Optimizations for Configuring and Mapping Software Pipelines in Many Core Systems

Janmartin Jahn, Santiago Pagani, Sebastian Kobbe, Jian-Jia Chen, Jörg Henkel
Karlsruhe Institute of Technology (KIT), Germany
{ jahn, santiago.pagani, sebastian.kobbe, j.chen, henkel }@kit.edu

Abstract—Efficiently utilizing the computational resources of many core systems is one of the most prominent challenges. The problem worsens when resource requirements vary unpredictably and applications may be started/stopped at any time. To address this challenge, we propose two schemes that calculate and adapt task mappings at runtime: a centralized, optimal mapping scheme and a distributed, hierarchical mapping scheme that trades optimality for a high degree of scalability. Experiments on Intel’s 48-core Single-Chip Cloud Computer and in a many core simulator show that a significant improvement in system performance can be achieved over current state-of-the-art.

I. INTRODUCTION AND NOVEL CONTRIBUTION

While many core systems offer the potential of vastly increasing computational performance as Moore’s Law continues, in practice there are significant hurdles to actually utilize these resources and to profit from scalability. One key issue is the mapping of applications to the available cores. When considering dynamic runtime scenarios, i.e. when applications may be started, stopped, or when their resource demands (i.e. the computational demands of their tasks and the communication requirements between them) may vary unpredictably at any time (e.g. due to user interactions or changing input data), the problem of mapping tasks is extended from finding mappings to also adapting them at runtime. In such scenarios, it is crucial to adapt mappings to account for such changes in order to maintain high system performance: Section II shows an example how significantly changing resource demands of an application may lead, for an established task mapping, to a degradation of the system throughput of more than 50% compared to an adapted mapping. The same problem may arise when applications are started or stopped unpredictably (e.g. by the user). Such scenarios are increasingly common and thus need to be addressed [16]. To tackle this challenge, one solution is to employ so-called malleable applications that provide the flexibility of using more or less cores dynamically (e.g. [5], [20]). They can change their degree of parallelism at runtime so that a system may re-distribute the cores among applications to increase its performance [10]. Our approach focuses on software pipelines because they are well-established means to parallelize a large class of complex applications. Especially stream-processing applications, among which are very common image/video and networking applications, are well suitable for software pipelining. Multiple approaches to extract software pipelines (semi-)automatically from sequential C code by parallelizing compilers [3], [13], [19] have been presented.

For the rest of this paper, we use the following definitions:

Software pipelines consist of multiple stages, each processing subsequent iterations on a stream of input data. Each stage is an individual task consisting of a working set, program code, and task state. The output data of one stage forms the input data of its direct successor. There is no further communication.

A malleable software pipeline can reduce the number of its stages (thus the number of cores used) at runtime by fusing consecutive stages so they are mapped to the same core (similar to fusing filters in StreamIt [6]). Consequently, no on-chip communication is necessary between them. Fused stages can be split through fissions until the initial degree of parallelism is restored.

We use throughput as a performance metric because software-pipelined applications often run continuously until they are stopped (thus metrics like makespan are not applicable). We define the throughput of an application as the number of iterations it completes per second, and the throughput of a system as the averaged throughputs of all running applications. A formal definition follows in Section IV.

In this paper, task migration is used to denote the transfer of the execution of a task from one core to another. Task remapping, in contrast, refers to the abstraction of deciding about task migrations based on a system model (i.e. the underlying algorithm or heuristic). We target systems with many cores, private, distributed memories and Network-on-Chips. Figure 1 shows this exemplarily.

In this paper, our novel contributions are:

1) We present a centralized, optimal mapping scheme for malleable software pipelines.
2) As an extension, we present a distributed, hierarchical mapping scheme that trades the optimum for a high degree of scalability.

To illustrate the effectiveness of our schemes, we have implemented them on Intel’s Single-Chip Cloud Computer (SCC) [9] and in a high-level many core system simulator. Our centralized scheme requires approx. 60ms for calculating optimal mappings for 48 cores, while our distributed scheme calculates near-optimal mappings for 1024 cores in less than 1ms.

The rest of this paper is organized as follows: Section II presents a motivational example, Section III discusses the state of the art, and Section IV presents the system model. We define the mapping problem in Section V and afterwards we present our centralized and our distributed solution (Sections VII and VIII). Section IX details our implementation, and Section X describes our experiments and comparison to the state of the art.

II. MOTIVATION

This section discusses the importance of adapting task mappings in dynamic runtime scenarios. Let us consider a simple example of a software-pipelined computer vision application (object tracking) with 8 stages mapped to a system with 4 cores. Figure 2 (a) shows how the average runtime of each stage changes when adding multiple tracked objects to the input scene. Figure 2 (b) shows that an established (optimal) task mapping (Core 1: Stages 1-3, Core 2: Stages 4-5, Core 3: Stage 6, Core 4: Stage 7) would be more efficient in this case.

Experiment conducted on a P45C core running at 800 MHz.

1For this example, we use 4 cores of Intel’s SCC [9].
In the following, we discuss the system model we use for malleable software-pipelined applications. Each application \(k\) forms a pipeline \(P_k\) with \(N_k\) stages. Every stage \(S_j\) is characterized by \(c_j, e_j\) and \(o_j\) that denote the time consumed (in each iteration) for computation, for receiving the input data from its direct predecessor, and for transferring the output data to its direct successor. Figure 3 illustrates this model. For notational brevity, \(e_i = \alpha_i - 1/\forall i > 1\).

In order to decide about the mapping of applications, it is important to model their throughput for a given mapping. To achieve this, we require that each core belongs to at most one application (i.e. cores may not be shared among applications). Furthermore, we need to determine their maximum throughput, which is limited by their slowest stage. We consequently denote the maximal response time \(R_k\) for pipeline \(P_k\) as:

\[
R_k = \max_{1 \leq j \leq N_k} \{e_j + c_j + o_j\}. \tag{1}
\]

Therefore, the maximum throughput of pipeline \(P_k\) is defined as \(\frac{1}{R_k}\).

We introduce the malleability property to software pipelines by defining the basic operation fusion (and the inverse operation fission), in which multiple consecutive pipeline stages are combined, similar to fusing filters in StreamIt [6].

A fusion of stages creates a new stage which combines the computational requirements of the original stages but does not require communication between them, as shown in Figure 4. 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.

V. PROBLEM DEFINITION

We divide the problem of mapping malleable software pipelines into:

1) How to distribute the cores of a system among the applications (Section V-A) so that the overall system throughput is maximized.
2) How to assign the stages of an application to a given number of cores (Section V-B), thus providing their fusions.

A. Global Problem: Optimizing System Throughput

Given a set of \(K\) weighted (weights express priority levels) applications \(P = \{P_1, P_2, \ldots, P_K\}\) with weights \(W = \{w_1, w_2, \ldots, w_K\}\), each application \(P_i\) uses up to \(M_i\) cores and has a maximal response time \(R_i\). The objective is to maximize the overall weighted system throughput by finding an optimal distribution of (up to) \(M\) available cores to the individual applications.

Fig. 2: Changing computational requirements and resulting throughputs

3: Stages 6-7, Core 4; Stage 8) achieves a throughput of approx. 20.13 iterations/second. Due to these changes, the average throughput drops to 8.35 iterations/second. A possible solution to this problem is to adapt the task mapping (Core 1: Stage 1, Core 2: Stage 2, Core 3: Stages 3-4, Core 4: Stages 5-8), which achieves a throughput of 18.40 iterations/second. Consequently, adapting the established task mapping based on observations about the (possibly unpredictable) resource demands can significantly improve the throughput of a system. A more complex example of a system with 128 cores running 35 instances of real-world applications concurrently is discussed in Section X-C.

However, adapting such mappings at runtime is a challenging problem because task mapping is NP-complete. Thus, calculating mappings for larger systems at runtime may require an infeasibly high overhead or may require heuristics that lead to suboptimal solutions.

Fig. 3: Software pipeline model

Dynamic scheduling of stream-processing applications, which are a complex example of a system with 128 cores running 35 instances of asynchronous, iterative algorithms (AIAC) in grid computing systems.

Fig. 4: Fusion of pipeline stages

The related work can be grouped into mapping schemes for software pipelines and for parallel applications in general, assuming either distributed or shared memories.

Mapping schemes specific to software-pipelined systems have been recently proposed: [16] suggests calculating a set of optimal mappings at design-time. This works well for a specific set of scenarios, but it does not aim at capturing cases where application resource demands are unknown at design time. This is, however, the case when they depend on user interactions or on (unpredictable) properties of the input data. Dynamic scheduling of stream-processing applications, which are a superset of software pipelines, to embedded multi-cores with scratchpad memories is proposed by [11]. The property deemed to be unpredictable, and hence targeted by this approach, is merely the availability of cores, while application resource demands are assumed to be static. With similar assumptions, [4] incorporates user behavior in runtime task mappings, but aims at minimizing communication energy and requires that the number of tasks is less or equal to the number of cores.

Fig. 5: Changed computational requirements and resulting throughputs

Task mapping of general parallel applications assuming distributed memories: [2] presents a heuristic runtime load balancing scheme for asynchnronous, iterative algorithms (AIAC) in grid computing systems. Due to its focus, it does not take inter-task communication into account. It therefore may achieve inferior performance when tasks communicate heavily, as it is the case for many complex, real-world applications. In [10], a distributed heuristic for (re-)mapping of malleable applications using multiple agents is proposed. It relies on runtime observations and on offline profiles. However, their approach for achieving a scalable solution limits their decisions to local regions, which results in a lower throughput of the system. As [2] and [10] are most similar to our contribution, Section X compares them to our proposed schemes.

A statistical approach based on extreme value theory is presented in [14]. It generates a large random set of task assignments and has a runtime of 25 minutes to 2 hours, which we consider infeasible for dynamic scenarios that require to update mappings at runtime. In contrast to this, our restriction to (malleable) software pipelines allows to calculate optimal mappings in polynomial time, and near-optimal mappings in nearly constant time.

Task mapping for general parallel applications assuming shared memories: [12] and [15] propose runtime load balancing for symmetric multiprocessing systems. The authors of [17] propose to derive co-schedules based on offline profiles, with an extension to support different priority levels [18]. The focus of these schemes is on architectures with few cores and they require a shared address space. They are thus not directly applicable to many core systems with distributed memories.

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

III. RELATED WORK

The related work can be grouped into mapping schemes for software pipelines and for parallel applications in general, assuming either distributed or shared memories.

Mapping schemes specific to software-pipelined systems have been recently proposed: [16] suggests calculating a set of optimal mappings at design-time. This works well for a specific set of scenarios, but it does not aim at capturing cases where application resource demands are unknown at design time. This is, however, the case when they depend on user interactions or on (unpredictable) properties of the input data. Dynamic scheduling of stream-processing applications, which are a superset of software pipelines, to embedded multi-cores with scratchpad memories is proposed by [11]. The property deemed to be unpredictable, and hence targeted by this approach, is merely the availability of cores, while application resource demands are assumed to be static. With similar assumptions, [4] incorporates user behavior in runtime task mappings, but aims at minimizing communication energy and requires that the number of tasks is less or equal to the number of cores.

Task mapping of general parallel applications assuming distributed memories: [2] presents a heuristic runtime load balancing scheme for asynchronous, iterative algorithms (AIAC) in grid computing systems. Due to its focus, it does not take inter-task communication into account. It therefore may achieve inferior performance when tasks communicate heavily, as it is the case for many complex, real-world applications. In [10], a distributed heuristic for (re-)mapping of malleable applications using multiple agents is proposed. It relies on runtime observations and on offline profiles. However, their approach for achieving a scalable solution limits their decisions to local regions, which results in a lower throughput of the system. As [2] and [10] are most similar to our contribution, Section X compares them to our proposed schemes.

A statistical approach based on extreme value theory is presented in [14]. It generates a large random set of task assignments and has a runtime of 25 minutes to 2 hours, which we consider infeasible for dynamic scenarios that require to update mappings at runtime. In contrast to this, our restriction to (malleable) software pipelines allows to calculate optimal mappings in polynomial time, and near-optimal mappings in nearly constant time.

Task mapping for general parallel applications assuming shared memories: [12] and [15] propose runtime load balancing for symmetric multiprocessing systems. The authors of [17] propose to derive co-schedules based on offline profiles, with an extension to support different priority levels [18]. The focus of these schemes is on architectures with few cores and they require a shared address space. They are thus not directly applicable to many core systems with distributed memories.

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

IV. SYSTEM MODEL
Maximize \( \left\{ \sum_{k=1}^{K} \frac{w_k}{R_k^\ell} \right\} \) \quad \text{such that} \quad \sum_{k=1}^{K} M_k \leq M \quad \text{(2)}

We present a centralized, optimal scheme in Section VII and a highly scalable, distributed scheme in Section VIII.

To solve this problem, however, we need to solve the sub-problem of fusing pipeline stages first:

B. Sub-Problem: Fusion of Pipeline Stages

The throughput of an application is affected by how the stages are fused. Thus, we define a sub-problem that minimizes the maximal response time of each pipeline \( P_k \) (with \( N_k \) stages) by fusing stages for an optimal throughput when utilizing at most \( M_k \) cores.

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

VI. FUSION OF PIPELINE STAGES

In order to find an optimal solution to the problem of Section V-B, 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 for adapting mappings at runtime. We therefore propose an algorithm based on dynamic programming that derives optimal solutions for minimizing the maximal response time by using \( M_k \) cores to execute pipeline \( P_k \).

Let \( P_{k,j} \) be a sub-pipeline 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.

That is, let \( R_k(j,m) \) be the minimum maximal response time for executing \( P_{k,j} \) on \( m \) cores. Moreover, we build table \( F_k(\ell,j) \) for all \( \ell,j \) such that \( 1 \leq \ell \leq j \leq N_k \) in which

\[
F_k(\ell,j) = c_\ell + o_j + \sum_{k=\ell}^{j} c_k.
\]

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

\[
R_k(j,0) = \infty \quad \forall j = 1, \ldots, N_k
\]

\[
R_k(j,1) = F_k(1,j) \quad \forall j = 1, \ldots, N_k
\]

Furthermore, we define function \( \text{minmaxRF}_k(j,m) \) as:

\[
\text{minmaxRF}_k(j,m) = \min_{m-1 \leq c < \ell} \left\{ \max \{ R_k(\ell,m-1), F_k(\ell+1,j) \} \right\}.
\]

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

\[
R_k(j,m) = \begin{cases} 
R_k(j,m-1) & j < m \\
\min \{ R_k(j,m-1), \text{minmaxRF}_k(j,m) \} & j \geq m
\end{cases}
\]

The dynamic programming starts by computing the resulting maximal response times utilizing only one core for the first \( j = 1 \ldots N_k \) stages. Then, the programming computes the maximal response times for the first \( j = 1 \ldots N_k \) stages, on up to two cores. Since the programming already stored the resulting maximal response times of using only one core for the first \( j \) stages, it can easily choose whether to use one or two cores (in one of the possible fusion combinations) for the same \( j \) stages. The process is repeated once again for three cores, knowing in advance if it is optimal to use one or two cores for the first \( j \) stages, so it only needs to compare the previous result with any new possible fusion for the same \( j \) stages but now utilizing up to three cores. Thus, iteratively, an optimal solution is achieved because all combinations of stages and cores are considered, but the complexity is reduced since optimal solutions are stored in tables and do not need to be recomputed.

The space/time complexity is \( O(N_k^3) \) 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 pseudo-code for this dynamic programming.

**Algorithm 1 Maximal Response Time Minimization**

**Input:** The times \( c, o \) and \( a \) for the \( N_k \) stages of pipeline \( P_k \), and the maximum \( M_k \) cores available;

**Output:** The minimal maximal response time using at most \( M_k \) cores;

1: Initialize \( F_k(\ell,j) \) according to Eq. (3), \( \forall (\ell,j) \) such that \( 1 \leq \ell \leq j \leq N_k \);
2: for \( m = 0 \) to \( M_k \) do
3: \quad for \( j = 1 \) to \( N_k \) do
4: \quad \quad if \( m \leq 1 \) then
5: \quad \quad \quad Build \( R_k(j,m) \) according to Eq. (4);
6: \quad \quad else
7: \quad \quad \quad Build \( R_k(j,m) \) according to Eq. (6);
8: \quad \quad end if
9: \quad end for
10: end for
11: return \( R_k(N_k,M_k) \);

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 size \( O(M_k N_k) \). When building the \( TR_k(j,m) \) table, each cell holds the \( j^* \) value of the sub-solution 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 \( R_k(j,m-1) \) turned 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 \( \ell \) from Equation 5 that made this improvement possible and therefore \( TR_k(j,m) \neq j \).

The fusions that give an optimal maximal response time can be derived from table \( TR_k(N_k,M_k) \) as follows: starting from cell \( (j,m) = (N_k,1) \), 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 sub-solution 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 mapped 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. Section S2 discusses a detailed example.

VII. CENTRALIZED SCHEME

With the dynamic programming of Section VI, we can decide how to maximize the overall weighted system throughput in a centralized manner. Suppose that \( R_k(N_k,m) \) for \( m = 1,2,\ldots,\min\{N_k,M_k\} \) has been built. For notational brevity, let \( N_k < M_k \) and 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. \( k > m \), the function \( G(k,m) \) is defined to \( -\infty \). Then, we know that the initial (boundary) condition for \( G(1,m) \) is:

\[
G(1,m) = \frac{w_1}{R_1(N_1,m)} \quad \forall m = 1,2,\ldots,M
\]

The recursive function for \( G(k,m) \) with \( k \geq 2 \) is expressed in Equation (8). The time complexity, provided that \( R_k(N_k,m) \) is known, is \( O(M_k^2) \). 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 and thus serves as its *speed-up vector*. Algorithm 2 shows a pseudo-code for this dynamic programming.

\[
G(k,m) = \begin{cases} 
-\infty & k > m \\
\max_{k-1 \leq m' < m} \left\{ G(k-1,m') + \frac{w_k}{R_k(N_k,m'-m')} \right\} & k \leq m
\end{cases}
\]

An additional tracking table \( TG(K,M) \) of size \( O(M/M) \) allows for easily deriving how many cores should be assigned to each pipeline. When building the \( TG(k,m) \) table, each cell holds the \( m^* \) value of the sub-solution that makes the programming optimal. For the initial condition \( k = 1 \), \( TG(k,m) \) is set to zero. When \( k > m \), then
There are clusters in level 1 (\(\ell = 1\)) containing the information of the weights \(w_k\) and tables \(R_k(N_k, m)\) for \(m = 1, 2, \ldots, M\); similarly, the clusters \(G_i^1\) (level 2) contain the information of table \(G(K^*, M)\) of its child clusters \(C^1_i\) (level 1). This applies likewise to all upper levels. In this way, each level distributes cores among its children based solely on this (limited) information. Consequently, the computational requirement is distributed hierarchically among the system. By limiting the frequency of propagating the aforementioned tables, our distributed scheme largely reduces the computational overhead, but is unable to achieve optimal mappings if the tables change more frequently than they are propagated.

We denote \(G^i_k(k, m)\) as the table for the modified version of the dynamic programming in Section VII. Considering that \(w^*_1, R^*_1, N^*_1\) are the parameters of the first child (pipeline) of node \(C^1_i\), node \(C^1_{i-1}\) is the first child of node \(C^1_i\), value \(K^*_{i-1}\) is the number of children of node \(C^1_i\), and value \(K^*_{i-2}\) is the number of children of node \(C^1_{i-1}\) and node \(G^i_k(1, m)\), then the initial conditions of \(G^i_k(1, m)\) are:

\[
G^i_k(1, m) = \frac{w^*_i}{R^*_i(N^*_i, m)} \quad \forall m = 1, 2, \ldots, M \quad \text{when } \ell = 1
\]

\[
G^i_k(1, m) = G^{i-1,k}_{i-1}(K^*_{i-2}, m) \quad \forall m = 1, 2, \ldots, M \quad \text{when } \ell \geq 2,
\]

The recursive function when \(\ell = 1\) and \(k \leq m\) is:

\[
G^i_k(k, m) = \max_{k-1 \leq m' \leq m} \left\{ G^i_{k-1}(k-1, m') + \frac{w^*_i}{R^*_k(N_k, m-m')} \right\}, \quad (10)
\]

and finally, the result is found in cell \(G^i_k(K^*_{i-1}, m)\).

IX. SYSTEM DETAILS

In the following, we discuss the components of our schemes and their implementation details. We have implemented both schemes on Intel’s Single-Chip Cloud Computer (SCC) and in a high-level system simulator detailed in Section X-A. Both schemes employ several components written in C++ that communicate by exchanging network messages:

A. Components

The **centralized scheme** employs application heads and a centralized controller. Each application denotes one of its cores to its children pipelines, independently upon the other clusters of the same level. Similarly, \(G^{i-1,k}_{i-2}\) is the number of cores to its children pipelines, independently upon the other clusters of the same level.
B. Implementation of our Schemes

Figure 6 gives an overview of our implementation, which is divided into a compile-time and a run-time part. At compile-time, initial \( R_k \) tables are derived from profiling. Partitioning the application into a software pipeline defines its finest degree of parallelism. At runtime, our implementation is split into the application- and the OS-layer.

The application layer contains the components of Section IX-A and a checkpointing-based implementation of task migration (see below). Each core that is assigned to one application executes the same executable file, while its parameters control which of its stages are executed. A pseudo-code of this main procedure is shown in Section S1.

The OS layer provides the communication infrastructure (to allow stages to communicate via an MPI-like interface, orthogonal to their physical location and fusion) and supports task migrations.

On the SCC, our components are implemented as daemons for Intel’s 3.1.4 ubuntu-based linux. The components of both schemes communicate via sockets and separate program- and control communication by two logical channels. To measure the communication overhead of our schemes, we log the communication volume in the control channel. For obtaining the computational overhead of the distributed scheme, we average core distribution/fusion calculations in each cluster/application head over 1,000,000 times by comparing CPU ticks before and after.

C. Implementation of Task Migration

Task migration is carried out on application level through checkpointing after each iteration, with a lightweight support by the middleware (to start executables as needed). When the controlling scheme (both our centralized and distributed schemes) chooses to change the fusions or re-distributes cores among applications, the respective stages are notified by the middleware: When the corresponding stage reaches a checkpoint, it saves its state and requests the middleware at the destination core to start its executable file if the destination core formerly belonged to a different application. It then sends its state to the newly started executable, which then continues the execution of the (fused) stage procedure. The corresponding overheads of these operations are evaluated and discussed in Section X-E.

X. EXPERIMENTAL RESULTS

A. System Setup

Our experiments have been conducted on Intel’s Single-Chip Cloud Computer (SCC) [9] and using a high-level many core simulator. The SCC is a platform that integrates 48 x86 cores in 24 tiles (two cores each) on a single chip. The individual PS4C cores (45mm process) run at 800 MHz, are connected via a 2 GHz network-on-chip with a bisection bandwidth of 2TB/s. Each core has 16 KB of instruction- and 16 KB of data cache, and 256 KB 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 is written in C++, executes task traces collected on the SCC, and simulates the network-on-chip interconnect. The simulator delivers accurate information on the application

<table>
<thead>
<tr>
<th>Name</th>
<th>Stages</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>automotive</td>
<td>21</td>
<td>see Section X-II</td>
</tr>
<tr>
<td>h264ref</td>
<td>4</td>
<td>SPEC CPU 2006 [8]</td>
</tr>
<tr>
<td>lame</td>
<td>4</td>
<td>MiBench [7]</td>
</tr>
<tr>
<td>PGP</td>
<td>5</td>
<td>MiBench [7]</td>
</tr>
<tr>
<td>sifiunx3</td>
<td>22</td>
<td>SPEC CPU 2006 [8]</td>
</tr>
</tbody>
</table>

TABLE I: Benchmark applications

/ system throughputs and the communication volumes / overheads (algorithm runtimes have been collected on the SCC). It runs on a six-core AMD OpteronTM8431 CPU (2.4 GHz) with 64 GB DDR3 RAM. The SCC allows measuring the computational overhead accurately, but as 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 to dispose of the cores physically.

Measurements conducted on the SCC:

- Computational overhead for up to 1024 cores.
- Throughput of the centralized scheme for up to 48 cores.
- Fusion/fission overheads.

Experiments conducted using our simulator:

- Communication overhead.
- Throughput of our centralized and of our distributed schemes.

For the experiments, we spawn the benchmark applications listed in Table I 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).

B. Benchmark Scenario

Table I shows an overview of the benchmark applications and their number of stages. The applications have been manually parallelized to form malleable software pipelines. We chose this set of applications because they are most suitable to form software pipelines. The implementation details of how we adapted the state-of-the-art schemes of [2], [10] to compare them against our schemes can be found in Section S4.

The automotive application is a vision-based application that takes its algorithms from the IVT library [1]. 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 (648x480 pixels at 30fps). The other applications have been taken from the respective benchmark suites.


\begin{tabular}{|c|c|c|c|c|c|}
\hline
Application & Min & Max & Avg & \sigma & Old Core & New Core \\
\hline
automotive & 1 & 32 & 19 & 15.21 & 0.63 & 22 \\
h264ref & 13 & 53 & 27 & 22.73 & 1.07 & 76 \\
lame & 9 & 10 & 9 & 1.32 & 0.18 & 19 \\
PGP & 1 & 27 & 12 & 9.11 & 0.30 & 66 \\
SPHINX3 & 12 & 22 & 17 & 4.21 & 0.51 & 44 \\
\hline
\end{tabular}

\textbf{TABLE II:} Overheads of fusion/fission operations

\section*{C. Achieved System Throughput}

In the following, we compare the throughput achieved by the distributed scheme with the centralized scheme (thus against optimal mapping) and with two state-of-the-art runtime remapping schemes, DistRM [10] and AIAC [2]. Figure 7 shows the average system throughput over 50 runs when running 7 instances of each benchmark application (35 applications, or 392 stages in total) on 128 cores, connected by a NoC mesh as featured by the SCC. To show how each scheme gradually improves the mapping of stages to cores, we initially start all stages on a single core and let the corresponding schemes improve the system throughput incrementally. After this is achieved, we randomly stop 25% of the applications at \( t = 10 \) seconds. While the centralized (thus optimal) scheme achieves an increased throughput of roughly 13.7%, the average system throughput drops for the other schemes from roughly ca. 17-29 iterations/second to ca. 9-17 iterations/second because without adapting the mapping, the cores that formerly executed the stopped applications are now idle. The schemes then improve the throughput by adapting the mapping of stages to cores. Our distributed scheme achieves a system throughput of 94.78% compared to the optimal (centralized) scheme. On average, the distributed scheme increases the throughput by 11.3% and 60.6% over [10] and [2], respectively.

\section*{D. Computational and Communication Overheads}

Figure 8 shows how the computational overhead of our centralized scheme grows with a growing number of cores and applications. Up to a considerable problem size (e.g., 64 cores, 64 applications), optimal mappings can be calculated in less than 0.5s, which may be sufficiently fast for certain systems. However, this overhead is significantly larger for larger systems as the runtime grows quickly beyond 35 seconds.

The distributed scheme has a constant time complexity as each cluster head on level 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 1024 cores).

Figure 9 compares the total communication overhead of our centralized and of our distributed schemes. This overhead includes status updates and notifications, the updates of the \( c_i, c_o, \) and \( \sigma_i \) values, as well as the propagation of all tables and speed-up vectors. As this overhead merely reaches around 365.3 KiB/s (0.025% of the total communication for a system with 1024 cores, 275 applications) for our centralized scheme and roughly 138 KiB/s (0.009%) for our distributed scheme, we consider it as negligible.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig9}
\caption{Comparison of communication overheads}
\end{figure}

\section*{E. Fusion/Fission (i.e. Task Migration) Overhead}

Table II summarizes the overhead for fusions/fissions of two stages of each application (collected on Intel’s Single-Chip Cloud Computer). When the fusion/fission operation 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 very small. Otherwise, the executable file of the application needs to be started by the middleware, which takes considerably more time. However, our experiments show that this is only the case in less than 5% of the conducted fusion/fission operations. Hence, the overhead of our proposed schemes is small and thus, we find that our centralized scheme is well suitable for managing smaller many core systems, while our distributed schemes is well suitable for systems with hundreds of cores.

\section*{XI. CONCLUSION REMARKS}

In this paper, we show how a high system throughput can be achieved and maintained even in large many core systems despite unpredictable, significant variances in the demand for both computational as well as for communication resources. This is achieved by optimizing the configurations (fusion of stages) and the distribution of cores among the applications at runtime. Additionally, to proposing an optimal scheme, we show how optimality can be sacrificed to maintain near-optimal throughputs even for large systems with hundreds of cores.

\section*{XII. 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).

\section*{REFERENCES}

SUPPLEMENTAL MATERIAL

S1. PIPELINE PROCEDURES

Each application corresponds to a single executable file. Two parameters, FirstStage and Fusions, controls which of its stages are executed. Both parameters are supplied from the command line when the application is started on a core, but may be changed by an application head during runtime to change the fusions/fissions of stages.

Example 1 Pseudo-code of a Pipeline Procedure

This is the main (entry point) procedure for an application which is executed on each core that is assigned to the application. StageFunction is the individual function that executes the functionality of stage $i$. 

Input:

FirstStage
Fusions

1: while true do 
2: if FirstStage > 1 then 
3: data = ReceiveData( FirstStage − 1 ); 
4: end if 
5: for $i$ = FirstStage to Fusions do 
6: if Task Migration Triggered (send state) then 
7: SendState( Destination ) 
8: continue 
9: end if 
10: if Task Migration Received (receive state) then 
11: ReceiveState( ) 
12: continue 
13: end if 
14: data = StageFunction( data ) 
15: end for 
16: if FirstStage + Fusions < N_k then 
17: SendData( FirstStage + Fusions + 1, data ); 
18: end if 
19: end while

The main pipeline procedure forms the entry point which is started on each core that is assigned to an application. Example 1 illustrates this: For each iteration, an iteration starts with receiving the required input data (if it is not the first stage) (lines 2-4). Then, all stages that are fused on the current core are executed sequentially (lines 5-15). Finally, the output data is sent to the succeeding stage, if applicable (lines 16-18).

We carry out our application-layer task migration in this loop: When one or more stages are migrated to a different core, they save and transmit their state. The middleware supports this when our schemes decide to assign a new core to an application (i.e. this core was not already assigned to this application) by starting the

S2. EXAMPLE OF FUSING PIPELINE STAGES

Given the pipeline $k$ shown in Figure S-1 with $N_k = 4$ stages and having available up to $M_k = 4$ cores to assign to it, we first proceed to build table $F_k(i, j)$ according to Equation (3), as stated in Algorithm 1:

$F_k(1, 1) = 60 \quad F_k(2, 2) = 110 \quad F_k(3, 3) = 110$

$F_k(1, 2) = 150 \quad F_k(2, 3) = 60 \quad F_k(3, 4) = 140$

$F_k(1, 3) = 100 \quad F_k(2, 4) = 90 \quad F_k(4, 4) = 70$

The next step is to compute the initial conditions for the table $R_k(j, m)$ according to Equation (4), which in other words means to compute the maximal computational requirements for a sub-pipeline with $j$ stages using up to $m$ cores. Since this are initial conditions, the tracking table $TR_k(j, m)$ for $m = 0$, has no previous value for $j^*$, and is therefore filled with zeros.

From now on, since $m ≥ 2$, the table $R_k(j, m)$ is build according to Equation (6). In this particular example, when $m = 2$ the solution for every sub-pipeline chooses to use the result from $R_k(1, 1)$ and to fuse the rest of the stages together in one core, thus, the tracking table $TR_k(j, 2)$ will be filled with $j^* = 1$ for any $j$.

The result shown in Table S-I. The optimal solution can be derived from table $TR_k(j, m)$: Starting from cell $(j, m) = (4, 4)$ and traverse the table in the direction $(j^*, m − 1)$, one can derive that the optimal solution fuses stages $S_2$ and $S_4$, and leave stages $S_1$ and $S_3$ as they are.

<table>
<thead>
<tr>
<th>$j$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>60</td>
<td>150</td>
<td>100</td>
<td>130</td>
</tr>
<tr>
<td>1</td>
<td>60</td>
<td>110</td>
<td>60</td>
<td>90</td>
</tr>
<tr>
<td>2</td>
<td>60</td>
<td>110</td>
<td>60</td>
<td>70</td>
</tr>
<tr>
<td>3</td>
<td>60</td>
<td>110</td>
<td>60</td>
<td>70</td>
</tr>
<tr>
<td>4</td>
<td>60</td>
<td>110</td>
<td>60</td>
<td>70</td>
</tr>
</tbody>
</table>

TABLE S-I: Example tables from Algorithm 1

S3. EXAMPLE OF CONSTRUCTING TABLES

Example: Given the pipelines $R_1$, $R_2$ and $R_3$ shown in Table S-II, with weights $w_1 = w_2 = w_3 = 10000$ and having up to $M = 6$ available in the system, in order to find the maximal overall system throughput we first proceed to compute the initial conditions for the table $G(k, m)$ according to Equation (7). Since this are initial conditions, the tracking table $TG(k, m)$ for $k = 1$ has no previous value for $m^*$, and is therefore filled with zeros.

From now on, since $k ≥ 2$, the table $G(k, m)$ is build according to Equation (8).

The fully completed results are shown in Table S-III. Looking at table $TG(k, m)$, starting from cell $(k, m) = (3, 6)$ and traversing the table in the direction $(k − 1, m^*)$, one can derive that the optimal solution will assign one core to pipeline $R_1$, one core to pipeline $R_2$ and four cores to pipeline $R_3$.

<table>
<thead>
<tr>
<th>$k$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>76.92</td>
<td>−∞</td>
<td>−∞</td>
<td>−∞</td>
</tr>
<tr>
<td>2</td>
<td>111.11</td>
<td>160.26</td>
<td>−∞</td>
<td>−∞</td>
</tr>
<tr>
<td>3</td>
<td>142.86</td>
<td>194.44</td>
<td>193.59</td>
<td>−∞</td>
</tr>
<tr>
<td>4</td>
<td>142.86</td>
<td>226.19</td>
<td>227.78</td>
<td>−∞</td>
</tr>
<tr>
<td>5</td>
<td>142.86</td>
<td>233.76</td>
<td>285.26</td>
<td>−∞</td>
</tr>
<tr>
<td>6</td>
<td>142.86</td>
<td>242.86</td>
<td>410.25</td>
<td>−∞</td>
</tr>
</tbody>
</table>

TABLE S-II: Example tables of different pipelines


This section details how we adapted the state-of-the-art schemes of [2] and [10] in order to achieve a fair comparison to our proposed schemes.

$G(3, 6)$: Overall Performance

<table>
<thead>
<tr>
<th>$k$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

TABLE S-III: Example tables from algorithm 2
A. AIAC [2]

AIAC exchanges workload between physically neighboring cores to balance the computational load evenly. To adapt this scheme for 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 demands among all stages on each core exceeds \( c_i \). To achieve a fair comparison, we relax the assumption that only consecutive stages may be mapped to the same core. For our implementation of AIAC, a core may execute any stage from any application.

B. DistRM [10]

DistRM [10] distributes cores among applications, but relies on the applications to themselves decide how to distribute their tasks accordingly. Therefore, we use our optimal fusion algorithm from Section VI to achieve a fair comparison. Consequently, only the number of cores assigned to each application differs between DistRM and our schemes. Fusions of pipeline stages are carried out identically. We also adapt DistRM by using the speed-up vectors according to Section VI. As DistRM remains in local optima if the speed-up of an application does not increase with another core (even if this was the case for a larger number of additional cores), we report marginal improvements (we choose an \( \epsilon = 5 \times 10^{-4} \)) as long as the number of cores does not exceed the number of stages of the corresponding application. Using the described adaptions, we can achieve a fair comparison with DistRM.