Runtime Resource Allocation for Software Pipelines

JANMARTIN JAHN, SANTIAGO PAGANI, SEBASTIAN KOBBE, JIAN-JIA CHEN, and JÖRG HENKEL, Karlsruhe Institute of Technology (KIT), Germany

Efficiently allocating the computational resources of many-core systems is one of the most prominent challenges, especially when resource requirements may vary unpredictably at runtime. This is even more challenging when facing unreliable cores—a scenario that becomes common as the number of cores increases and integration sizes shrink.

To address this challenge, this article presents an optimal method for the allocation of the resources to software-pipelined applications. Here we show how runtime observations of the resource requirements of tasks can be used to adapt resource allocations. Furthermore, we show how the optimum can be traded for a high degree of scalability by clustering applications in a distributed, hierarchical manner. To diminish the negative effects of unreliable cores, this article shows how self-organization can effectively restore the integrity of such a hierarchy when it is corrupted by a failing core. Experiments on Intel's 48-core Single-Chip Cloud Computer and in a many-core simulator show that a significant improvement in system throughput can be achieved over the current state of the art.

Categories and Subject Descriptors: B.8.0 [**Performance**]; B.8.1 [**Reliability/Fault Tolerance**]; C.1.4. [**Parallel Architectures**]; C.4 [**Performance of Systems**]; D.1.3. [**Parallel Programming**]

General Terms: Performance

Additional Key Words and Phrases: Many-core systems, resource allocation, task mapping, runtime system management, distributed systems, software pipelines

ACM Reference Format:
Janmartin Jahn, Santiago Pagani, Sebastian Kobbe, Jian-Jia Chen, and Jörg Henkel. 2015. Runtime resource allocation for software pipelines. ACM Trans. Parallel Comput. 2, 1, Article 5 (May 2015), 23 pages.
DOI: http://dx.doi.org/10.1145/2742347

1. INTRODUCTION

Many-core systems offer the potential for parallel processing. The cores and the communication links between them (i.e., their *resources*) must be used efficiently to achieve a high system throughput. This is largely determined by the allocation of cores to the applications [Singh et al. 2013]. In scenarios where the resource requirements of tasks may change at any time or where tasks may be started or stopped at any time (we call such scenarios *dynamic scenarios*), the problem of allocating resources is extended from *finding* allocations to *adapting* them at runtime to reflect those changes. Otherwise, a significant degradation of the system throughput is observed in many instances [Jahn and Henkel 2013; Kobbe et al. 2011; Schor et al. 2012] (Section 3 discusses an

Authors' addresses: J. Jahn, S. Pagani, and J. henkel, Karlsruhe Institute of Technology, Chair for Embedded Systems, Haid-und-Neu-Str. 7, 76131 Karlsruhe, Germany; emails: (jahn, santiago.pagani, henkel)@kit.edu; J.-J. Chen, University of Dortmund, Fakultät für Informatik, Informatik 12, Otto-Hahn-Str. 16, 44227 Dortmund, Germany; email: jian-jia.chen@cs.uni-dortmund.de.

DOI: http://dx.doi.org/10.1145/2742347

example). Dynamic scenarios are increasingly common, and therefore efficient and effective resource (re-)allocation is of paramount importance [Schor et al. 2012].

State-of-the-art resource allocation techniques do not sufficiently address dynamic scenarios as they target a limited set of predefined scenarios (e.g., Schor et al. [2012]), because they balance the computational load while neglecting intertask communication (e.g., Bahi et al. [2005]), or because their observations and resource allocations are limited to local decisions (e.g., Jahn and Henkel [2013] and Kobbe et al. [2011]). A thorough discussion of the state of the art can be found in Section 4. Complementing the state of the art, we address the problem of efficiently allocating resources in dynamic scenarios for *malleable software pipelines* by adapting allocations based on runtime observations. Malleable applications provide the flexibility of deploying cores dynamically (e.g., Turek et al. [1992] and Feitelson et al. [1997]) and thus allow reallocation of resources at runtime. This article focuses on software pipelines, as they are a well-established means for parallelizing a large class of complex applications. Many stream-processing applications, including common image/video and networking applications, are well suited for software pipelining.

We start with a centralized method (as presented in Jahn et al. [2013]) that allocates resources for the entire system based on global system knowledge. Based on this, we detail how this centralized resource allocation is extended in a distributed, hierarchical way to reduce its high computational and communication overheads. Our proposed distributed allocation employs several *nodes* that communicate hierarchically to achieve near-optimal resource allocations and a high degree of *scalability*—that is, limited overheads even in scenarios with a large number of cores. This way, the computational overhead is distributed among cores. However, distributed allocation may be vulnerable to unreliable cores because the communication of nodes can be significantly impaired by malfunctioning cores. A decreased reliability of cores can follow the continued shrinking of integration sizes beyond 22nm. This shrinking causes high power densities [Narayanan and Xie 2006] and an increasing impact on process variations [Borkar 2005]. High power densities may lead to thermal problems that can cause temporary or permanent malfunction of cores [Brooks et al. 2007; Narayanan and Xie 2006]. Due to process variations, manufacturing many-core systems for a guaranteed reliability may lead to low yields. Both effects suggest that in the future, systems may need to be able to operate on unreliable cores [Borkar 2005] even though distributed resource allocation does not form a single point of failure (as opposed to centralized allocation): the malfunctioning of a core may remove a node of the distributed resource allocation, which can jeopardize its effectiveness and thus decrease the system throughput significantly. We show that this can be severe, and we employ *self-organizing* nodes to address this problem by increasing the *resilience* (i.e., the impact of crashing cores on the system throughput) of our distributed allocation.

Self-organization is a paradigm for managing complex systems in a distributed manner and is successfully employed in many instances. A comprehensive survey is presented by Huebscher and McCann [2008]. In the context of this article, self-organization means that communicating nodes are responsible for finding their communication counterparts and for observing the system state that is relevant to them. We use self-organization to avoid the decreased throughput that would otherwise result when the hierarchy of our distributed resource allocation is corrupted by crashing nodes.

We target systems with many cores; private, distributed memories; and network-on-chips.

The contributions of this work are as follows:

(1) We present a novel concept of self-organizing software pipelines that increase the resilience of distributed resource allocation against unreliable cores through a novel concept of self-organizing software pipelines.

(2) We detail the centralized, optimal method for the allocation of resources to malleable software pipelines for *dynamic scenarios* presented in Jahn et al. [2013]. Furthermore, we explain how the optimum can be traded for a high degree of scalability.

The rest of this article is organized as follows. Section 2 defines some terms, and Section 3 illustrates the need to adapt resource allocations at runtime. Section 4 discusses the related work, and Sections 5 and 6 detail our application models and define the problem of allocating resources. Section 7 describes our algorithmic solution for allocating resources. Section 8 outlines implementation details, and experiments on the performance and overheads are presented and discussed in Section 9. Section 10 concludes this article.

2. TERMS AND DEFINITIONS

For the rest of this article, we will use the following terms and definitions:

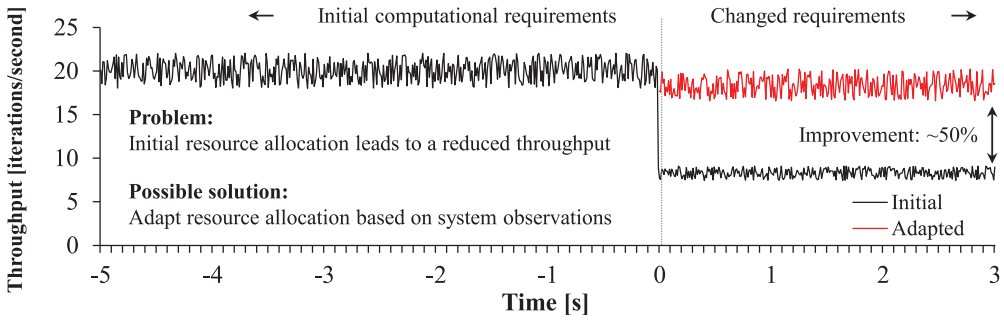
—*Resource allocation* refers to decisions of (1) assigning cores to parallel applications (consisting of multiple tasks) and of (2) choosing how to distribute the tasks of each application to the cores assigned to it.
—A *software pipeline* is a parallel application that consists of several communicating *stages* that repeatedly compute *iterations* on a stream of input data. Each stage is a *task* that is assigned to a core. The output of one stage forms the input data of its direct successor; there is no further communication between stages.
—A *malleable software pipeline* is a software pipeline that can reduce the number of cores that it uses by *fusing* consecutive stages so that they are allocated to the same core, similar to fusing filters in StreamIt [Thies et al. 2001]. Consequently, no intercore communication is necessary between fused stages. Fused stages can be split.
—*Task migration* refers to the mechanism for transferring the execution of a task from one core to another.
—The *throughput* of an application denotes the number of iterations that it finishes per second. Accordingly, the *system throughput* denotes the average throughput of all running applications. A formal definition can be found in Section 5.
—In *dynamic scenarios*, the set of concurrently running applications, the resource requirements of the individual tasks and the volume of intertask communication may change unpredictably at any time. This can arise, for example, through user interactions or because of changes in the input data.

3. MOTIVATION

Let us consider a simple example of a software-pipelined computer vision application (object tracking) with eight stages allocated to a system with four cores.[1]

Figure 1(a) shows that an established resource allocation (core 1: stages 1 through 3, core 2: stages 4 and 5, core 3: stages 6 and 7, core 4: stage 8) achieves a throughput of approximately 20.13 iterations/second. The resource requirements change at runtime when multiple tracked objects are added to the input scene. Figure 1(b) shows how the average runtime of each stage changes. Due to these changing resource requirements, the average throughput drops to 8.35 iterations/second. A possible solution to this problem is to adapt the resource allocation as shown in Figure 1(d) (core 1: stage 1, core 2: stage 2, core 3: stages 3 and 4, core 4: stages 5 through 8), which results in a throughput of 18.40 iterations/second. Consequently, adapting resource allocations at runtime based on observations of the (unpredictable) resource requirements can

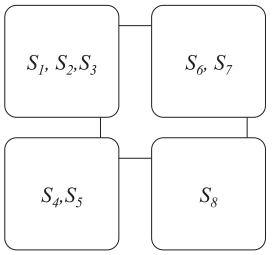
[1]For this example, four cores of Intel's Single-Chip Cloud Computer (SCC) [Howard et al. 2010] are used.



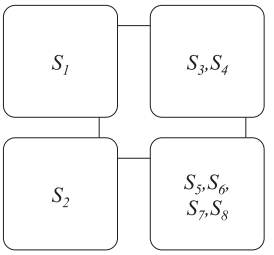
(a) Throughput achieved by initial and adapted resource allocation

Stage	Initial	Changed
1	24.3	35.3
2	11.7	54.3
3	13.2	31.3
4	22.9	20.8
5	26.2	11.7
6	23.8	13.2
7	26.1	23.1
8	49.3	21.2

(b) Avg. comp. requirements [ms]



(c) Initial resource allocation



(d) Adapted resource allocation

Fig. 1. Changing computational requirements, resulting throughputs, and initial/adapted resource allocations of stages S_{1-8} to four cores.

significantly improve the throughput. A more complex example of a system with up to 1,024 cores running up to 275 instances of real-world applications concurrently is discussed in Section 9.3.

However, adapting resource allocation at runtime is a challenging problem because balancing the resource requirements of tasks leads to a large problem space (e.g., there are approximately $1.6 \cdot 10^{160}$ possible ways to allocate 100 stages to 48 cores). Jointly balancing the computational and communication requirements between cores is an NP-complete problem in a strong sense, which can be reduced from the 3-PARTITION problem [Johnson and Garey 1979]. Hence, allocating resources for larger systems at runtime may require a prohibitively high overhead or may require heuristics that lead to suboptimal solutions.

4. RELATED WORK

This section highlights why the state of the art does not sufficiently address resource allocation in *dynamic scenarios*. To do this, we group the related work into resource allocation for software pipelines and similar concepts such as Kahn process networks (KPNs) and synchronous dataflow (SDF), and for parallel applications in general, assuming either distributed or shared memories.

Resource allocation specific to software-pipelined systems has recently been proposed. The authors of Schor et al. [2012] suggest calculating a set of optimal resource allocations for given scenarios at design time. This works well when the set of possible scenarios is known at design time, but it does not aim at capturing cases where the

resource requirements are unpredictable. This is, however, the case when they depend on user interactions or on the input data. Dynamic allocation of stream-processing applications, which are a superset of software pipelines, to embedded multicores with scratchpad memories is proposed by Lee et al. [2012]. This approach targets unpredictable resource availability, whereas the resource requirements of applications are assumed to remain constant.

For application scenarios that consist of a single application, multiple methods for resource allocation have recently been proposed. Stream-processing applications modeled as SDF can be allocated to multicore processors using design-time adjustments of the granularity of parallelism through loop unrolling [Che and Chatha 2012]. This approach employs compiler techniques that aim at maximizing the use of on-chip scratchpad memory and combines this with compiler-instrumented data buffering and background data prefetching. Similarly, SDF-based applications can be allocated to SPM-based multi/many-cores based on an evolutionary algorithm-based technique as proposed by Choi et al. [2012]. This method generates a static schedule at design time, which includes a schedule for prefetching data from off-chip DRAM to on-chip scratchpads using hardware DMA units.

For hard-real-time tasks, Bamakharma and Stefanov [2012] propose to decompose cyclostatic SDF graphs into asynchronous sets of periodic tasks with implicit deadlines. Under some conditions, this allows achievement of the maximum throughput. To achieve this, the authors propose allocating resources to task subsets and focus on finding good task set representations.

To jointly optimize resource allocation for computation and communication, Castrillon et al. [2012] propose a heuristic design-time resource allocation phase following a phase of application synthesis of KPNs. However, a large runtime results due to the large complexity that arises from the expressiveness of KPNs.

Resource allocation for parallel applications in general, assuming distributed memories. Bahi et al. [2005] present a heuristic runtime load balancing method for asynchronous, iterative algorithms (AIAC) in grid computing systems. It aims at balancing the computational load among cores by exchanging workload when imbalances are detected. To achieve this, this method repeatedly observes the workloads of all cores and performs distributed interactions among them. Workload is exchanged by transferring an application's individual work items, such as video frames, between neighboring cores. Due to its focus, it does not take intertask communication into account. It therefore may achieve inferior performance when tasks communicate heavily, as is the case for many complex, real-world applications.

In Kobbe et al. [2011], DistRM, a distributed heuristic for resource allocation, is presented, which uses interacting software agents. Based on runtime observations and offline profiles, agents possess local information and interact to allocate or re-allocate resources when applications are started or stopped, or when their computational requirements vary significantly. However, their approach for achieving a scalable solution for up to 4,096 cores limits their decisions to local regions, which can potentially result in a low throughput of the system.

Considering that AIAC [Bahi et al. 2005] and DistRM [Kobbe et al. 2011] are the most similar to the methods for resource allocation presented in this article, Section 9 compares them to our proposed methods.

A statistical approach based on extreme value theory is presented in Radojković et al. [2012]. It generates a large random set of resource allocations, and the calculations have a runtime of 25 minutes to 2 hours, which we consider infeasible for dynamic scenarios that require updating of resource allocations at runtime. In contrast to this, our methods for resource allocation allow calculation of optimal resource allocations in polynomial time and near-optimal resource allocations in nearly constant time.

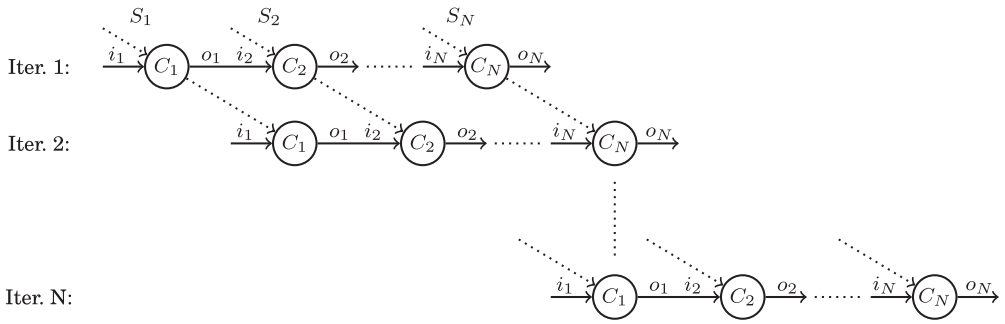


Fig. 2. Our model for malleable software pipelines.

Resource allocation for parallel applications in general, assuming shared memories. For shared-memory systems, numerous works [Stender et al. 2006; Luk et al. 2009; Nollet et al. 2005; Carvalho et al. 2007; Klues et al. 2010; Lakshmanan et al. 2009; Li et al. 2007; Rajagopalan et al. 2007] propose adaptive resource allocation for balancing the computational load at runtime. Snavely and Tullsen [2000] propose deriving co-schedules based on offline profiles, with an extension to support different priority levels [Snavely et al. 2002]. However, the focus of these methods is on architectures with few cores, and they require a shared address space.

To summarize, state-of-the-art resource allocations either achieve inferior performance due to their scope, are not applicable to systems with distributed memories, or do not target dynamic scenarios. However, it is important to address these scenarios for systems with many cores and distributed memories.

5. PIPELINE MODEL

This section discusses the model for malleable software pipelines. Each application k forms a pipeline P_k with N_k stages. Every stage S_j is characterized by c_j, i_j, and o_j that denote the time consumed (in each iteration) for computation, receiving the input data from its direct predecessor, and transferring the output data to its direct successor, respectively (Figure 2).

To decide upon the allocation of resources to applications, it is necessary to model the throughput for a given allocation. Therefore, we require that each core is allocated to at most one application (i.e., cores may not be shared among applications). Their maximum throughput is limited by their slowest stage. The *maximal response time* R_k for pipeline P_k can be defined consequently:

$$R_k = \max_{1 \le j \le N_k} \{i_j + c_j + o_j\}. \tag{1}$$

Hence, the maximum *throughput* of pipeline P_k is defined as $\frac{1s}{R_k}$.

We introduce the *malleability* property to software pipelines by defining the basic operation *fusion*, in which multiple consecutive pipeline stages are combined.

A fusion of stages creates a new stage that combines the computational requirements of the original stages but does not require communication between them, as shown in Figure 3. Communication between fused stages is not necessary because they are executed sequentially. Thus, no concurrent memory access can occur; consequently, fused stages can safely use shared memory. Furthermore, as fused stages form a single task, no task scheduling is necessary. This way, fusing stages may reduce the maximal response time R_k of a pipeline. Additionally, fusing stages changes the degree of parallelism of the application, which then runs on a smaller number of cores.

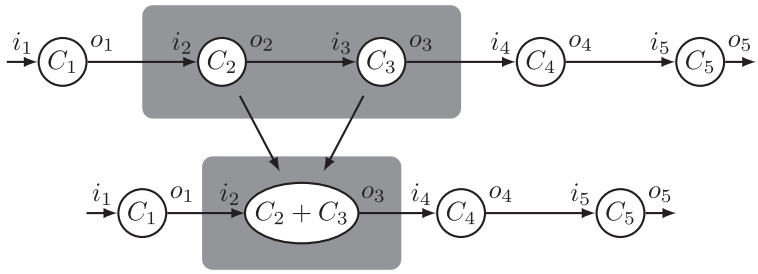


Fig. 3. Fusion of pipeline stages.

6. PROBLEM DEFINITION

This section defines the problem of allocating resources to malleable software pipelines. To simplify this problem, it is divided into two parts:

(1) *Global problem*: Distributing the cores of a system among the applications (Section 6.1) so that the overall system throughput is maximized.
(2) *Subproblem*: Assigning the stages of an application to a given number of cores (Section 6.2), thus providing the fusions of the pipeline stages.

6.1. Global Problem: Optimizing System Throughput

Given a set of K applications $P = \{P_1, P_2, \ldots, P_K\}$ with weights $W = \{w_1, w_2, \ldots, w_K\} \in \mathbb{R}^K$ (weights express priority levels), each application P_k uses up to M_k cores and has a maximal response time R_k. The objective is to *maximize the overall weighted system throughput* by finding an optimal allocation of (up to) M cores to the individual applications.

$$\text{Maximize} \left\{ \sum_{k=1}^{K} \frac{w_k}{R_k} \right\} \quad \text{such that} \sum_{k=1}^{K} M_k \leq M \tag{2}$$

This definition of the global problem maximizes the overall weighted throughput. This implies that applications with low weights may suffer from very low throughputs in favor of the throughput of applications with high weights. In scenarios where this is unacceptable, a minimum throughput can be guaranteed for each application. To do so, the global problem may be formulated alternatively:

$$\text{Maximize} \left\{ \min_{1 \leq k \leq K} \left\{ \frac{w_k}{R_k} \right\} \right\} \quad \text{such that} \sum_{k=1}^{K} M_k \leq M. \tag{3}$$

This ensures a minimum target performance for each application. To solve the global problem, we present centralized, optimal resource allocation in Section 7.2 and highly scalable, distributed resource allocation in Section 7.3. However, solving the global problem requires solving the Subproblem of fusing pipeline stages first.

6.2. Subproblem: Fusion of Pipeline Stages

The throughput of an application is affected by *how* the stages are fused. Thus, the subproblem for fusing the stages of each application $k \in K$ minimizes the maximal response time R_k.

For this, for each application k, a set of F_k fusions is defined as

$$\{F_1(1, j_1), F_2(j_1 + 1, j_2), \ldots, F_{F_k}(j_{F_k - 1} + 1, N_k)\} \quad \text{such that } F_k \leq M_k$$

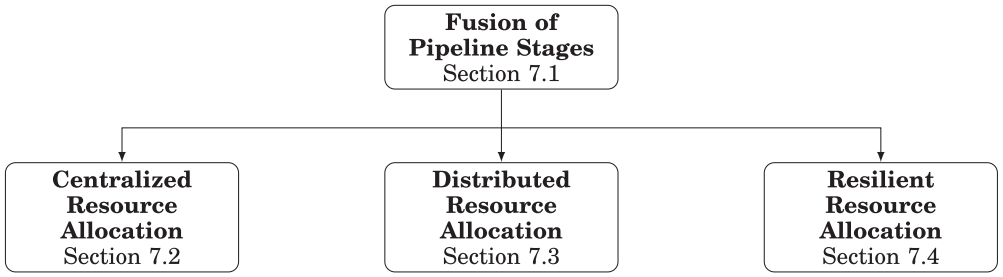


Fig. 4. Overview of our solution. Our centralized and distributed resource allocation, as well as the extension for an increased resilience against malfunctioning cores, use the solution of fusing pipeline stages. Hence, for each application, the optimal fusion of its pipeline stages is calculated for any number of cores up to the number of its stages. Based on this solution, it is decided how many cores to allocate to each application so that the system throughput is maximized.

so that each application k uses not more than M_k cores. Furthermore, according to Section 5, we define

$$F_i(l, j) = i_l + o_j + \sum_{h=l}^{j} c_h$$
$$1 \leq l \leq N_k,$$
$$l \leq j \leq N_k,$$

and thus the subproblem is defined as

$$Minimize \left\{ \max_{1 \leq f \leq F_k} \{F_f\} \right\} \quad | \forall k \in K. \tag{4}$$

We present an algorithm to solve this problem in Section 7.1.

7. ALGORITHMIC SOLUTION FOR ALLOCATING RESOURCES

In the following, our solution for the problem of resource allocation is detailed. Figure 4 shows an overview. Our centralized and distributed resource allocation (Sections 7.2 and 7.3) use the solution of the subproblem of fusing pipeline stages (Section 7.1). Section 7.4 shows how distributed resource allocation can be resilient against unreliable cores.

For simplicity, the equations are explained for the centralized resource allocation, whereas the proposed distributed resource allocation extends this concept in a hierarchical way.

7.1. Fusion of Pipeline Stages

To find an optimal solution to this subproblem (i.e., a solution that results in the maximum throughput compared to all other possible fusions), all possible combinations of fusions need to be taken into consideration. An exhaustive search would result in an exponential time complexity, which may be unacceptable, especially since resources allocations are adapted at runtime. We therefore propose an algorithm based on dynamic programming that derives optimal solutions for minimizing the maximal response time by using m cores to execute pipeline P_k.

Let $P_{k,j}$ be a *subpipeline* by considering only the pipeline stages from stage S_1 to stage S_j of pipeline P_k. The dynamic programming defines a recursive function $R_k(j, m)$ to store the optimal configurations for the maximal response time minimization for $P_{k,j}$ with (at most) m cores. In other words, let $R_k(j, m)$ be the minimum maximal response

time for executing $P_{k,j}$ on m cores. Moreover, table $F_k(\ell, j)$ is built for all ℓ, j such that $1 < \ell \leq j \leq N_k$, in which

$$F_k(\ell, j) = i_\ell + o_j + \sum_{h=\ell}^{j} c_h. \tag{5}$$

Then, the initial boundary conditions for $R_k(j, 0)$ and $R_k(j, 1)$ are

$$\begin{array}{l} R_k(j, 0) = \infty \qquad \forall j = 1 \ldots N_k \\ R_k(j, 1) = F_k(1, j) \;\; \forall j = 1 \ldots N_k. \end{array} \tag{6}$$

Furthermore, a function $\text{minmax}RF_k(j, m)$ is defined as

$$\text{minmax}RF_k(j, m) = \min_{m-1 \leq \ell < j} \{\max\{R_k(\ell, m-1), F_k(\ell+1, j)\}\}. \tag{7}$$

The recursive function for $R_k(j, m)$ with $m \geq 2$ is defined as

$$R_k(j, m) = \begin{cases} R_k(j, m-1) & j < m \\ \min\{R_k(j, m-1), \text{minmax}RF_k(j, m)\} & j \geq m. \end{cases} \tag{8}$$

First, the maximal response times using one core are calculated for the first $j = 1 \ldots N_k$ stages. Then, the maximal response times for the first $j = 1 \ldots N_k$ stages are calculated for up to two cores. Since the resulting maximal response times of using only one core for the first j stages have already been calculated, it can be easily decided whether to use one or two cores (in one of the possible fusion combinations) for the same j stages. The process is repeated for three cores, knowing in advance if it is optimal to use one or two cores for the first j stages. Hence, the previous result needs to be compared to any new possible fusion for the same j stages, but now utilizing up to three cores. Thus, iteratively, an optimal solution is obtained as all combinations of stages and cores are considered, but the complexity is reduced since the subsolutions do not need to be recalculated.

The space/time complexity is $O(N_k^2)$ for building the table F_k. The time complexity for building an entry $R_k(j, m)$ is $O(j) = O(N_k)$. The size of the table $R_k(j, m)$ is $O(M_k N_k)$. Therefore, the total time complexity is $O(M_k N_k^2)$. The maximal response time by using at most M_k cores for pipeline P_k is stored in $R_k(N_k, M_k)$. Algorithm 1 shows the pseudocode for this dynamic programming problem.

The actual fusions that lead to the optimal result can be derived by backtracking the dynamic programming table or by using an additional *tracking table* $TR_k(N_k, M_k)$ of

ALGORITHM 1: Maximal Response Time Minimization

Data: The computational requirements e, c, and o for the N_k stages of pipeline P_k, and the maximum M_k cores available;
Result: The minimal maximal computational requirements using at most M_k cores;
Initialize table $F_k(\ell, j)$ according to Equation (5), $\forall(\ell, j)$ such that $1 \leq \ell \leq j \leq N_k$;
for $m = 0$ *to* M_k **do**
  for $j = 1$ *to* N_k **do**
    if $m \leq 1$ **then**
      Build $R_k(j, m)$ according to Equation (6);
    else
      Build $R_k(j, m)$ according to Equation (8);
    end
  end
end
return $R_k(N_k, M_k)$;

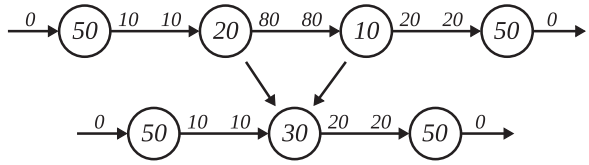


Fig. 5. Pipeline example.

Table I. Example Tables from Algorithm 1

$R_k(4,4)$: Worst-case response

m \ j	**1**	**2**	**3**	**4**
1	60	150	100	130
2	60	110	60	90
3	60	110	60	70
4	60	110	60	70

$TR_k(4,4)$: Tracking

m \ j	**1**	**2**	**3**	**4**
1	0	0	0	0
2	1	1	1	1
3	1	2	3	3
4	1	2	3	4

size $O(M_kN_k)$. When building $TR_k(j,m)$, each cell holds the j^* value of the subsolution that makes the programming optimal. For the initial condition $m = 1$, $TR_k(j,m)$ is set to zero. When $j < m$, or when $j \geq m$ and $R_k(j,m-1)$ turn out to be minimal, then $TR_k(j,m) = j$. In the case where an additional core provides improvement, $TR_k(j,m)$ will be set to the index ℓ from Equation (7) that made this improvement possible, and therefore $TR_k(j,m) \neq j$.

The fusions that provide an optimal maximal response time can be derived from table $TR_k(N_k, M_k)$ as follows: starting from cell $(j,m) = (N_k, M_k)$, the table is traversed in the direction $(TR_k(j,m), m-1)$.

If $TR_k(j,m) = j$, this means that it is not possible to assign more cores to the pipeline since no finer granularity can be achieved or that no additional core may improve the throughput and the subsolution that uses one less core was already optimal.

If $TR_k(j,m) \neq j$, an additional core provides improvement, so if $TR_k(j,m)+1 = j$, then stage S_j is allocated to one core, and if $TR_k(j,m)+1 < j$, all stages between $TR_k(j,m)+1$ and j (both inclusive) should be fused.

Example. Given the pipeline k of Figure 5 with $N_k = 4$ stages and $M_k = 4$ cores, table $F_k(l,j)$ is built according to Equation (5), as stated in Algorithm 1:

$$\begin{array}{lll} F_k(1,1) = 60 & F_k(2,2) = 110 & F_k(3,3) = 110 \\ F_k(1,2) = 150 & F_k(2,3) = 60 & F_k(3,4) = 140 \\ F_k(1,3) = 100 & F_k(2,4) = 90 & \\ F_k(1,4) = 130 & & F_k(4,4) = 70. \end{array}$$

After deriving the initial conditions for $R_k(j,m)$ according to Equation (6), the tracking table $TR_k(j,m)$ for $m = 0, 1$ is filled with zeros.

Table $R_k(j,m)$ is built for $m \geq 2$ according to Equation (8). With $m = 2$, the solution for every subpipeline chooses to use the result from $R_k(1,1)$ and to fuse the rest of the stages. Hence, table $TR_k(j,2)$ contains $j^* = 1$ for any j.

The results are shown in Table I. The solution is derived by traversing table $TR_k(j,m)$ from cell $(j,m) = (4,4)$ in the direction $(j^*, m-1)$: stages S_2 and S_3 are fused, and stages S_1 and S_4 remain as they are.

7.1.1. Proof of Optimality.

PROOF. We prove the optimality by induction. Initially, $\forall j$, $R_k(j, 1)$, is optimal because all stages are fused according to $F_k(1, j)$.

Suppose that $R_k(j', m)$ stores the optimal solution to minimize the maximal response time for a given $j' = 1, 2, \ldots, j$ and m. We will prove that $R_k(j, m+1)$ is optimally derived by using Equation (7). Assume for contradiction that Equation (7) does not give an optimal solution to minimize the maximal response time for allocating the first j stages of application P_k on $m+1$ cores. In the optimal solution, there must be a core executing the j-th stage of application P_k, in which the ℓ'-th stage is the first stage fused with the j-th stage on the same core, $\ell' \leq j$. Therefore, according to the definition, we know that the maximal response time of the preceding optimal solution is the maximum between $F_k(\ell', j)$ and the maximal response time on the rest of m cores for executing the first $\ell' - 1$ stages of application P_k. Therefore, the assumption that Equation (7) does not give an optimal solution implies that the maximal response time on the rest of the m cores for executing the first $\ell' - 1$ stages implies that there exists $j' = 1, 2, \ldots, j$ and m in which $R_k(j', m)$ is not optimal, which contradicts the assumption.

As a result, by the mathematical induction hypothesis, we reach the conclusion that $R_k(j, m)$ is optimal $\forall j, \forall m$. □

7.2. Centralized Resource Allocation

With the dynamic programming of Section 7.1, the overall weighted system throughput can be maximized in a centralized manner. Suppose that $R_k(N_k, m)$ for $m = 1, 2, \ldots, \min\{N_k, M\}$ has been built. For notational brevity, if $N_k < M$, we define $R_k(N_k, m) = R_k(N_k, N_k)$ for any $m \geq N_k$. Let $G(k, m)$ be the maximum centralized weighted system throughput for the first k pipelines based on any arbitrary order of pipelines on at most m cores. Moreover, when there is no feasible solution (i.e., $k > m$), $G(k, m)$ is defined as $-\infty$. Then, the initial (boundary) condition for $G(1, m)$ is

$$G(1, m) = \frac{w_1}{R_1(N_1, m)} \qquad \forall m = 1, 2, \ldots, M. \tag{9}$$

The recursive function for $G(k, m)$ with $k \geq 2$ is expressed in Equation (10). The time complexity, provided that $R_k(N_k, m)$ is known, is $O(KM^2)$. Note that the last column of R_k (i.e., $R_k(N_k, m) \forall m = 1, 2, \ldots, M$) contains the application's weighted throughput.

ALGORITHM 2: Maximizing Overall Weighted System Throughput

Data: The maximum number of available M cores. For every pipeline P_k, the weights w_k and tables $R_k(N_k, m)$ for $m = 1, 2, \ldots, M$;
Result: Maximum overall weighted system throughput for K pipelines, using at most M cores;
for $k = 1$ *to* K **do**
  for $m = 1$ *to* M **do**
    if $k = 1$ **then**
      Build $G(k, m)$ according to Equation (9);
    else
      Build $G(k, m)$ according to Equation (10);
    end
  end
end
return $G(K, M)$;

Table II. Example Tables of Different Pipelines

$R_1(3,6)$

m	$R_1(3,m)$
1	130
2	90
3	70
4	70
5	70
6	70

$R_2(5,6)$

m	$R_2(5,m)$
1	120
2	110
3	100
4	90
5	80
6	80

$R_3(4,6)$

m	$R_3(4,m)$
1	300
2	300
3	80
4	40
5	40
6	40

Algorithm 2 shows a pseudocode for this dynamic programming.

$$G(k,m) = \begin{cases} -\infty & k > m \\ \max\limits_{k-1 \le m' < m} \left\{ G(k-1,m') + \frac{w_k}{R_k(N_k, m-m')} \right\} & k \le m \end{cases} \tag{10}$$

An additional tracking table $TG(K,M)$ of size $O(KM)$ allows for deriving how many cores should be allocated to each pipeline. Each cell of $TG(k,m)$ holds the m^* value of the subsolution that makes the solution optimal. For the initial condition $k = 1$, $TG(k,m)$ is set to zero. When $k > m$, then $TG(k,m) = -1$. In the case where $k \le m$, $TG(k,m)$ will be set to the value of m' from Equation (10) that made this subsolution optimal.

Once table $TG(K,M)$ has been built, the number of cores for each pipeline can be derived from it. Starting from the final cell $(k,m) = (K,M)$, the table is traversed in the direction $(k-1, TG(k,m))$. If $TG(k,m) = -1$, then there is no feasible solution for this set of values. In any other case, cores between $TG(k,m)+1$ and m (both inclusive) should be assigned to application k.

Example. Given the pipelines R_1, R_2, and R_3 shown in Table II, with weights $w_1 = w_2 = w_3 = 10{,}000$ and having up to $M = 6$ cores, $G(k,m)$ is calculated as follows. First, the initial conditions are calculated according to Equation (9). The tracking table $TG(k,m)$ for $k = 1$ holds no value for m^* and is therefore filled with zeros.

Table III contains $G(k,m)$ according to Equation (10). The solution can be derived by traversing $TG(k,m)$ from cell $(k,m) = (3,6)$ in the direction $(k-1, m^*)$: one core is allocated to R_1, one core to R_2, and four cores to pipeline R_3.

7.3. Distributed, Hierarchical Resource Allocation

The method for resource allocation presented in Section 7.2 is designed in a centralized manner. This requires global knowledge and allocates the resources to all running applications en bloc. This leads to a quadratic time complexity, which may be infeasible for large systems. To achieve a highly scalable solution (i.e., its overhead should not grow faster than the system size), this section proposes distributed, hierarchical resource allocation for which the pipelines are grouped into several independent clusters. Clusters are grouped hierarchically into larger clusters and so on, therefore constructing a tree, as shown in Figure 6.

Definition of the tree structure. There are K_0 pipelines $P_1, P_2, \ldots, P_{K_0}$ on level 0, which form the leaves of the tree. They are clustered hierarchically in L levels. For for each

Table III. Example Tables from Algorithm 2

$G(3,6)$: Overall Performance

m \ k	1	2	3
1	76.92	$-\infty$	$-\infty$
2	111.11	160.26	$-\infty$
3	142.86	194.44	193.59
4	142.86	226.19	227.78
5	142.86	233.76	285.26
6	142.86	242.86	410.25

$TG(3,6)$: Tracking

m \ k	1	2	3
1	0	-1	-1
2	0	1	-1
3	0	2	2
4	0	3	3
5	0	3	2
6	0	3	2

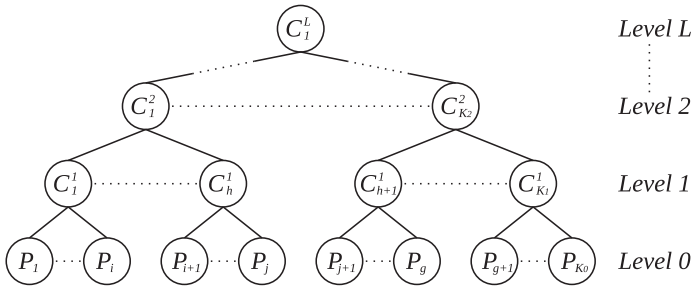


Fig. 6. Distributed solution. Applications are clustered at *Level*0, and these clusters are clustered hierarchically to form a tree.

cluster C_i^ℓ, ℓ denotes index of the cluster on level i. All clusters on level 1 ($\ell = 1$) are adjacent to the pipelines. Hence, there are L levels in the tree, where level L is the root of the tree, and each level ℓ holds K_ℓ nodes.

With this distributed model, the solution from Section 7.1 is used to build the tables $R_k(N_k, M)$ for every pipeline P_k, where M continues to be the total amount of cores available in the system. In the following, we will detail how the Equations (9) and (10) can be adapted in a distributed, hierarchical way.

Each cluster C_i^1 (level 1) contains the information of the weights w_k and tables $R_k(N_k, M)$ of its children (pipelines) and utilizes the solutions of Section 7.2 to build the corresponding tables $G(K^*, M)$, where K^* is the number of child nodes of the cluster. This table contains the best configuration for cluster C_i^1 by allocating $m = 1, 2, \ldots, M$ cores to its children pipelines, independently of the other clusters of the same level.

Similarly, the clusters C_i^2 (level 2) contain the information of table $G(K^*, M)$ of its child clusters C_i^1 (level 1). This applies likewise to all upper levels. In this way, each level allocates cores among its children based solely on this (limited) information. Consequently, the computational requirement is distributed hierarchically among the system. When a cluster C_i^j (re)computes $G(K^*, M)$, this table must be (re)computed for C_i^{j-1} and for all C_i^{j+1}. When this is not performed immediately (as is the case in our implementation), the results may no longer be optimal.

$G_i^\ell(k,m)$ denotes the table for the modified version of the dynamic programming in Section 7.2. Considering that w_1^*, R_1^*, N_1^* are the parameters of the first child (pipeline) of node C_1^1, node $C_{1^*}^{\ell-1}$ is the first child of node C_i^ℓ, value $K_{\ell-1}^*$ is the number of children of node C_i^ℓ, and value $K_{\ell-2}^*$ is the number of children of node $C_{1^*}^{\ell-1}$ and node $C_k^{\ell-1}$, then the initial conditions of $G_i^\ell(1,m)$ are analogous to Equation (9):

$$\begin{aligned} G_i^\ell(1,m) &= \tfrac{w_1^*}{R_1^*(N_1^*,m)} \qquad \forall m = 1,2,\ldots,M \text{ when } \ell = 1 \\ G_i^\ell(1,m) &= G_{1^*}^{\ell-1}(K_{\ell-2}^*,m) \;\; \forall m = 1,2,\ldots,M \text{ when } \ell \geq 2, \end{aligned} \tag{11}$$

the value of $G_i^\ell(k,m)$ is set to $-\infty$ whenever $k > m$, the recursive function when $\ell = 1$ and $k \leq m$ is

$$G_i^\ell(k,m) = \max_{k-1 \leq m' < m} \left\{ G_i^\ell(k-1,m') + \tfrac{w_k}{R_k(N_k,m-m')} \right\}, \tag{12}$$

the recursive function when $\ell \geq 2$ and $k \leq m$ is

$$G_i^\ell(k,m) = \max_{k-1 \leq m' < m} \left\{ G_i^\ell(k-1,m') + G_k^{\ell-1}\left(K_{\ell-2}^*, m-m'\right) \right\}, \tag{13}$$

and finally, the result is found in cell $G_i^\ell(K_{\ell-1}^*, M)$.

For $\ell \geq 2$, a cluster C_i^ℓ allocates cores to its $K_{\ell-1}^*$ children. Hence, similarly to the recursive function of Equation (10), Equation (12) calculates G_i^ℓ for $\ell = 1$ based on the solution of the problem of Section 6.2 for the applications R_k, and Equation (13) recursively calculates G_i^ℓ for the clusters of $\ell \geq 2$.

It is important to note that even though the root node makes decisions that affect every pipeline, this is still distributed and scalable resource allocation, as every node only contains the partial information of its children.

7.4. Resilience against Malfunctioning Cores

When the reliability of a system cannot be guaranteed, cores may *malfunction* at runtime. In the following, we illustrate how unpredictably malfunctioning cores can lead to a significantly decreased system throughput and how the *resilience* of our distributed resource allocation can be increased.

Fault model. In the following, we assume that cores may *malfunction* unpredictably at any time. As a consequence, all tasks assigned to them crash. This has the following consequences. (1) First, they stop to respond to other clusters. (2) Second, if a cluster head is assigned to a malfunctioning core, it is terminated and all of its data is lost. (3) Third, during runtime, it is assumed that a malfunctioning core does not resume operation.

In the following, we refer to the task that forms the instance of a cluster as defined in Section 7.3 as a *cluster head*. When a cluster head C_i^l is allocated to a malfunctioning core, the integrity of the hierarchical tree structure (Section 7.3) of clusters is corrupted. The reason is that all C_*^{l-1} become roots of subtrees. Figure 7 shows an example how the integrity of the hierarchical tree of clusters can be corrupted (a) and how this leads to disconnected subtrees (b).

As a result, our distributed resource allocation is still able to (re)allocate cores in the remaining subtrees. However, there is no exchange of cores among subtrees. To illustrate how this may lead to a significantly decreased system throughput, let us consider a case of two disconnected subtrees A and B, and each subtree contains only one application, a in A and b in B. When the resource requirements of a increase and the requirements of b remain the same (or decrease), assigning cores from B to A would increase the system throughput. However, as A and B are disconnected, this

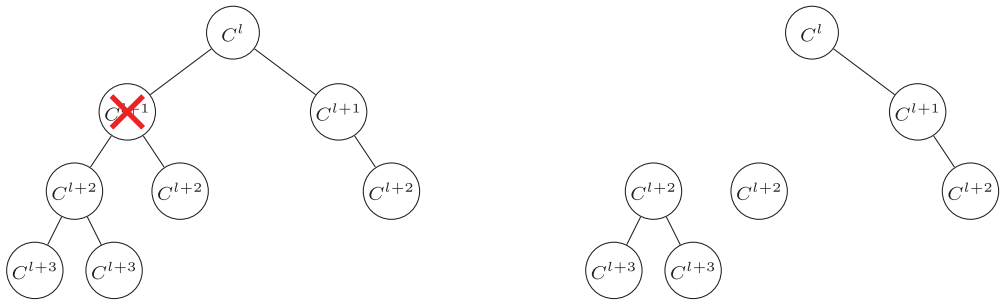
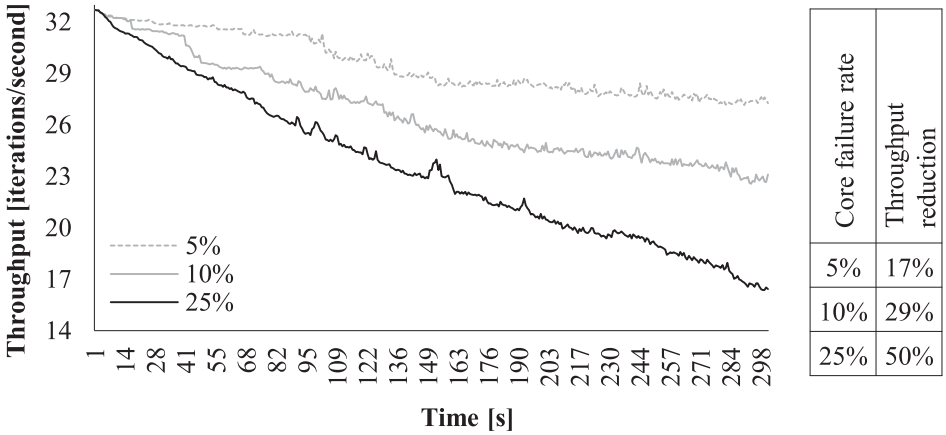


Fig. 7. Effect of malfunctioning cores that corrupt the integrity of the hierarchical tree: a malfunctioning core "removes" a cluster head and thus leaves disconnected subtrees.



Core failure rate	Throughput reduction
5%	17%
10%	29%
25%	50%

Fig. 8. Effect when 5%, 10%, and 25% of cores malfunction over a period of 300 seconds in a 1,024-core system running 275 applications. The resulting disconnected subtrees lead to significantly decreasing throughput.

is not possible. This is also the case for more than one application per subtree and for more than two subtrees. Thus, disconnected subtrees may lead to core distributions that result in a decreased throughput. Simulations of a 1,024-core system where cores malfunction randomly during a 300-second interval show that terminated cluster heads can result in significantly reduced system throughputs of 17%, 29%, and 50% for a malfunction rate of 5%, 10%, and 25% of the cores, respectively, as shown in Figure 8. To address this problem, we propose employing self-organization to restore the integrity of the hierarchical structure of cluster heads even when cores malfunction. Self-organization of cluster heads is characterized as follows:

—A cluster head detects when its parent cluster head is terminated (i.e., it does not respond).
—When its parent is terminated, a cluster head searches for another cluster head (that is not among its children) by sending a connection request to randomly selected cores.
—Once such a cluster head is found, both cluster heads join their subtrees.
—This way, the integrity of the hierarchical tree is restored.

Algorithm 3 shows how this can be achieved. When a cluster head h finds that its parent is terminated, it starts to resolve this in parallel to its duties of allocating cores among its children. Until all cores have been searched or a new parent is found, the cluster head tries to establish a connection to a random core. If a cluster head h' is

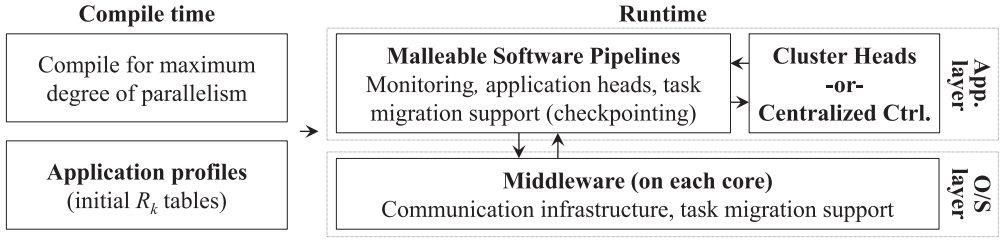


Fig. 9. Overview of our implementation. At compile time, the applications are compiled for a maximum degree of parallelism, whereas initial R_k tables are obtained through profiling. Resource allocation is performed at runtime through application-layer control and task migration support by the middleware.

ALGORITHM 3: Self-Organization to Restore the Integrity of the Hierarchical Structure of Cluster Heads

```
Data: Current cluster head h
while Parent terminated AND not all cores searched do
    c = Random core;
    if a cluster head h' is assigned to c then
        if h' does not contain h among its children then
            set h' as new parent;
        end
    end
end
```

assigned to this core and h' is not among its children, it *joins* h'—that is, h' lists h among its children, and h sets h' as its new parent. When this is achieved, it stops searching for a new parent. A handshake protocol prevents corner cases of two subtrees that both search for a new parent to join each other.

This way, the integrity of the hierarchical structure of cluster heads of our distributed resource allocation can be restored.

8. IMPLEMENTATION DETAILS

In the following, we discuss the components of our methods for resource allocation and their implementation details. We have implemented our resource allocation on Intel's Single-Chip Cloud Computer (SCC) [Howard et al. 2010] and in a high-level system simulator detailed in Section 9.1. Our resource allocation employs several components written in C++ that communicate by exchanging network messages.

8.1. Components

The *centralized resource allocation* employs application heads and a centralized controller (Figure 9). Each application denotes one of its cores to form its application head (this core may execute a stage). The application head registers and signs off the application with the centralized controller on starting and stopping of the application. To register an application, the application head sends a message including a unique identifier of the application (4 bytes), its number of stages (4 bytes), and an initial R_k table of Section 7.1 (4 bytes $\cdot$ N_k stages $\times$ M_k cores). During runtime, application heads recompute their R_k table when the values of i_i, c_i, or o_i for one of their stages change and send this to the centralized controller. The centralized controller (re)computes the optimal allocation of cores and sends a list of cores to each application. Based on this list, the application heads fuse and migrate their stages.

The *distributed resource allocation* employs application heads (see previous discussion) and cluster heads: for each cluster, a cluster head receives the R_k tables (4 bytes

for each entry) from each of its children, which may be either cluster heads themselves or application heads. However, instead of calculating the resource allocations globally, cluster heads only calculate resource allocations for their children and propagate the combined R_k tables to their parent.

8.2. Implementation of Task Migration

Task migration is required to fuse stages at runtime. It is carried out on application level with a lightweight support by the middleware. The middleware is responsible to load applications as needed. When the controlling resource allocation (this applies both to the centralized and distributed resource allocation) chooses to change the fusions or reallocates cores to applications, the respective stages are notified by the middleware. When the corresponding stage reaches a checkpoint (after completing an iteration), it saves its state and requests the middleware at the destination core to load the executable file (if it is not already running). Then, it sends the saved state to the destination core, which then continues the execution of the (fused) stage procedure. The corresponding overheads of these operations are evaluated and discussed in Section 9.6.

9. EXPERIMENTAL RESULTS

In the following, we describe the system setup of our experiments, our benchmark scenario, and the results on the achieved system throughput and the involved overheads of our resource allocation.

9.1. System Setup

Our experiments have been conducted on Intel's SCC [Howard et al. 2010] and by using a high-level many core simulator. The SCC is a platform that integrates 48 identical x86 cores in 24 tiles (two cores each) on a single chip. The individual P54C cores (45nm process) run at 800MHz, are connected via a 2GHz network-on-chip with a bisection bandwidth of 2TB/s. Each core has 16KiB of instruction- and 16KiB of data cache, as well as 256KiB of unified instruction/data L2 cache. It runs a single-core Ubuntu Linux (kernel 3.1.4) on each core.

Our high-level many-core simulator executes task traces collected on the SCC and simulates the network-on-chip interconnect. For experiments of large systems, simulated instances of the cores, network-on-chip routers, memories, and interconnect of the SCC are replicated. This does not require changing the number of ports of a router, the buffer size, and so forth due to the mesh topology of the interconnect of the SCC. There is no hardware cache coherence on the SCC. Our simulations follow this paradigm. This way, the networks-on-chip are simulated to correspond to the physical properties of the SCC—that is, each router performs X-Y-routing and transmits data at 800MHz with a two-cycle latency and a flit size of 32 bytes. Corresponding the the SCC, simulated routers arbitrate the communication links and queue transfers when required. This way, our simulator delivers accurate information on the application/system throughputs and on the communication volumes/overheads (algorithm runtimes have been collected on the SCC). It runs on a six-core AMD Opteron™ 8431 CPU (2.4GHz) with 64GB DDR3 RAM. The SCC allows measuring the computational overhead of our resource allocation accurately, but considering that it integrates 48 cores, we cannot analyze the system throughputs and the communication overhead for larger systems. However, we measured the computational overhead on the SCC even for (virtually) large systems because these computations do not demand physical disposal of cores.

Measurements conducted on the SCC were as follows:

Table IV. Benchmark Applications

Name	Stages	Source
automotive	21	See Section 9.2.
h264ref	4	SPEC CPU 2006 [Henning 2006]
lame	4	MiBench [Guthaus et al. 2001]
PGP	5	MiBench [Guthaus et al. 2001]
sphinx3	22	SPEC CPU 2006 [Henning 2006]

—Computational overhead for up to 1,024 cores
—Throughput of the centralized resource allocation for up to 48 cores
—Fusion overheads.

Experiments conducted using our simulator were as follows:

—Communication overhead
—Throughput of our centralized and our distributed resource allocation.

For the experiments, the benchmark applications listed in Table IV are started multiple times so that the total number of stages in the system exceeds the number of cores by at least a factor of 3 (we chose this number arbitrarily to establish a considerable system load).

9.2. Benchmark Scenario and Adaption of the State of the Art

Table IV shows an overview of the benchmark applications and their number of stages. The applications have been parallelized manually to form malleable software pipelines. We chose this set of applications because they are complex, communicate heavily, and are well suited to form software pipelines.

The automotive application is a vision-based application that takes its algorithms from the IVT library [Azad et al. 2008]. It performs stereo vision, image enhancement, object recognition (based on scale-invariant feature transform (SIFT) and Harris corner detection), morphological operations, and pattern matching algorithms to identify and track objects in a continuous stream of color stereo video data (648×480 pixels at 30fps). The other applications have been taken from the respective benchmark suites.

To achieve a fair comparison with our proposed resource allocation, we adapted the state-of-the-art methods [Bahi et al. 2005] and [Kobbe et al. 2011].

AIAC. AIAC exchanges workload between physically neighboring cores to balance the computational load evenly. To adapt AIAC [Bahi et al. 2005] to software-pipelined applications in many-core systems, we exchange workload by migrating pipeline stages when the computational load is not balanced. This is achieved by comparing the load of adjacent cores and migrating a pipeline stage i when the difference of the summed computational requirements among all stages on each core exceeds c_i. To achieve a fair comparison, we relax the assumption that only consecutive stages may be allocated to the same core. For our implementation of AIAC, a core may execute any stage from any application.

DistRM [Kobbe et al. 2011]. This distributed resource allocation distributes cores among applications but relies on the applications to decide themselves how to distribute their tasks. Therefore, we use our optimal fusion algorithm from Section 7.1 to achieve a fair comparison. Consequently, only the number of cores assigned to each application differs between DistRM and our resource allocation, whereas the fusions of pipeline stages are carried out identically. We adapt DistRM by using the tables shown in Section 7.1. As DistRM remains in local optima if the speedup of an application does not increase with another core (even if this was the case for a larger number of additional

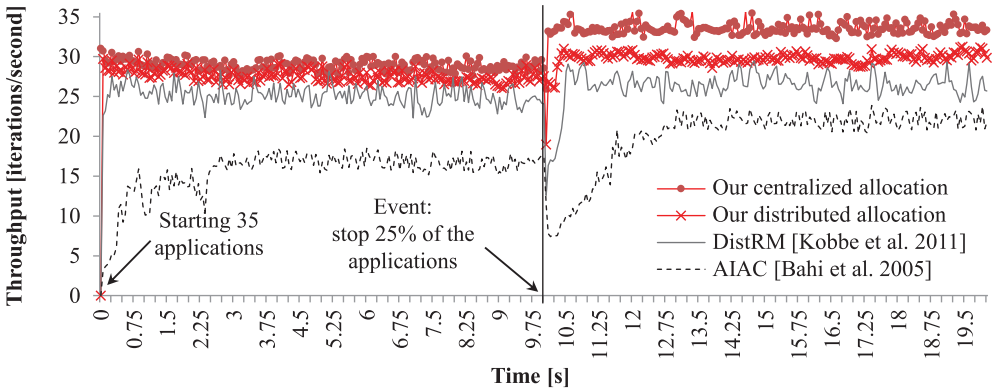


Fig. 10. Comparison of the system throughput that is achieved by our solutions and the state of the art.

cores), we report marginal improvements (we choose an $\epsilon = 5 * 10^{-4}$) as long as the number of cores does not exceed the number of stages of the corresponding application. Using the described adaptations, we can achieve a fair comparison with DistRM.

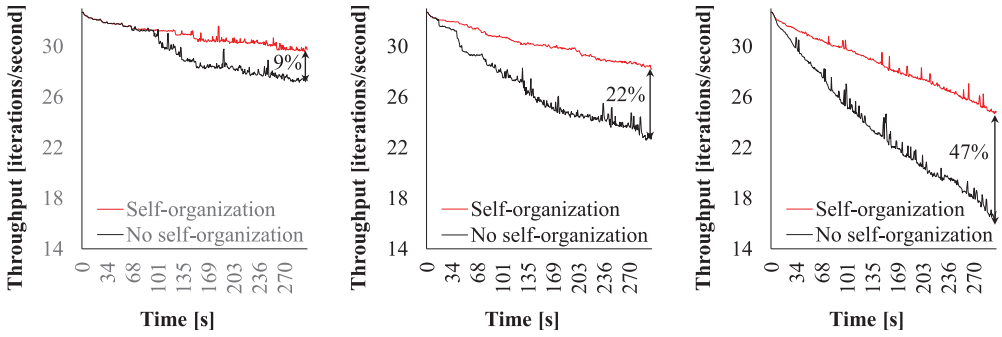
9.3. System Throughput

In the following, we compare the throughput achieved by our distributed allocation with our centralized resource allocation (thus against optimal resource allocation) and with two state-of-the-art methods for runtime resource allocation: DistRM [Kobbe et al. 2011] and AIAC [Bahi et al. 2005]. Figure 10 shows the average system throughput over 50 runs when running seven instances of each benchmark application (35 applications, or 392 stages in total) on 128 cores, connected by a network-on-chip mesh as provided by the SCC. To show how each method gradually improves the resource allocation, we initially start all stages on a single core and let the corresponding resource allocation improve the system throughput incrementally. Then, 25% of the applications are randomly stopped at $t = 10$ seconds. Whereas our centralized resource allocation achieves an increased throughput of roughly 13.7%, the average system throughput drops for the other methods from ca. 17 to 29 iterations/second to ca. 9 to 17 iterations/second because without adapting the resource allocations, the cores that formerly executed the stopped applications are now idle. The resource allocation then improves the throughput by adapting the allocation of resources. Our distributed resource allocation achieves a system throughput of 94.78% compared to our optimal (centralized) resource allocation. On average, the distributed resource allocation increases the throughput by 11.3% and 60.6% over Kobbe et al. [2011] and Bahi et al. [2005], respectively. We thus conclude that our distributed resource allocation is near optimal, and that both methods for allocating resources increase the throughput significantly over the state of the art.

9.4. Improved Resilience through Self-Organization

Figure 11 shows how the resilience of our distributed resource allocation against unreliable cores is improved through self-organization. In a period of 300 seconds, 5% (a), 10% (b), and 25% (c) of the cores of a 1024-core system (in a benchmark scenario according to Section 9.2) are shut down randomly. This way, runtime faults of cores are simulated.

Such a scenario could be caused, for example, by exceptionally high on-chip temperatures due to a malfunctioning cooling system or fan. Through self-organization, an increased system throughput of 9% (a), 22% (b), and 47% (c) can be achieved. Hence,



(a) 5% of cores malfunction (b) 10% of cores malfunction (c) 25% of cores malfunction

Fig. 11. Resilience against unreliable cores can be improved through self-organization. In a period of 300 seconds, 5%, (a), 10% (b), and 25% (c) of the cores of a 1,024-core system are randomly shut down to simulate malfunction at runtime.

Centralized [s]

Cores	Applications					
	4	8	16	32	64	128
4	0.00	(0.01)	(0.02)	(0.04)	(0.09)	(0.19)
8	0.01	0.01	(0.03)	(0.06)	(0.14)	(0.30)
16	0.01	0.02	0.05	(0.10)	(0.23)	(0.46)
32	0.01	0.03	0.07	0.15	(0.32)	(0.72)
64	0.02	0.05	0.12	0.27	0.49	(1.10)
128	0.05	0.11	0.22	0.51	1.00	1.92
256	0.11	0.25	0.56	1.12	2.19	4.14
512	0.29	0.69	1.39	2.81	5.73	11.33
1024	0.93	1.88	4.32	9.21	17.89	37.32

Distributed [ms]

Cores	Runtime [ms]
16	0.18
32	0.30
64	0.55
128	0.44
256	0.63
512	0.57
1024	0.82

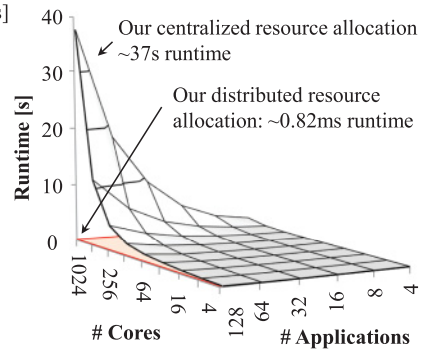


Fig. 12. Computational overhead of our methods for resource allocation. Infeasible combinations of applications/cores in brackets. Only one column is shown for our distributed resource allocation, as the runtime does not change significantly with the number of applications.

we find that self-organization can increase the resilience of our distributed approach significantly.

9.5. Overheads

Figure 12 shows how the computational overhead of our centralized resource allocation grows with a growing number of cores and growing number of applications. Up to a considerable problem size (e.g., 64 cores, less than 64 applications), optimal resource allocations can be calculated in less than 0.5 seconds, but this overhead is significantly larger for larger systems or more applications as the computational overhead grows quickly beyond 37 seconds. We find that this is not acceptable when applications may be started/stopped at any time or when the resource requirements of applications change frequently.

The distributed resource allocation has a constant time complexity, as each cluster head on level 1 C_h^1 only calculates the optimal distribution of the cores to its children (which does not grow with the problem sizes). Thus, its computational overhead is small (less than 0.1ms for 1,024 cores).

We measured the memory overhead of our centralized resource allocation to be approximately 24KiB. The maximum memory overhead of any node of our distributed resource allocation is approximately 4KiB.

Cores	16	32	64	128	256	512	1024
Applications	5	10	20	35	70	140	275
Stages	56	112	224	392	784	1568	3080
Total Application Comm. Volume [MB/s]	28.3	56.6	98.9	198.1	367.3	792.4	1455.0
Overhead (Centralized Allocation) [KB/s]	0.64	2.27	5.53	15.2	39.2	178.5	365.3
Overhead (Distributed Allocation) [KB/s]	0.69	1.58	3.34	6.79	19.0	54.2	138.0

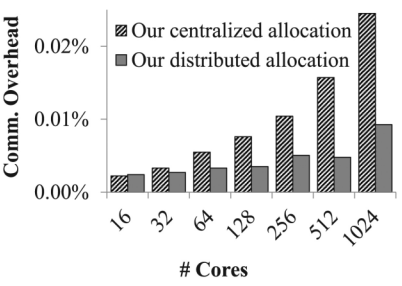


Fig. 13. Comparison of the communication overhead of our methods for resource allocation for different numbers of cores.

Table V. Overheads of Fusion Operations

	Carried State [KB]				Overhead [ms]	
Application	Min	Max	Avg	σ	Old Core	New Core
automotive	1	32	19	15.21	0.63	22
h264ref [Henning 2006]	13	53	27	22.73	1.07	76
lame [Guthaus et al. 2001]	9	10	9	1.32	0.18	19
PGP [Guthaus et al. 2001]	1	27	12	9.11	0.30	66
SPHINX3 [Henning 2006]	12	22	17	4.21	0.51	44

Figure 13 compares the total communication overhead of our centralized and our distributed resource allocation. This overhead includes status updates and notifications; the updates of the i_i, c_i, and o_i values; and the propagation of all tables and speedup vectors. As this overhead merely reaches around 365.3KiB/s (0.025% of the total communication for a system with 1,024 cores, 275 applications) for our centralized resource allocation and roughly 138KiB/s (0.009%) for our distributed resource allocation, we consider it negligible.

9.6. Fusion Overhead

Table V summarizes the overhead for fusions—that is, the overhead of the required task migrations—of two stages of each application (collected on Intel's SCC). When the fusion incurs an old core (i.e., the application is already running on this core before this operation), the overhead is limited to transferring the carried state of the stage and is thus small (on average less than 0.6ms). Otherwise, the executable file of the application needs to be started, which takes considerably more time (roughly 45ms on average). However, our experiments show that this is only the case in less than 5% of all conducted fusions.

The resulting average fusion overhead is less than 2.4ms, which allows adaptation of resource allocations frequently.

We therefore conclude that the overhead of our proposed methods is small, and thus we find that our methods for resource allocation are well suited for managing the resource allocation of many-core systems at runtime.

10. CONCLUSION

In this article, we show how resources can be allocated to software pipelines in a way that achieves and maintains high system throughput even in large many-core systems despite unpredictable, significant variances in the demand for both computational and

communication resources. This is achieved by optimizing the configurations (fusion of stages) of software pipelines and the distribution of cores among them at runtime. Furthermore, we show how optimality can be sacrificed to achieve a high degree of scalability in many-core systems with hundreds of cores. Problems that arise from unreliable cores in current and future many-core systems are addressed by employing self-organization. We show how self-organization can help to effectively counteract the impaired system throughput that may arise from failing cores.

REFERENCES

Pedram Azad, Tilo Gockel, and Rudiger Dillmann. 2008. *Computer Vision: Principles and Practice*. Elektor.

Jacques M. Bahi, Sylvain Contassot-Vivier, and Raphael Couturier. 2005. Dynamic load balancing and efficient load estimators for asynchronous iterative algorithms. *IEEE Transactions on Parallel and Distributed Systems* 16, 4, 289–299.

Mohamed A. Bamakharma and Todor Stefanov. 2012. Managing latency in embedded streaming applications under hard-real-time scheduling. In *Proceedings of the ACM International Symposium on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. 83–92.

Shekhar Borkar. 2005. Designing reliable systems from unreliable components: The challenges of transistor variability and degradation. *IEEE Micro* 25, 6, 10–16.

David Brooks, Robert P. Dick, Russ Joseph, and Li Shang. 2007. Power, thermal, and reliability modeling in nanometer-scale microprocessors. *IEEE Micro* 27, 3, 49–62.

Ewerson Carvalho, Ney Calazans, and Fernando Moraes. 2007. Heuristics for dynamic task mapping in NoC-based heterogeneous MPSoCs. In *Proceedings of the IEEE/IFIP International Workshop on Rapid System Prototyping (RSP)*. 34–40.

Jeronimo Castrillon, Andreas Tretter, Rainer Leupers, and Gerd Ascheid. 2012. Communication-aware mapping of KPN applications onto heterogeneous MPSoCs. In *Proceedings of the IEEE/ACM Design Automation Conference (DAC)*. 1262–1267.

Weijia Che and Karam S. Chatha. 2012. Unrolling and retiming of stream applications onto embedded multicore processors. In *Proceedings of the IEEE/ACM Design Automation Conference (DAC)*. 1272–1277.

Junchul Choi, Hyunok Oh, Sungchan Kim, and Soonhoi Ha. 2012. Executing synchronous dataflow graphs on a SPM-based multicore architecture. In *Proceedings of the IEEE/ACM Design Automation Conference (DAC)*. 664–671.

Dror G. Feitelson, Larry Rudolph, Uwe Schwiegelshohn, Kenneth C. Sevcik, and Parkson Wong. 1997. In *Proceedings of the Job Scheduling Strategies for Parallel Processing (IPPS'97)*. 1–34.

Matthew Guthaus, Jeff Ringenberg, Todd Austin, Trevor Mudge, and Richard Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Workshop on Workload Characterization (WWC-4)*. 3–14.

John L. Henning. 2006. SPEC CPU2006 benchmark descriptions. *SIGARCH Computer Architecture News* 34, 4, 1–17.

Jason Howard, Saurabh Dighe, Yatin Hoskote, Sriram Vangal, David Finan, Gregory Ruhl, David Jenkins, Howard Wilson, Nitin Borkar, Gerhard Schrom, Fabrice, Shailendra, Tiju Jacob, Satish Yada, Sraven Marella, Praveen Salihundam, Vasantha Erraguntla, Michael Konow, Michael Riepen, Guido Droege, Joerg Lindemann, Matthias Gries, Thomas Apel, Kersten Henriss, Tor Lund-Larsen, Sebastian Steibl, Shekhar Borkar, Vivek De, Rob Van Der Wijngaart, and Timothy Mattson. 2010. A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS. In *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC)*. 108–109.

Markus C. Huebscher and Julie A. McCann. 2008. A survey of autonomic computing—degrees, models, and applications. *ACM Computing Surveys* 40, 3, 7:1–7:28.

Janmartin Jahn and Jörg Henkel. 2013. Pipelets: Self-organizing software pipelines for many core architectures. In *Proceedings of the IEEE/ACM International Conference on Design, Automation, and Test in Europe (DATE)*. 1516–1521.

Janmartin Jahn, Santiago Pagani, Sebastian Kobbe, Jian-Jia Chen, and Jörg Henkel. 2013. Optimizations for configuring and mapping software pipelines in many core systems. In *Proceedings of the IEEE/ACM Design Automation Conference (DAC)*. Article No. 130.

Kevin Klues, Barret Rhoden, Andrew Waterman, and Eric Brewer. 2010. Processes and resource management in a scalable many-core OS. In *Proceedings of the USENIX Workshop on Hot Topics in Parallelism (HotPar)*. 1–6.

Sebastian Kobbe, Lars Bauer, Daniel Lohmann, Wolfgang Schröder-Preikschat, and Jörg Henkel. 2011. DistRM: Distributed resource management for on-chip many-core systems. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'11)*. 119–128.

Karthik Lakshmanan, Ragunathan Rajkumar, and John Lehoczky. 2009. Partitioned fixed-priority preemptive scheduling for multi-core processors. In *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS)*. 239–248.

Haeseung Lee, Weijia Che, and Karam Chatha. 2012. Dynamic scheduling of stream programs on embedded multi-core processors. In *Proceedings of the ACM International Symposium on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. 93–102.

Tong Li, Dan Baumberger, David A. Koufaty, and Scott Hahn. 2007. Efficient operating system scheduling for performance-asymmetric multi-core architectures. In *Proceedings of the International Conference on Supercomputing (ICS)*. 53:1–53:11.

Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. 2009. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. 45–55.

David S. Johnson and Michael R. Garey. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Co.

Vijaykrishnan Narayanan and Yuan Xie. 2006. Reliability concerns in embedded system designs. *IEEE Transactions on Computers* 39, 1, 118–120.

Vincent Nollet, Theodore Marescaux, Prabhat Avasare, Diederik Verkest, and Jean-Yves Mignolet. 2005. Centralized run-time resource management in a network-on-chip containing reconfigurable hardware tiles. In *Proceedings of the IEEE/ACM International Conference on Design, Automation, and Test in Europe (DATE)*. 234–239.

Mohan Rajagopalan, Brian T. Lewis, and Todd A. Anderson. 2007. Thread scheduling for multi-core platforms. In *Proceedings of the 11th USENIX Workshop on Hot Topics in Operating Systems (HOTOS'07)*. 2:1–2:6.

Petar Radojković, Vladimir Čakarević, Miquel Moretó, Javier Verdú, Alex Pajuelo, Francisco J. Cazorla, Mario Nemirovsky, and Mateo Valero. 2012. Optimal task assignment in multithreaded processors: A statistical approach. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 235–248.

Lars Schor, Iuliana Bacivarov, Devendra Rai, Hoeseok Yang, Shin Haeng Kang, and Lothar Thiele. 2012. Scenario-based design flow for mapping streaming applications onto on-chip many-core systems. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*. 71–80.

Amit Kumar Singh, Muhammad Shafique, Akash Kumar, and Jörg Henkel. 2013. Mapping on multi/many-core systems: Survey of current and emerging trends. In *Proceedings of the 50th Annual Design Automation Conference (DAC)*. Article No. 1.

Allan Snavely and Dean Tullsen. 2000. Symbiotic jobscheduling for a simultaneous multithreading processor. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 234–244.

Allan Snavely, Dean M. Tullsen, and Geoff Voelker. 2002. Symbiotic jobscheduling with priorities for a simultaneous multithreading processor. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX)*. 66–76.

Jan Stender, Silvan Kaiser, and Sahin Albayrak. 2006. Mobility-based runtime load balancing in multi-agent systems. In *Proceedings of the International Conference on Software Engineering and Knowledge Engineering (SEKE)*. 688–693.

William Thies, Michal Karczmarek, Michael Gordon, David Maze, Jeremy Wong, Henry Hoffmann, Matthew Brown, and Saman Amarasinghe. 2001. StreamIt: A language for streaming applications. In *Proceedings of the 11th International Conference on Compiler Construction (ICCC)*. 179–196.

John Turek, Joel L. Wolf, and Philip S. Yu. 1992. Approximate algorithms for scheduling parallelizable tasks. In *Proceedings of the Symposium on Parallel Algorithms and Architectures (SPAA)*. 323–332.

Received August 2013; revised August 2014; accepted November 2014