algorithm 2 shows how each Pipelet estimates the impact of a potential remapping on T_C, T_{Recv}, and T_{Send}: The input parameters are the measured resource demands of the Pipelet and the direction of the mapping (i.e. either mapping *to* or *away from* the a core, i.e. requiring or giving up resources). Timing estimations are defined as the product of the measured timings (T_C, T_{Recv} and T_{Send}) and the relative changes in resource availability (as required or given up by the Pipelet that is potentially remapped). When T_{Recv} or T_C increase, the *slack* decreases until it reaches 0.

G. Handling Heterogeneity of Cores

Heterogeneity (e.g. different extensions, performance, voltage/frequency levels, etc.) can increase performance and efficiency over homogeneous architectures [16]. Pipelets can take advantage of this by introducing a heterogeneity factor H_p^t that denotes how T_C of a Pipelet p depends on the core type t (based on offline profiling). A value of ∞ prohibits a remapping of the Pipelet to a core type. For homogeneous systems, H_p^t is equal for all p, t.

As a summary, the new concept of Pipelets exploits properties of software pipelines to derive remapping decisions that *resolve performance bottlenecks* or reduce communication distances by *contraction*. They interact based on spatial or temporal proximity to limit their overhead.

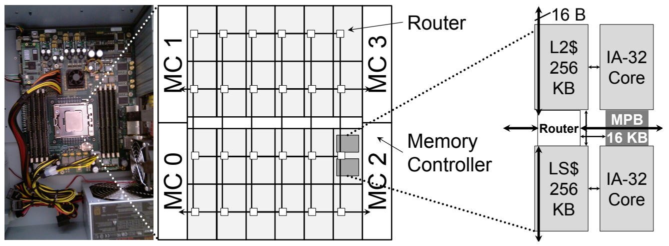

Fig. 6. Intel's Single-Chip Cloud Computer (SCC)

V. Experimental Results

In this section, we examine the following experiments to show the effectiveness and scalability of Pipelets:

- We compare the throughput of Pipelets to a state-of-the-art load balancing scheme, AIAC [4].
- Similarly, we compare their bandwidth requirements.
- We measure the overhead of Pipelets on 24, 48 cores.

Target Architecture: We target many-core systems where each core has private memory, is multi-threaded, and is connected via a high-bandwidth, low-latency Network-on-Chip. For our experiments, we evaluate Pipelets on Intel's Single-Chip Cloud Computer (SCC) that integrates 48 x86 (P54C) cores (45nm technology [13]) at 800 MHz each, shown in Figure 6. 24 tiles contain 2 cores, 2 private L2 caches (256 KB each), a network interface (router) for the 2GHz NoC interconnect (bisection bandwidth: 256 GB/s) and a 16 KB message-passing buffer (MPB) each. Four memory controllers access off-chip DRAM. The SCC is well-suited for Pipelets because it provides many (multithreaded) cores and a fast Network-on-Chip for Pipelet communication.

Application scenario: We analyze a system that simultaneously runs two instances (each) of three complex real-world software-pipelined applications written in C++ in order to generate a highly complex scenario: Machine-vision-based *'Assisted Drive'* (17 stages) visually detects objects (e.g. cars, humans, and traffic signs) from video sequences (similar to [18]) and displays an augmented reality image. The deployed algorithms stem from the Integrated Vision Toolkit (IVT [1]) and include, among others, FFT, Harris Corner Detection and Scale-Invariant Feature Transform (SIFT). *'Speech Recognition'* (20 stages) is based on the CMU Sphinx3 [2] toolset, while *'Video Telephony'* (10 stages) includes H.264, MP3 encoding, and 3DES encryption. The applications have been pipelined manually and use OpenMPI 1.6.1 [3] for communication. Figure 7 shows how the computational demands of the stages of *'Assisted Drive'* depend on the input sequences of high, medium and low traffic.

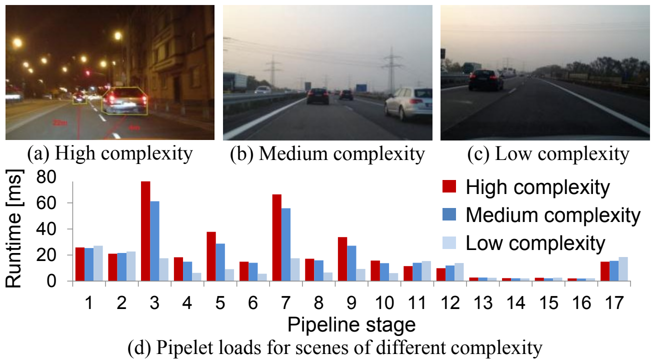

Fig. 7. Inputs and resource demands of *'Assisted Drive'*

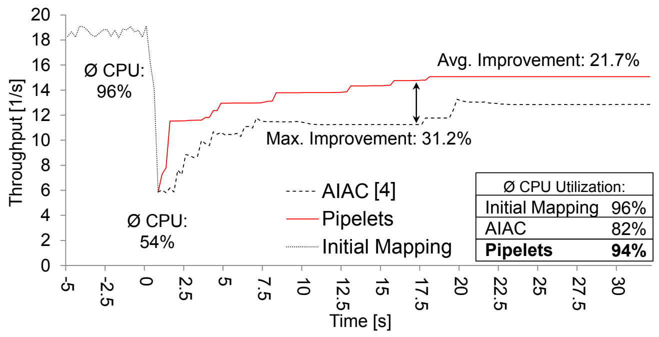

Fig. 8. System throughput of Pipelets compared to AIAC [4]

Likewise, the computational demands of *'Video Telephony'* alter depending on slow or fast motion in its input video or changing speakers (requiring audio de- or encoding, or both). *'Speech Recognition'* is computationally expensive when the user directs commands to it and idles otherwise.

Results: For high system load, we run 2 (independent) instances of each application (total: 94 Pipelets) on the SCC. We randomly change the input data of *'Assisted Drive'*, *'Video Telephony'* and *'Speech Recognition'*. Figure 8 compares the average system throughput across all applications achieved by Pipelets and AIAC [4]: the throughput achieved by Pipelets exceeds AIAC by a minimum of 10.3%, a maximum improvement of 31.2%, and an average improvement of 21.7%. The CPU utilization drops from an initial 96% to 54% when the input data changes. By re-balancing the load, Pipelets achieve an average CPU utilization of 94%, while AIAC achieves 82%.

Figure 9 depicts the throughput of each application before and after the input data changes. The throughput of *'Assisted Drive'* drops by 45.6% when switching from the low-complexity to the medium-complexity video scenes. Then, Pipelets carry out 4 remappings until they cannot improve the throughput further, which yields an improvement of 1.27x until the Pipelet responsible for a bottleneck neither shares CPU resources nor communication links to its predecessor and successor with other Pipelets.

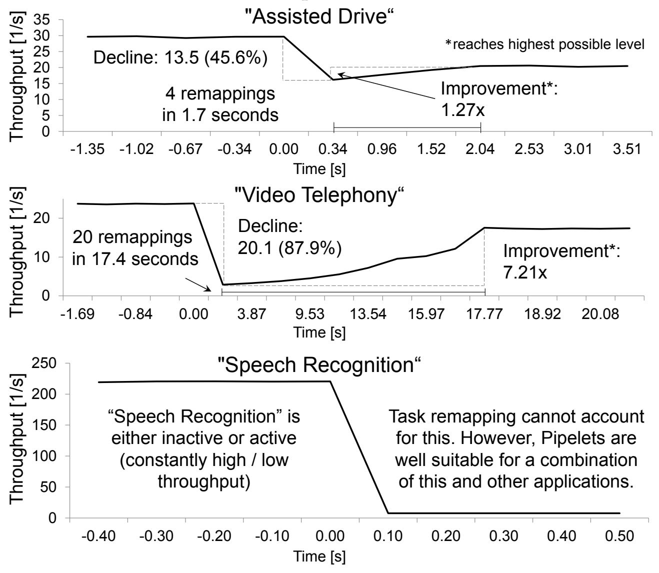

Fig. 9. Application throughput over time achieved by Pipelets

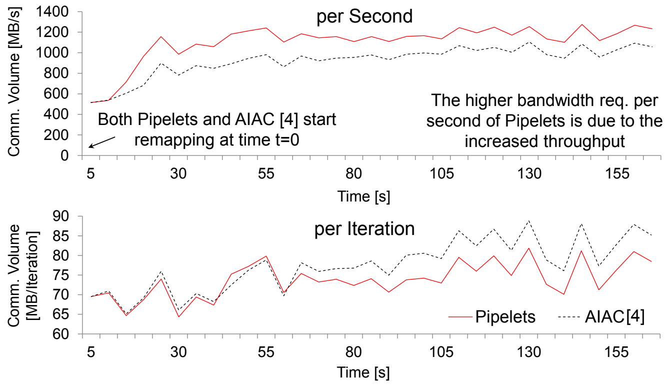

Fig. 10. Comparing bandwidth requirements to state of the art [4]

When changing from a video scene with low to fast motion, the throughput of *'Video Telephony'* drops by 87.9% from approx. $24\frac{1}{s}$ to $2.89\frac{1}{s}$. After 20 remappings in 17.4 seconds (due to the low throughput of $2.89\frac{1}{s}$), Pipelets restore a throughput of around $17.5\frac{1}{s}$, which corresponds to a speedup of 7.21x. By activating *'Speech Recognition'*, its throughput drops from $220\frac{1}{s}$ (idle) to $9\frac{1}{s}$ (active). We choose this example to investigate an application where Pipelets cannot adapt mappings to improve the throughput: Even though load balancing cannot improve it's throughput, this has no negative effects on the overall system and other applications. Figure 10 compares the bandwidth requirement of the applications when using Pipelets and AIAC [4]. After remapping, Pipelets require more bandwidth as they achieve a higher throughputs, thus the Pipelets communicate more often. Per iteration, however, Pipelets significantly reduce the bandwidth requirements. Therefore, Pipelets base remappings both on communication and on computational demands.
Figure 11 shows the communication overhead of Pipelets as measured for 24 and 48 cores. When Pipelets do not remap, the total (summed) communication overhead of 8.9 KB/s (24 cores) or 13.2 KB/s (48 cores) can be considered as negligible. The communication overhead does not increase significantly with a growing number of cores because Pipelets limit interactions on spacial and temporal proximity, and thus achieve runtime load balancing in a scalable manner. The communication overhead grows with a growing frequency of changes in the input data that cause Pipelets to remap, which is dominated by the overhead for transferring the (task) context of the Pipelets.

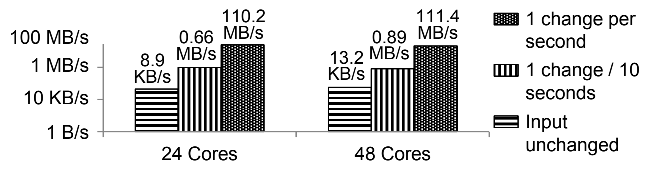

Fig. 11. Communication overhead (incl. transfer of task contexts)

VI. Conclusion

We have introduced the new concept of Pipelets that exploits properties of the paradigm of software pipelines and enhances it to increase the performance of many-core systems by up to 31.2% compared to state-of-the-art when their resource demands alter unpredictably. The latter is the case for many complex applications.

We have shown that the concept of Pipelets offers a promising potential for an application to adapt during runtime and therefore to resolve computational and communication bottlenecks that would otherwise result in lower throughput when state-of-the-art is used [4]. The main contributing property of Pipelets is their ability to self-organize i.e. to adapt mappings during execution when inefficiencies are detected and performance improvement appears to be achievable. The use of Pipelets and their exploitation is not limited to an application's property. Instead, Pipelets can be deployed in all cases where regular software pipelines are deployed.

VII. 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).

References

[1] http://ivt.sourceforge.net.
[2] http://cmusphinx.sourceforge.net.
[3] http://www.open-mpi.org/software/ompi/v1.6/.
[4] J. M. Bahi et al. Dynamic load balancing and efficient load estimators for asynchronous iterative algorithms. *IEEE Trans. Parallel Distrib. Syst.*, 16:289–299, April 2005.
[5] S. Bertozzi et al. Supporting task migration in multi-processor systems-on-chip: A feasibility study. *Proc. ACM/IEEE Design, Automation and Test in Europe (DATE)*, 2006.
[6] 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.
[7] J. Cheng et al. MAPS: An Integrated Framework for MPSoC Application Parallelization. In *Proc. ACM/IEEE Design Automation Conf. (DAC)*, 2008.
[8] C. Chou et al. Incremental run-time application mapping for homogeneous nocs with multiple voltage levels. In *Proc. of Int. Conf. on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2007.
[9] C.-L. Chou et al. Contention-Aware Application Mapping for Network-on-Chip Communication Architectures. In *IEEE Int. Conf. on Computer Design (ICCD)*, 2008.
[10] M. Al Faruque et al. ADAM: Run-Time Agent-Based Distributed Application Mapping for On-Chip Communication. In *Proc. ACM/IEEE Design Automation Conf. (DAC)*, 2008.
[11] Y. Ge et al. Distributed task migration for thermal management in many-core systems. In *Proc. ACM/IEEE Design Automation Conf. (DAC)*, 2010.
[12] P. Horn. Autonomic Computing: IBM's Perspective on the State of Information Technology, IBM Corporation. http://www.research.ibm.com/autonomic/manifesto/ autonomic_computing.pdf, 2001.
[13] J. Howard et al. A 48-Core IA-32 Message-Passing Processor with DVFS in 45nm CMOS. In *IEEE Int. Solid-Stat Circuits Conference (ISSCC)*, 2010.
[14] M. C. Huebscher and J. A. McCann. A survey of autonomic computing degrees, models, and applications. *ACM Comput. Surv.*, 40(3), August 2008.
[15] J. Jahn et al. CARAT: Context-aware Runtime Adaptive Task Migration for Multi Core Architectures. In *Proc. ACM/IEEE Design, Automation and Test in Europe (DATE)*, 2011.
[16] R. Kumar et al. Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. In *Proc. of the Int. Symposium on Computer Architecture*, 2004.
[17] 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.
[18] M. Müller et al. Design of an automotive traffic sign recognition system targeting a multi-core soc implementation. In *Proc. ACM/IEEE Design, Automation and Test in Europe (DATE)*, 2010.
[19] 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.
[20] G. Ottoni et al. Automatic Thread Extraction with Decoupled Software Pipelining. In *Micro, IEEE*, 2005.
[21] M. Rajagopalan et al. Thread scheduling for Multi-Core Platforms. In *USENIX HotOS*, 2007.
[22] J. Stender et al. Mobility-based runtime load balancing in multi-agent systems. In *Proc. of the Int. Conf. on Software Engineering and Knowledge Engineering (SEKE)*, 2006.
[23] W. Thies et al. A practical approach to exploiting coarse-grained pipeline parallelism in C programs. In *Micro, IEEE*, 2007.