Enforcing Deadlines for Skeleton-based Parallel Programming

Enforcing Deadlines for Skeleton-based Parallel Programming

Paul Metzger*, Murray Cole*, Christian Fensch*, Marco Aldinucci†, Enrico Bini†
*School of Informatics, University of Edinburgh, Edinburgh EH8 9AB, UK
†Department of Computer Science, University of Torino, 10149 Torino, Italy
paul.metzger@ed.ac.uk, m.cole@inf.ed.ac.uk, c.fensch@ed.ac.uk, marco.aldinucci@unito.it, enrico.bini@unito.it

***Abstract*—High throughput applications with real-time guarantees are increasingly relevant. For these applications, parallelism must be exposed to meet deadlines. Directed Acyclic Graphs (DAGs) are a popular and very general application model that can capture any possible interaction among threads. However, we argue that by constraining the application structure to a set of composable "skeletons", at the price of losing some generality w.r.t. DAGs, the following advantages are gained: (i) a finer model of the application enables tighter analysis, (ii) specialised scheduling policies are applicable, (iii) programming is simplified, (iv) specialised implementation techniques can be exploited transparently, and (v) the program can be automatically tuned to minimise resource usage while still meeting its hard deadlines.**

As a first step towards a set of real-time skeletons we conduct a case study with the job farm skeleton and the hard real-time XMOS xCore-200 microcontroller. We present an analytical framework for job farms that reduces the number of required cores by scheduling jobs in batches, while ensuring that deadlines are still met. Our experimental results demonstrate that batching reduces the minimum sustainable period by up to 22%, leading to a reduced number of required cores. The framework chooses the best parameters in 83% of cases and never selects parameters that cause deadline misses. Finally, we show that the overheads introduced by the skeleton abstraction layer are negligible.

I. Introduction

High throughput applications with timing constraints, such as autonomous driving [1], and network applications [2] [3], drive the development of parallel real-time systems [1], [2], [3]. To program these, programmers have to resort to low-level programming language models such as message passing and threads [4], [5], [6], [7], [8], These are considered error prone, non-portable and inefficient in terms of programmer productivity [9], [10], [11], [12]. As previous work points out, new high-level programming language models for real-time systems are therefore needed [13]. Directed Acyclic Graphs (DAGs) are a very general application model that can capture any possible interaction among threads [14]. In contrast, we propose to constrain the application structure to a set of composable skeletons to improve programmability, resource usage and timing analysis at the price of often expendable generality w.r.t. DAGs. Individual skeletons may cover seperate portions of an application. Therefore, we envision a framework of different real-time skeletons that can be composed to implement full applications as is the case for mainstream parallel systems [15], [16], [17], [18].

As a first step towards such a framework we conduct a case study with the *job farm* skeleton [4] that is applicable to a wide range of applications [17], [19], [20]. Job farms lend themselves to programs that process streams of input data such as signal processing, graphics and networking applications [21], [22], [23], [24]. Thies et al. report 51 applications that use farms with a median of eight farm instances per application [25]. These applications include a "Ground Moving Target Indicator", "2D Inverse Discrete Cosin Transform", and "Fast Fourier Transform".

Structural information encoded in skeletons can be used to automatically tune applications. As an example of this, we introduce *job batching* for real-time systems. Batching reduces the overheads that come with parallelism and so decreases the required core counts. Alternatively, it allows the use of less powerful and so cheaper hardware, or adding additional workload without increasing resources. We show that batching is viable in the context of real-time systems and that it can be implemented transparently to developers with the farm skeleton. To further ease programming, we devise an analytical framework for the computation of farm internal parameters, which would have to be carefully chosen by hand otherwise. Using this framework, we implement Peso [5], a deterministic and self-tuning *farm* skeleton library for the hard real-time XMOS xCore-200 microcontroller.

Interference in the memory system poses a challenge for predictability on conventional multi-core systems [26], [27]. In contrast, our evaluation platform by XMOS, its compiler tool chain, and WCET predictor have been co-designed to remove this issue [28] (see Sec. VII-B). Consequently, our model and techniques also benefit from this property.

[1]https://www.mobileye.com/our-technology/evolution-eyeq-chip/

[2]https://www.kalrayinc.com/download/wp_kalray-nvme-of-target-controller-solutions/

[3]https://www.cisco.com/c/en/us/products/collateral/routers/asr-1000-series-aggregation-services-routers/solution_overview_c22-448936.html

[4]We remark that this skeleton is called "task farm" in the parallel computing community. However, to not overload the term "task" with conflicting interpretations, we use the term *job farm*, which, we believe, better represents our intended interpretation in real-time systems.

[5]https://github.com/paulmetzger/Peso

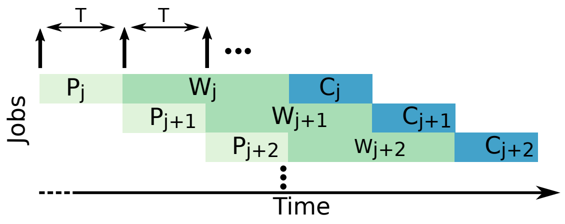


Fig. 1: Illustration of computations that can be implemented with job farms. Such computations have three generic phases: a producer (P), a worker (W), and a consumer phase (C).

We show experimentally that the computed parameter choices are the same or are very close to the best parameter choices that we determine through brute-force searches, and never cause deadline misses. Batching reduces the minimum task period that can be sustained by a given application and core count by 22.38%. Therefore, it can improve the throughput by the same percentage or reduce the core count. Finally, we show that the overheads introduced by Peso over hand-coded solutions are negligible.

The remainder of this paper is structured as follows: Sec. II introduces skeletons and the job farm skeleton. Sec. III motivates and describes job batching. Sec. IV describes our system model. Sec. V presents our analytical framework for farm parameters. Sec. VI introduces our Peso library. Sections VII and VIII present the experimental setup and results. Finally, Sections IX and X discuss related work and conclude.

II. Background

A. Skeletons

Algorithmic Skeletons are high-level programming constructs for typical parallel computations [15]. They implement generic code concerned with parallelism. Developers pass sequential application specific code to them that is then executed in parallel. A range of skeletons have been proposed for other fields. Well known examples are: map, and reduce [16] [29]. We focus on the job farm skeleton, as it is widely applicable to real-time systems (see Sec. I).

Benefits of skeletons are more efficient implementations and increased programmer productivity. Structural information encoded in skeletons can be used for implementations with lower execution time, lower energy consumption or higher throughput [30], [31], [32]. Skeletons improve programmer productivity by off-loading the error prone task of writing and tuning low-level parallel code to library or compiler developers [9]. Skeleton libraries have a small overhead (e.g. due to extra function calls).

B. Job Farms

Farms are composed of a set of workers that run in parallel and apply a function to a stream of inputs. Inputs are generated by a producer and the workers' results are sent to a consumer. Fig. 1 illustrates applications that lend themselves to farms with a set of jobs, released every period T. Each job j is composed of a producer phase P_j that generates input data,

```
CONFIGURE_FARM(farm, //Farm name
  producer_func, worker_func, consumer_func)

void main() {
  start_farm();
}
```

Fig. 2: Illustration of a task farm API. The functions ending with `_func` are provided by application developers. Example implementations of these functions are shown in Fig. 7.

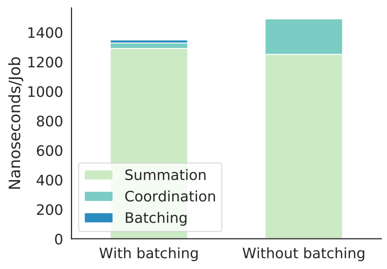


Fig. 3: Break down of the per job execution time in a farm worker with and without batching for a simple example application that sums up 30 integers per job (less is better). 10 jobs are aggregated to a batch on the left-hand side.

a worker phase W_j that processes the data, and a consumer phase C_j that consumes the results.

Here the phases P_j can be mapped to the same cores as their execution does not overlap. The same is true for the C_j phases. However, W_j and W_{j+1} cannot be mapped to the same core as their execution overlaps.

Fig. 2 illustrates a task farm API. The functions that are passed to `CONFIGURE_FARM` implement P_j, W_j, and C_j of Fig. 1. Synchronisation and farm internal communication between producer, workers, and consumer as well as the parallel execution of workers is implemented by a skeleton library or compiler. The sole task of application developers is to provide the sequential worker, producer, and consumer functions. Internal communication is implemented with the so-called *dispatcher* and *aggregator*. They are hidden from application developers behind the farm API. The dispatcher schedules jobs on workers and the aggregator informs the consumer when new worker generated results are available (see Sec. VI-B for their implementation in our Peso library).

Non real-time farms have demand-driven implementations like work stealing which hinder WCET calculations. In contrast, our farm implementation is timing predictable.

III. The Case for Job Batching & Self-Adaptation

This section motivates batching with a simple example, and argues for a self-adaptive implementation to further ease programming.

A. Reduced Core Count via Job Batching

Passing a job through a job farm incurs bookkeeping overheads as some of the execution time is spent by coordination

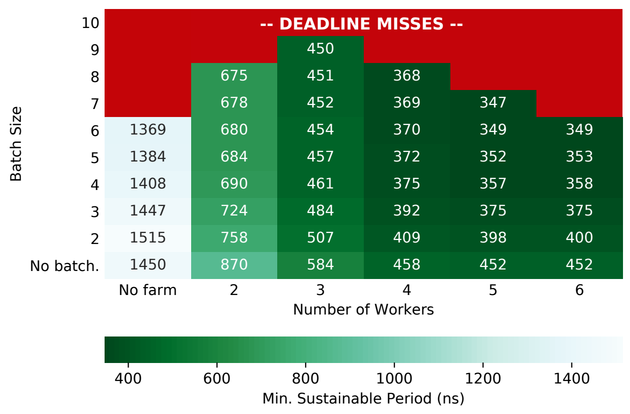


Fig. 4: Measured minimum supported task periods with increasing batch size and worker count on our evaluation platform (see Section VII-A). Each job computes the sum of 30 integers and the relative deadlines are $15\mu s$.

between producer, worker, and consumer threads. In the case of our evaluation platform and library, this means passing a pointer to the in- and output data of a job from the producer to a worker, and from a worker to the consumer. Note, the costs of this do not depend on the in- and output size of a job. These coordination costs can be substantial for tasks with short running rapidly arriving jobs. We propose *job batching* to reduce these overheads.

In a simple farm (with no batching), jobs are executed immediately when their input data and the necessary computing resources are available. Job batching exploits the slack time to the deadline to reduce communication overheads. With job batching, jobs are halted and aggregated to be then dispatched and processed in batches. This way communication costs are spent only per batch and not per job. Reduced overheads in turn allow tasks to run on cheaper hardware with less cores.

As a preliminary investigation, we break down the execution time that each job of a simple example task spends in a worker with and without batching. Fig. 3 shows this break-down for a sample application. Job batching reduces the communication costs by $10\times$ here and introduces a small overhead that comes from additional instructions that implement batching. The number of instructions executed during each summation is slightly higher when batching is used as the instructions generated by the compiler are slightly different.

Multiple instances of this simple computation need to be run in parallel if new input data arrives rapidly and a single core cannot meet the target period (see next section). In this example job batching allows for 12% lower task periods than without batching. Section VIII-B presents a quantitative study of the benefits of batching.

B. *Improved Ease of Programming Through Self-Adaptation*

The number of jobs in batches and worker counts have to be carefully chosen to avoid deadline misses. Choosing these parameters is non-trivial for developers as the batch size and worker count parameter space is difficult to navigate. For example, Fig. 4 shows the minimum supported periods for all possible parameter combinations with the same simple task used in Fig. 3 on our evaluation platform. As can be seen if this task has a period of 400ns then four workers and batch sizes larger than two are best. Given a task and hardware platform it is not obvious what the best parameter choice is if data like the one in this heatmap is not available. Without our analytical framework, this data can only be attained through a time consuming and so often impractical exhaustive search.

IV. System Model

The relation between the number of cores, batch size, and characteristics of the farm workers is established in this section. The presented system model allows the implementation of a farm skeleton to automatically pick the minimal number of cores required to meet the application's period and deadline.

A. *Jobs, Job Releases, and Deadlines*

The workload to be executed by the job farm is modelled by a periodic *task* that releases a sequence of jobs. The k-th job is released at time

$$r_k = (k-1)\,T, \qquad k = 1, 2, \ldots$$

with T being the *period* of job releases. When a job is released, it processes its input data and generates the corresponding output data. All jobs have a *deadline* D relative to the release instant. This means that the k-th job cannot finish later than

$$d_k = r_k + D = (k-1)\,T + D.$$

We do not set any constraint on the deadline (neither implicit nor constrained deadline model). Hence, we assume to have an *arbitrary* deadline.

The *response time* R_k is the time taken by the k-th job to complete starting from its release at r_k. Hence, no job misses any deadline if

$$\forall k = 1, 2, \ldots, \quad r_k + R_k \le d_k$$

which is equivalent to

$$\forall k = 1, 2, \ldots \quad R_k \le D. \tag{1}$$

The computation of the response time R_k depends on several scheduling decisions and is investigated in Sec. V.

B. *Cores and Batch Size*

To minimise the communication costs incurred through parallelisation, jobs are grouped in *batches* of size b. An entire batch of b jobs is then executed on the same core. The number of available cores that process batches is denoted by m. Usually the term "worker" is used to denote a thread that executes the worker function of the jobs that are assigned to it, while a "core" is a physical piece of hardware capable of executing instructions. However, from a scheduling point of view the distinction between these two notions vanishes since a static 1-to-1 assignment from workers to cores is used. Hence, we use the terms *worker cores* and *workers* interchangeably.

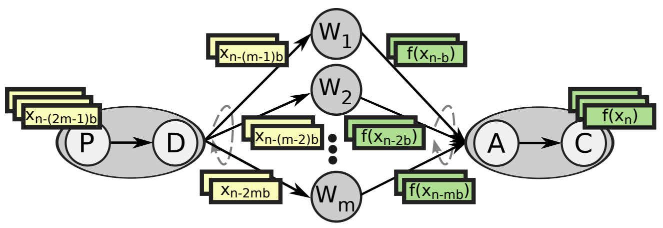


Fig. 5: Illustration of a farm implementation. The producer (P) generates jobs that are scheduled in batches of size b over m workers (W) by a dispatcher (D). An aggregator (A) receives results and sends them to the consumer (C). Grey ellipses indicate that producer and dispatcher, and aggregator and consumer share a core each. Workers execute on their private cores. Light and dark green boxes are in- and output data. Drawn through and perforated arrows indicate communication channels and round robin scheduling.

For the analysis we assume that code executing on one core cannot influence the WCET of code on another. We discuss when this holds for our evaluation platform and how we allow for situations in which it does not hold in Sec. VII-C.

C. Execution Time

Introducing job batching requires a deeper understanding of the job execution time, which goes beyond a single worst-case execution time (WCET). For this reason, we split the job execution time into time intervals, which map to the execution phases of Fig. 5. Most terms stand for time spent in a worker and are denoted with $C_{\mathsf{W}\dots}$.

- C_{D} is the execution time spent in the dispatcher (see Sections II-B and VI-B).
- C_{com} is the latency of farm internal communication that is required for the coordination between dispatcher, workers, and aggregator. Therefore, this is application independent. C_{com} does not include execution time that is required to tear down a communication channel on the sender side and set up a channel on the receiver side because these instructions do not contribute to the latency. More specifically, this is the communication delay between dispatcher and workers, and workers and aggregator (see Sections II-B and VI-B). On our evaluation system this is the delay for communication via a crossbar.
- C_{Wc} is the execution time spent by a worker in farm internal communication. Unlike C_{com}, it includes the code necessary to set up and tear down the communication channels. This corresponds to the *communication* time in Fig. 3.
- C_{Wsetup} is the execution time of code that sets up the execution of a batch and so is executed once per batch. Hence, in the special case when $b = 1$ (no job batching) this term is $C_{\mathsf{Wsetup}} = 0$.
- C_{WonceJ} is the execution time of code that implements batching and so is executed once per job. For the same reason as above, if $b = 1$ then $C_{\mathsf{WonceJ}} = 0$. The sum of C_{Wsetup} and C_{WonceJ} corresponds to the time spent in *batching* in Fig. 3.
- C_{Wuser} is the execution time for the user provided worker function. This function is executed once per job. This corresponds to the time spent in the *summation* in Fig. 3.
- C_{A} is execution time that is spent in the aggregator (see Sections II-B and VI-B), once per batch.
- C_{C} is the execution time to unbatch the results of a job. This takes place before results are used by the consumer and is necessary to hide batching from application developers. Since unbatching happens sequentially

$$C_{\mathsf{C}} \leq T \tag{2}$$

must hold. Otherwise, the unbatching phase is overloaded. As above for C_{Wsetup}, if $b = 1$ then $C_{\mathsf{C}} = 0$.

The concrete execution times that we used for our experiments are listed in Table I. These execution times may be summed up depending on whether they are executed once per job or once per batch. To highlight these two portions of time, we define the following quantities:

- $C_{\mathsf{WonceB}} = C_{\mathsf{Wc}} + C_{\mathsf{Wsetup}}$ subsumes execution time that is spent once per batch.
- $C_{\mathsf{WfullJ}} = C_{\mathsf{WonceJ}} + C_{\mathsf{Wuser}}$ subsumes execution time of code that is executed once per job.

Finally, we also set

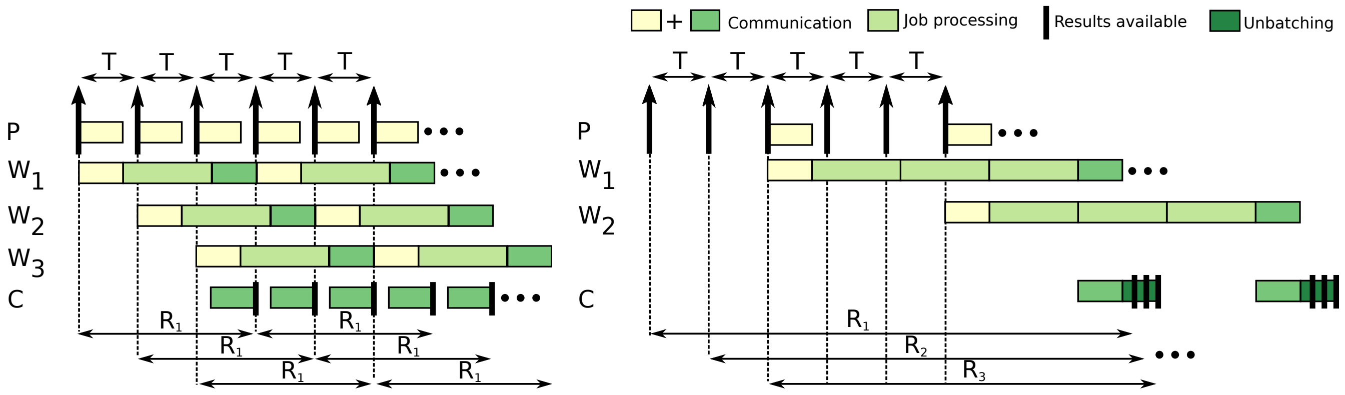
- $C_{\mathsf{WfullB}} = C_{\mathsf{WonceB}} + C_{\mathsf{WfullJ}} b$, which is the execution time required to process an entire batch.
- $C_{\mathsf{O}} = C_{\mathsf{A}} + 2C_{\mathsf{com}} + C_{\mathsf{D}}$, which are the overheads of code that implements the parallel execution. C_{com} is multiplied by two to account for the communication between dispatcher and a worker, and a worker and the aggregator. This is the only term that subsumes execution time spent outside the workers.

From a farm's perspective jobs arrive when the input data associated with a job is ready to be processed. Therefore, we do not introduce a term for any external input data preparation.

Two example schedules of the same sequence of jobs with and without batching are illustrated in Fig. 6. As shown in Fig. 6b, thanks to the savings of communication cost, one worker core less is required if three consecutive jobs are grouped in a batch, at the price of an increase of response time. If a task's deadline D allows for larger response times as in Fig. 6b then batching reduces the number of cores required for a task. Communication blocks correspond to C_{com} (see above). The next section is dedicated to the formalisation of this qualitative argument.

V. Our Analytical Framework: Analysis of Batch Scheduling

This section presents an analytical framework that allows farm skeleton implementations such as Peso to automatically choose the number of worker cores and the batch size. Firstly, we compute the minimum worker core count m as a function of the batch size b and demonstrate that the required worker core count m decreases with the batch size b. As the intuition



(a) Execution without batching. (b) Execution with batching and a batch size of three.

Fig. 6: Illustration that shows that batching reduces the required worker core count at the expense of longer job response times R_k. (a) and (b) illustrate the execution of a job farm with three and two workers (W_i), a producer (P), an aggregator (A), a consumer (C), and with the same period in both subfigures. Time flows from left to right. (a) does not use batching, but (b) does with a batch size of three. The farm in (b) needs less worker cores for the same task due to batching. The response times are higher in (b) than in (a) because jobs have to wait for other jobs in the same batch. Some of the arrows in (b) that indicate the arrival of a new job are not followed by communication because batches are only dispatched after enough jobs have been accumulated. The index k of R_k in (b) indicates the relative position of a job in its batch.

suggests, we show that the job response times increase linearly with the batch size b. Hence, the job deadline sets a natural upper limit on the batch size b. The maximum feasible batch size b_{max} is then the value that:

- maximises throughput if the core count is given, or
- minimises resource usage if the job period T is given.

A. Worker Core Count vs. Task Period

We establish here the relationship between the task period T, which determines the required throughput, and the worker core count m, which determines the available throughput. Clearly, a shorter task period T needs more worker cores m and vice versa.

The first step is to define the minimum sustainable period.

Definition 1. Given m worker cores and a batch size of b, we define the *minimum sustainable period* $T_{\mathsf{min}}(b,m)$ of a task as the minimum period which does not cause overload and so deadline misses.

Such a term is well defined, since for an arbitrarily small period T the job farm will be overloaded at some point, while it will never be overloaded for an arbitrarily large T.

As a first step, we compute the minimum period $T_{\mathsf{min}}(b,1)$ that can be sustained if only a *single* worker is used. A batch of b jobs is ready to be processed every $b \times T$ time units. The time required by a worker to process such a batch is C_{WfullB}. Hence, with only one worker no overload occurs as long as

$$b \times T \geq C_{\mathsf{WfullB}} \qquad b = 2, 3, \ldots \tag{3}$$

which means that

$$\begin{aligned} T_{\mathsf{min}}(b,1) &= \frac{C_{\mathsf{WfullB}}}{b}, \qquad b = 2, 3, \ldots \\ &= \frac{C_{\mathsf{WonceB}} + C_{\mathsf{WfullJ}} b}{b} \\ &= \frac{C_{\mathsf{Wc}} + C_{\mathsf{Wsetup}} + (C_{\mathsf{Wuser}} + C_{\mathsf{WonceJ}}) b}{b}. \end{aligned} \tag{4}$$

Note that Equation (3) is only concerned with hypothetical job farms with a single worker and so does not stand in contradiction with Fig. 6, which uses multiple workers.

Following the same arguments that we used to derive (4), if m worker cores are available (instead of only 1) a lower period can be sustained. Therefore, we construct

$$T_{\mathsf{min}}(b,m) = \frac{C_{\mathsf{WonceB}} + C_{\mathsf{WfullJ}} b}{bm}, \qquad b = 2, 3, \ldots, \tag{5}$$

by multiplying the denominator of (4) by m.

$T_{\mathsf{min}}(1,m)$, which is the minimum sustainable period without batching ($b = 1$) and with m worker cores cannot be determined by setting $b = 1$ in (5). $T_{\mathsf{min}}(b,m)$ in (5) accounts for all batching overheads which are clearly not present if jobs are not batched. Hence, we have

$$T_{\mathsf{min}}(1,m) = \frac{C_{\mathsf{Wc}} + C_{\mathsf{Wuser}}}{m} \tag{6}$$

that is based on the same reasoning as (5) except that C_{Wsetup} and C_{WonceJ} are set to 0 and b is set to 1 because jobs are not processed in batches.

To show that batching can reduce the minimum sustainable period we compute the factor by which batching improves the minimum sustainable period over an implementation without batching. More specifically, we compute the ratio between $T_{\mathsf{min}}(b,m)$ of (5) and $T_{\mathsf{min}}(1,m)$ of (6)

$$\begin{aligned}
&\frac{T_{\mathsf{min}}(b,m)}{T_{\mathsf{min}}(1,m)} = \frac{\frac{C_{\mathsf{WonceB}}}{b} + C_{\mathsf{WonceJ}} + C_{\mathsf{Wuser}}}{C_{\mathsf{Wc}} + C_{\mathsf{Wuser}}}\\
\implies &\lim_{b\to\infty} \frac{T_{\mathsf{min}}(b,m)}{T_{\mathsf{min}}(1,m)} = \frac{C_{\mathsf{WonceJ}} + C_{\mathsf{Wuser}}}{C_{\mathsf{Wc}} + C_{\mathsf{Wuser}}} \qquad (7)\\
\implies &\lim_{b\to\infty} \frac{T_{\mathsf{min}}(b,m)}{T_{\mathsf{min}}(1,m)} \begin{cases} < 1, \text{ if } C_{\mathsf{Wc}} > C_{\mathsf{WonceJ}} \\ \geq 1, \text{ if } C_{\mathsf{Wc}} \leq C_{\mathsf{WonceJ}}. \end{cases}
\end{aligned}$$

Equation (7) shows two things. Firstly, batching improves the minimum sustainable period because $\frac{C_{\mathsf{WonceB}}}{b}$ approaches zero if b approaches infinity. Secondly, we can assert that batching reduces the minimum sustainable period $T_{\mathsf{min}}(b,m)$, if this factor is lower than 1 i.e. if the time required for batching C_{WonceJ} is lower than the time required for communication C_{Wc}.

Finally, we address a different although related problem: given an application period T, what is the minimum number of worker cores m that can match the demanded workload? If the following inequality holds strictly

$$T \geq T_{\mathsf{min}}(b,m) \qquad (8)$$

the slack between T and $T_{\mathsf{min}}(b,m)$ can be used to reduce the number of cores. From (5) and (8) it follows that

$$T \geq \frac{C_{\mathsf{WonceB}} + C_{\mathsf{WfullJ}} b}{mb}$$

and

$$m \geq \frac{C_{\mathsf{WonceB}} + C_{\mathsf{WfullJ}} b}{Tb}.$$

Since m must be an integer it must be

$$m \geq m_{\mathsf{min}}(b,T) = \left\lceil \frac{C_{\mathsf{WonceB}} + C_{\mathsf{WfullJ}} b}{Tb} \right\rceil = \left\lceil \frac{C_{\mathsf{WonceB}}}{Tb} + \frac{C_{\mathsf{WfullJ}}}{T} \right\rceil. \quad (9)$$

As can be seen in (9), the minimum number of required worker cores $m_{\mathsf{min}}(b,T)$ decreases with the batch size b. However, the batch size b cannot be chosen arbitrarily high to minimise the number of worker cores as it has a natural limit due to the task deadline D, as shown next.

B. *Job Batch Size vs. Task Deadline*

This section shows that the batch size cannot be chosen arbitrarily high. This is the case because, as the size of batches grows, the time needed for a job to pass through a job farm grows as well, which eventually leads to deadline violations.

Fig. 6 illustrates how batching affects the response time. As expected, the response time with batching (of Fig. 6b) is higher than the response time without batching (of Fig. 6a). Other factors that can increase response times are the same in both figures. As illustrated in Fig. 6b the response time increases with the batch size for multiple reasons:

- jobs are not immediately dispatched to workers but are halted until the batches to which they belong are full,
- jobs have to wait until the other jobs in the same batch are processed by a worker,
- jobs have to wait until other jobs are unbatched.

We start by computing the response time $R_k(b)$ of the k-th job, assuming a batch size b, which is

$$\underset{k=1,\ldots,b}{R_k(b)} = \underbrace{(b-k)T}_{\text{Batch aggregation}} + \underbrace{b\,C_{\mathsf{WfullJ}}}_{\text{Batch processing}} + \underbrace{(k-1)C_{\mathsf{c}} + C_{\mathsf{c}}}_{\text{Unbatching}} + C_{\mathsf{o}}. \quad (10)$$

Job response times are composed of several components that are discussed in detail below:

- $(b-k)T$ is the time spent to batch b jobs. The first job in a batch (with $k=1$) experiences the longest delay $(b-1)T$, while the last one (with $k=b$) completes a batch and experience no aggregation delay.
- $b\,C_{\mathsf{WfullJ}}$ is the batch processing time. As discussed in depth earlier, the completion of jobs is not communicated to the dispatcher until all b jobs in the same batch are processed. Therefore, all jobs in a batch experience a delay of $b\,C_{\mathsf{WfullJ}}$ which is the time required to process an entire batch.
- The time to unbatch the result of the k-th job is due to (i) the waiting time for the earlier jobs in the same batch $C_{\mathsf{c}}(k-1)$ to be unbatched, plus (ii) the time to unbatch the k-th job itself which is C_{c}. Hence, altogether the time needed to unbatch the result of the k-th job is $k\,C_{\mathsf{c}}$, which is the sum of two terms: time spent by the k-th job waiting for earlier jobs to be unbatched, and the time needed to unbatch the k-th job.
- The term C_{o} represents time spent in farm internal communication, dispatcher, and aggregator (see Sec. IV-C).

For clarity, some of the terms used in (10) are not explicitly shown in Fig. 6b. The *batch aggregation time* in (10) is the time between the arrival of a job and the subsequent communication between the producer and worker in Fig. 6b. The *batch processing* time corresponds to the light green "Job processing" boxes, and the time required for *unbatching* is directly shown in the figure. The term C_{o} corresponds to the yellow and green communication blocks.

Next, we determine the impact of the deadline constraint of (1) on the batch size b. Since the *task response time* R is

$$R = \max_k\{R_k\} = \max_{k=1,\ldots,b}\{R_k\} \qquad (11)$$

then the deadline constraint of (1) trivially becomes $R \leq D$.

From (10), the job response time R_k can be written as:

$$\begin{aligned}
\underset{k=1,..,b}{R_k(b)} &= (b-k)T + b\,C_{\mathsf{WfullJ}} + kC_{\mathsf{c}} + C_{\mathsf{o}}\\
&= bT + b\,C_{\mathsf{WfullJ}} + C_{\mathsf{o}} - k(T - C_{\mathsf{c}}). \qquad (12)
\end{aligned}$$

From the constraint of (2), it follows that $T - C_{\mathsf{c}} \geq 0$. Unsurprisingly, (12) is maximal for $k=1$ since the first job in a batch has the longest response time. By setting $k=1$ in (12), we find the response time R of the task that is

$$R = \max_{k=1,\ldots,b}\{R_k\} = (b-1)T + b\,C_{\mathsf{WfullJ}} + C_{\mathsf{o}} + C_{\mathsf{c}}. \qquad (13)$$

Finally, from the deadline constraint of

$$R \leq D$$

we can find the constraint on the batch size b, that is

$$\begin{aligned} R &= (b-1)T + b\,C_{\mathsf{WfullJ}} + C_{\mathsf{O}} + C_{\mathsf{C}} \leq D \\ &= b(T + C_{\mathsf{WfullJ}}) - T + C_{\mathsf{O}} + C_{\mathsf{C}} \leq D \end{aligned}$$

which allows us to find the maximum batch size $b_{\mathsf{max}}(T, D)$

$$b_{\mathsf{max}}(T,D) = \left\lfloor \frac{D + T - C_{\mathsf{O}} - C_{\mathsf{C}}}{T + C_{\mathsf{WfullJ}}} \right\rfloor. \tag{14}$$

We observe that the upper bound for $b_{\mathsf{max}}(T, D)$ is

$$b_{\mathsf{max}}(T,D) \leq \left\lfloor \frac{D}{T} \right\rfloor + 1$$

which states the natural fact that the number of jobs in a batch cannot exceed the maximum number of pending jobs.

Based on Equation (14) we determine when a task benefits from batching. Naturally, batching is beneficial if and only if

$$b_{\mathsf{max}}(T,D) \geq 2.$$

This means that batching is only applicable if it is possible to aggregate two or more tasks. This equation is equivalent to

$$\frac{D + T - C_{\mathsf{O}} - C_{\mathsf{C}}}{T + C_{\mathsf{WonceJ}} + C_{\mathsf{Wuser}}} \geq 2.$$

$$C_{\mathsf{Wuser}} \leq \frac{D - T - C_{\mathsf{O}} - C_{\mathsf{C}} - 2C_{\mathsf{WonceJ}}}{2}. \tag{15}$$

Clearly, a necessary condition for batching is $D > T$ since jobs cannot be aggregated otherwise.

For example, by replacing the constants in Equation (15) with values for our evaluation platform (see Table I) we get

$$C_{\mathsf{Wuser}} \leq \frac{D - T - 980\,\mathsf{ns}}{2}.$$

In this case a task with period $T = 1$ us and deadline $D = 5$ us benefits from batching as long as $C_{\mathsf{Wuser}} \leq 1.51$us.

Assuming that the maximum batch size b_{max} is used (there is no reason to do otherwise), we can find the minimum core count $m_{\mathsf{min}}(b_{\mathsf{max}}(T, D), T)$. In fact, as apparent from (9), it is always best to use the largest possible batch size. We find the minimum core count by setting $b = b_{\mathsf{max}}(T, D)$ in (9):

$$m_{\mathsf{min}}(b_{\mathsf{max}}(T,D),T) = \left\lceil \frac{C_{\mathsf{WonceB}}}{T \left\lfloor \frac{D+T-C_{\mathsf{O}}-C_{\mathsf{C}}}{T+C_{\mathsf{WfullJ}}} \right\rfloor} + \frac{C_{\mathsf{WfullJ}}}{T} \right\rceil. \tag{16}$$

The values of the terms of the right-hand sides of (14) and (16) are either task properties or can be determined with WCET analysis. These equations can thus be used to compute the minimum number of worker cores and the maximum batch size at compile time as demonstrated by our library.

VI. The Peso Library

We present a macro-based farm library for real-time systems that is statically scheduled to guarantee predictable WCETs. Peso uses the equations presented in Sec. V. In this section, we illustrate Peso's farm API and discuss its implementation.

```
 1 typedef struct per_job_data {
 2   int per_job_input_vector[PER_JOB_INPUT_SIZE];
 3   int result;
 4 } per_job_data_t;
 5 PREPARE_FARM(per_job_data_t, PERIOD_NS)
 6
 7 PRODUCER(producer_func, //Producer name
 8   while (1) {
 9       /*Wait for input data to be ready*/
10       submit_data();
11   })
12
13 WORKER_FUNCTION(worker_func, //Worker function name
14   data_t,
15   unsigned int result = 0;
16   for (int i = 0; i < PER_JOB_INPUT_SIZE; ++i) {
17       result += access_data()->
18                   per_job_input_vector[i];
19   }
20   access_data()->result = result;)
21
22 CONSUMER(consumer_func, //Consumer name
23   data_t,
24   printf("Result:%d", receive_data()->result);)
```

Fig. 7: Illustration of Peso's API. Each job computes the sum of `PER_JOB_INPUT_SIZE` integers. Multiple summations are executed in parallel but each one is performed sequentially.

A. API Concepts

Fig. 7 shows the implementation of an example application with Peso. The C macros `PRODUCER`, `CONSUMER`, and `WORKER_FUNCTION` are provided by the library. The first parameters are unique names that are required to instantiate the farm (see below) and the second is the implementation.

Peso provides functions for data accesses that hide the internal storage scheme. The producer uses `submit_data()` in line 10 to send input data to the job farm. Either the producer prepares the input data in `per_job_input_vector` or an external process that then signals the producer when the data is ready. The worker function accesses input data and stores results via `access_data()` in lines 17 and 20. Finally, the consumer receives results via `receive_data()`. Lines 1 to 4 specify the in- and output data of each job. Note that an instance of `per_job_data` is accessed by only a single worker but multiple instances are processed in parallel. Finally, line 5 in Fig. 7, and the code in Fig. 2 instantiate the farm.

B. Implementation & Internal Communication Overheads

Fig. 5 provides an overview over Peso's architecture. The farm is composed of two logical entities that are hidden from developers by the farm API: the dispatcher, and the aggregator. The dispatcher deals out batches to workers and the aggregator collects the corresponding results. Both serve workers in a round robin fashion to achieve predictability, for instance, the aggregator does not collect the n-th result of worker $i + 1$ before it has collected the n-th result of worker i.

Peso stores worker in- and output data consecutively in a buffer. The dispatcher sends a pointer into this buffer to a worker when it deals out a job or a batch if batching is used. Only the pointer to the inputs of the first job of a batch are sent. Data locations for other jobs in the same batch are computed based on this pointer. Time spent in sending these pointers makes up the internal communication overheads that we reduce through batching in this implementation.

VII. Experimental Setup

A. Evaluation Platform & Methodology

We use the XMOS xCore-200 microcontroller which is designed for hard real-time systems and has two clusters of eight cores (see below) [33]. We use the XCC compiler version 14.4.4 with the default `-O2` optimisation flag. No OS is on the device and no thread scheduler is required in the context of our experiments as all threads execute on dedicated cores.

We perform brute-force parameter space searches to determine optimal parameter choices for comparison with those computed by our analytical framework (see Sec. VIII-A1 and VIII-A2). These are based on experiments and measurements with timers provided by the target platform.

We take five samples per data point. This is true for the brute-force parameter searches as well. Each sample yields the same result except in a small number of cases in Sec. VIII-C where small variations in intercore communication cost (of up to 1ns) occur. Note that our WCET costs allow for this. Because of this invariability we do not show error margins.

B. Predictability & The Memory System

To achieve predictability the xCore-200 uses a barrel processor design. This design issues instructions of eight logical cores to a shared five stage pipeline in a round robin fashion [33]. This means a core is serviced every five cycles if five or less cores are used or every six, seven, or eight cycles if more are active. Consequently, the device must be programmed with multiple threads to achieve best performance. The memory system is designed so that all requests are fulfilled in five cycles. Therefore, at a device clock frequency of 500Mhz the WCET of memory accesses is: 10ns, 12ns, 14ns or 16ns depending on how many cores are active. The best-case execution time is always 10ns.

The device does not have data caches and translation lookaside buffers which can cause interference in conventional hardware [28], [33]. Each core has private registers, an instruction buffer, and access to shared SRAM. Memory accesses have exclusive access to the shared SRAM as all cores share the memory pipeline stage. The time to serve a request is not affected by reordering, or bank and row conflicts.

Cores can directly communicate with each other through a crossbar. We use this crossbar for farm internal communication (see C_{com} in Sec. IV-C) which is reduced through batching.

C. Worst Case Execution Times

XMOS provides a static code analysis tool for WCETs [6] that is similar to the OTAWA Eclipse plugin [34]. The tool provides WCETs on the level of code blocks, and single instructions.

[6]https://www.xmos.com/developer/published/xmos-timing-analyzer-manual

TABLE I: WCETs in cycles of the job farm components (see Sec. IV-C) for the sample applications (see Sec. VII-D) and the used input sizes. Each cycle is 2ns. Presented WCETs are determined with less than six cores (see Sec. VII-C). Only C_{Wuser} changes with the sample application and the input size.

App.	**DMV**		**RED**		**SMV**	
Input size	**5**	**10**	**15**	**30**	**10**	**15**
C_{D}	75	75	75	75	75	75
C_{com}	65	65	65	65	65	65
C_{Wc}	125	125	125	125	125	125
C_{Wsetup}	5	5	5	5	5	5
C_{WonceJ}	40	40	40	40	40	40
C_{Wuser}	*630*	*4030*	*415*	*790*	*1455*	*3435*
C_{A}	115	115	115	115	115	115
C_{C}	90	90	90	90	90	90

The WCET of a code section can be computed by summing up the WCETs of the relevant instructions.

We determine the WCETs on our evaluation hardware through static code analysis and measurements. The WCET of intercore communication C_{com} is determined through measurements under stress. To put maximum stress on the core-interconnect we execute dummy threads on all cores that use up all available interconnect communication channels and do nothing but constantly send data back and forth. The maximum of 10000 measurements is then used for C_{com}. Note, the WCET and best case execution time of intercore communication differs. All other WCETs are determined through static analysis.

The compiler generated instructions of the worker function and the farm internal worker code can slightly vary with the batch size. With rare instruction sequences the hardware cannot refill the instruction cache transparently and has to stall the pipeline for a single cycle to fetch instructions [28]. This can cause the statically determined execution times to vary with the batch size. Based on the maximum number of such pipeline stalls that we could observe for a set of instructions we manually increase the generated WCETs that we used for our evaluation to allow for these stalls. The so adjusted WCETs are presented in Table I. These stalls cause the slight increases in Fig. 4 when six instead of five workers are used.

Per instruction WCETs increase with each additional core if more than five cores are used (see Sec.VII-B). Peso considers this when it chooses the worker count and handles this further complexity for application developers.

D. Sample Applications

The sample applications are: dense matrix vector multiplication (DMV), sparse matrix vector multiplication (SMV), and reduction (RED). They are widely used across various domains such as computer vision and machine learning [35]. SMV uses the compressed sparse row format. RED computes the sum of a set number of integers. Each worker executes the computational kernel of an application (i.e. the reduction or matrix multiplication) sequentially. However, multiple kernel instances are executed in parallel. We use two input vector sizes with each application. Table I shows the relevant WCETs.

VIII. Evaluation

This section experimentally validates the analytical framework presented in Sec. V, and shows that Peso's overheads over hand-crafted code are small. As mentioned in Sec. VII-B, to achieve predictability our evaluation platform uses an atypical core design as opposed to the cores in conventional systems (see Sec. VII-B). For clarity, we refer to these as cores.

A. Experimental Validation of our Analytical Framework

1) Worker Core Counts: We validate the worker count choices of our framework (that is $m_{\mathsf{min}}(b_{\mathsf{max}}, T)$) as computed from (16) against, the best possible worker counts that we determine with an exhaustive search (see Sec. VII-A). For this we decrease the number of workers until deadline misses occur.

Fig. 8a shows that our framework chooses the best worker count in all cases except two and crucially, never makes choices that cause deadline misses. Possible explanations for these two cases are the pessimism added to WCETs by potential pipeline stalls, and intercore communication that can be faster than its WCET as discussed in Sec. VII. The pessimism added by stalls is highest in these two cases.

Required worker core counts decrease with higher periods, and increase with larger input sizes. Increasing periods mean less pressure on the farm and so allow for less worker cores. Larger input sizes cause higher batch processing times (see Sec. V) and so require more workers to match the task period.

2) Batch Sizes: We validate the batch sizes computed by our framework by comparing them with the best ones, which we determine through a brute-force search (see Sec. VII-A). For this we hard code the worker count to the maximum and increase the batch sizes until we measure deadline misses.

Fig. 8b shows that our framework chooses the best or close to best batch sizes and never ones that cause deadline misses. Again, differences between the two batch sizes can be explained with the pessimism inherent to the WCETs of the user code and intercore communication (see Sec. VIII-A1).

The batch sizes decrease with increasing periods (see DMV) and with larger job input sizes (see RED). Larger periods and input sizes cause longer job aggregation and batch processing times (see Sec. V) respectively and so only allow for smaller batch sizes with the same relative deadlines.

B. Fewer Cores with Batching & The Effect of Input Sizes

To quantify the impact of batching, Fig. 9 shows by how much batching reduces the min. sustainable periods $T_{\mathsf{min}}(1, m)$ for a given number m of cores over implementations without batching (see Equation (6)). A lower min. period means that tasks that previously needed additional cores because their periods are too low for a given core count can now be scheduled. We use a variant of Peso without batching for the baseline measurements. To experimentally determine the min. periods of both versions, we decrease the periods until the farm is overloaded and jobs miss their deadlines. The number of worker cores is hard coded to the maximum for this.

Batching lowers the min. periods $T_{\mathsf{min}}(b, m)$ (and so enables higher throughput) by up to 45.36%, 16.6% on average, and never degrades the min. period. However, the baselines of RED with input size 15 and DMV with input size 10 are affected by the pipeline stall issue discussed in Sec. VII-C. Therefore, in the interest of a fair comparison, omitting these gives a maximum and average lowering of the min. period of 22.38% and 12.54% respectively.

The improvements of the min. period decrease with increasing input sizes. For example, RED benefits more with an input size of 15 elements than with 30 elements. This is expected since larger input sizes mean higher application code WCETs. The execution time share spent in communication decreases with higher application code WCETs and so maximum achievable improvements through batching decrease.

C. Abstraction Layer Overheads

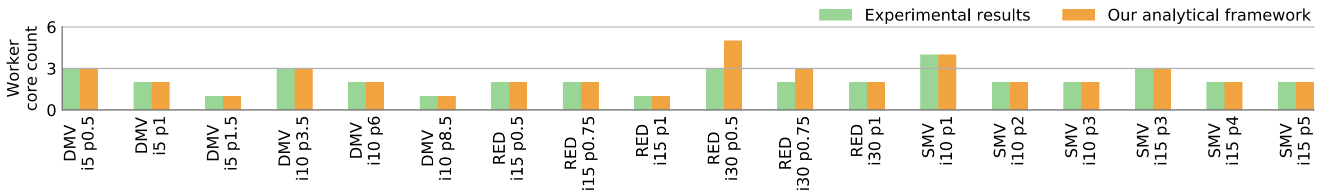
This section evaluates Peso's overheads in terms of the minimum sustainable period. Fig. 10 compares the min. periods of Peso based implementations with the ones of carefully hand-crafted code that does not use Peso's abstractions and thus parallelism and batching have to be implemented in the application code. To measure the min. period, we decreased the periods until the implementations are overloaded and deadline misses occur. The number of workers is set to the maximum for this.

The minimum, average, and maximum difference between the min. period of the hand and Peso implementations are 8ns, 18.6ns, and 24ns. The overheads introduced by Peso's abstraction layer over hand implementations translates to an on average 3.37% higher min. period. The maximum overhead is 8.66% and the minimum 1.06%. The min. period of the hand implementation of the reduction application with an input size of 30 is slightly higher than the period of the Peso based implementation due to the stall issue discussed in Sec. VII-C.

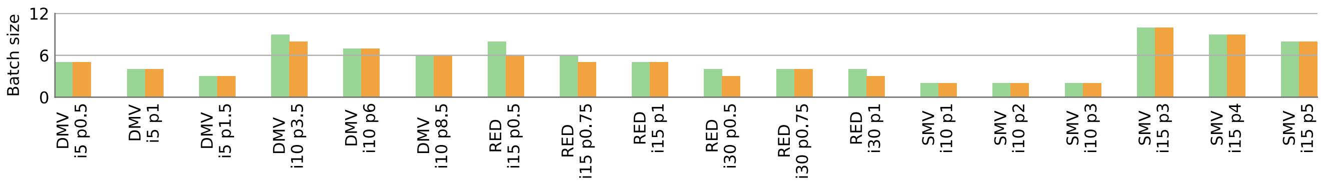
The overheads come from hiding job batching from application developers through Peso's abstraction layer. To make batching application developer transparent the consumer calls `access_data()` for each job even though all results of a batch of jobs are completed at the same time. Removing this abstraction and its associated costs lowers the min. period but exposes application developers to more complexity.

IX. Related Work

Previous works map jobs to multicore processors and group them to reduce communication costs [36], [37], [38], [39]. However, most existing parallel programming models for real-time systems focus on very general application models (such as dataflow). In contrast, we constrain the application structure to given skeletons. At the price of losing some generality w.r.t. DAGs, the advantages are: tighter analysis, skeleton informed scheduling policies, simplified programming, and programmer transparent efficient implementation techniques.



(a) Experimentally determined worker core counts and worker core counts chosen by our analytical framework.



(b) Experimentally determined batch sizes and batch sizes computed by our analytical framework.

Fig. 8: Job farm parameter comparisons. Sample applications: Reduction (*RED*), dense and sparse matrix-vector multiplication (*DMV* and *SMV*). Periods in μs are prefixed with a p and input sizes are prefixed with an i.

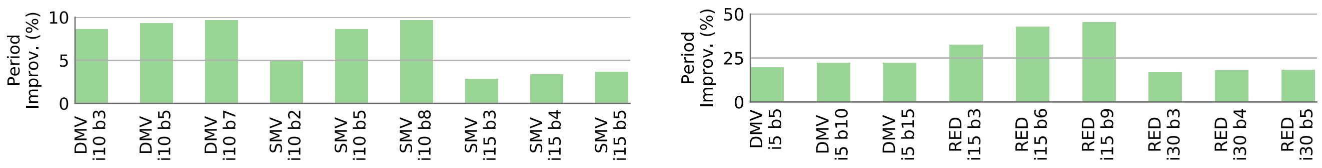


Fig. 9: Reduction in the minimum sustainable periods with batching over Peso without batching (higher is better). Sample applications: Reduction (*RED*), dense and sparse matrix-vector multiplication (*DMV and SMV*). Numbers after the sample application name indicate the number of input elements per job and batch sizes prefixed by i and b.

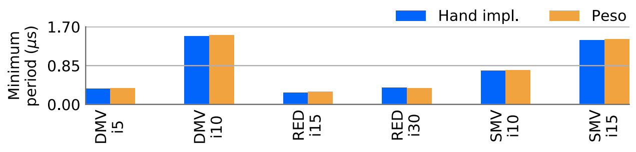


Fig. 10: Minimum sustainable period with and without Peso (lower is better). Sample applications: Reduction (*RED*), dense and sparse matrix-vector multiplication (*DMV and SMV*). Numbers after the sample application names indicate the number of input elements per job prefixed by i.

Stegmeier and Ungerer et al. are the first to use skeletons in the context of parallel real-time systems and built a library that offers a farm skeleton [20], [40]. We make three contributions that go beyond this initial work. Firstly, we exploit inter-job parallelism whereas the authors exploit parallelism within jobs. Secondly, we are the first to demonstrate that structural information encoded in skeletons can be used for performance improvements in the context of real-time systems as demonstrated before in other fields [16], [30], [31], [32]. Thirdly, we are the first to show that skeletons can be used for fine grained analytical application models. Stegmeier and Ungerer et al. provide a tool that recommends core counts for skeletons based on approximated WCETs. However, they do not consider synchronisation operations for their analysis. We improve on this and carefully analyse the execution of application and library code including the synchronisation of threads. Lastly, the authors evaluate their library on a simulator, and we use off the shelf hardware.

The P-SOCRATES project investigates the task based parallel programming model in the context of real-time systems (*task* has a different meaning here than in the context of real-time systems) [41], [42]. This model is on a lower abstraction level than skeletons. It encodes less structural information that allows for resource efficient implementations such as job batching. It puts a bigger burden on application developers as less parallelism related implementation details are hidden. Lastly, they do not investigate self-adaptation.

We review parallelism related constructs that are offered by the commonly used real-time APIs of POSIX, Java, and ADA [4], [5], [6]. These APIs require programmers to work and set periods and deadlines on the level of threads. Synchronisation and communication are implemented with mutexes, monitors, queues, remote procedure calls and/or critical sections. Except for queues, programmers have to implement mutual exclusion manually. Failing to do so correctly leads to subtle bugs that are hard to repeat and fix. Queues are on a higher abstraction level but are often used to send pointers to data structures. Programmers have then to implement mutually exclusive access across concurrently executing threads again themselves. These constructs are on a lower abstraction level than algorithmic skeletons. None of the currently used low-level constructs allow compilers or run-time systems to automatically set the degree of parallelism or hide and tune resource efficient implementations such as batching that are informed by the structure of a task.

X. Conclusion & Future Work

We argue for skeletons as an alternative to DAGs to program parallel hard real-time systems as they ease programming by abstracting implementation details. Structural information encoded in skeletons also allows for tight analysis and efficient scheduling.

We conduct a case study with the farm skeleton. We present an analytical framework that combines knowledge about this skeleton with predictable hardware to automatically choose the minimum core count. Based on this we develop an efficient execution strategy that reduces parallelism related overheads.

We demonstrate experimentally that in most cases our framework chooses the best or close to best parameters, and never makes choices that cause deadline misses. Our skeleton informed execution strategy improves minimum sustainable periods by up to 22.38% and so reduces required core counts. Lastly, compared to carefully hand-crafted code, the overheads of our farm are negligible. In the future we plan to investigate further skeletons for real-time systems as well as the combination and nesting of them.

Acknowledgments

We would like thank XMOS and Paul Neil from XMOS for donating the development board. This work started during a HiPEAC funded research visit of the first author in Marco Aldinucci's group. This work was also supported by the Engineering and Physical Sciences Research Council (grant EP/L01503X/1), EPSRC Centre for Doctoral Training in Pervasive Parallelism at the University of Edinburgh, School of Informatics.

References

[1] T. Ungerer, C. Bradatsch, M. Gerdes, F. Kluge, R. Jahr, J. Mische, J. Fernandes, P. G. Zaykov, Z. Petrov, B. Böddeker, S. Kehr, H. Regler, A. Hugl, C. Rochange, H. Ozaktas, H. Cassé, A. Bonenfant, P. Sainrat, I. Broster, N. Lay, D. George, E. Quiñones, M. Panic, J. Abella, F. Cazorla, S. Uhrig, M. Rohde, and A. Pyka, "parMERASA — multi-core execution of parallelised hard real-time applications supporting analysability," in *Proc. of the 13th Euromicro Conference on Digital System Design*, 2013, pp. 363–370.

[2] M. Yang, T. Amert, K. Yang, N. Otterness, J. H. Anderson, F. D. Smith, and S. Wang, "Making OpenVX really "real time"," in *Proc. of the 39th IEEE Real-Time Systems Symposium (RTSS)*, 2018, pp. 80–93.

[3] *AUTOSAR Guide to Multi-Core Systems*, AUTOSAR, 03 2014.

[4] *Realtime and Embedded Specification for Java*, aicas GmbH, 2019.

[5] *Single UNIX® Specification, Version 4*, The Open Group, 2017, ch. 2.8 Realtime.

[6] *Ada Reference Manual*, Ada Conformity Assessment Authority, 2016, ch. Real Time Systems.

[7] J. W. McCormick, F. Singhoff, and J. Hugues, *Building parallel, embedded, and real-time applications with Ada*. Cambridge University Press, 2011.

[8] *XMOS Programming Guide*, XMOS Ltd., 2015.

[9] S. Gorlatch, "Send-receive considered harmful: Myths and realities of message passing," *ACM Transactions on Programming Languages and Systems*, vol. 26, no. 1, pp. 47–56, 2004.

[10] E. A. Lee, "The problem with threads," *IEEE Computer*, vol. 39, no. 5, pp. 33–42, 2006.

[11] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: A comprehensive study on real world concurrency bug characteristics," in *Proc. of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008, pp. 329–339.

[12] H. Sutter and J. Larus, "Software and the concurrency revolution," *ACM Queue*, vol. 3, no. 7, pp. 54–62, 2005.

[13] H.-M. Huang, C. Gill, and C. Lu, "MCFlow: A real-time multi-core aware middleware for dependent task graphs," in *Proc. of the 18th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2012, pp. 104–113.

[14] Q. He, X. Jiang, N. Guan, and Z. Guo, "Intra-task priority assignment in real-time scheduling of dag tasks on multi-cores," *IEEE Transactions on Parallel and Distributed Systems*, 2019.

[15] M. Cole, "Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming," *Parallel computing*, vol. 30, no. 3, pp. 389–406, 2004.

[16] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[17] M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Meneghin, and M. Torquati, "Accelerating code on multi-cores with FastFlow," in *Proc. of 17th International European Conference on Parallel and Distributed Computing (Euro-Par)*. Springer, 2011, pp. 170–181.

[18] J. Enmyren and C. W. Kessler, "SkePU: a multi-backend skeleton programming library for multi-gpu systems," in *Proc. of the 4th ACM International Workshop on High-Level Parallel Programming and Applications (HLPP)*. ACM, 2010, pp. 5–14.

[19] D. De Sensi, T. De Matteis, M. Torquati, G. Mencagli, and M. Danelutto, "Bringing parallel patterns out of the corner: The P^{3}ARSEC benchmark suite," *ACM Transactions on Architecture and Code Optimization*, vol. 14, no. 4, pp. 33:1–33:26, 2017.

[20] T. Ungerer, C. Bradatsch, M. Frieb, F. Kluge, J. Mische, A. Stegmeier, R. Jahr, M. Gerdes, P. G. Zaykov, L. Matusova, Z. J. J. Li, Z. Petrov, B. Böddeker, S. Kehr, H. Regler, A. Hugl, C. Rochange, H. Ozaktas, H. Cassé, A. Bonenfant, P. Sainrat, N. Lay, D. George, I. Broster, E. Quiñones, M. Panic, J. Abella, C. Hernández, F. J. Cazorla, S. Uhrig, M. Rohde, and A. Pyka, "Parallelizing industrial hard real-time applications for the parMERASA multicore," *ACM Transactions on Embedded Computing Systems*, vol. 15, no. 3, pp. 53:1–53:27, 2016.

[21] M. K. Chen, X. Li, R. Lian, J. H. Lin, L. Liu, T. Liu, and R. Ju, "Shangri-La: achieving high performance from compiled network applications while enabling ease of programming," in *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005, pp. 224–236.

[22] U. J. Kapasi, S. Rixner, W. J. Dally, B. Khailany, J. H. Ahn, P. Mattson, and J. D. Owens, "Programmable stream processors," *IEEE Computer*, vol. 36, no. 8, pp. 54–62, 2003.

[23] S. Benkner, E. Bajrovic, E. Marth, M. Sandrieser, R. Namyst, and S. Thibault, "High-level support for pipeline parallelism on many-core architectures," in *Proc. of 18th International European Conference on Parallel and Distributed Computing (Euro-Par)*. Springer, 2012, pp. 614–625.

[24] J. C. Beard, P. Li, and R. D. Chamberlain, "Raftlib: A C++ template library for high performance stream parallel processing," *The International Journal of High Performance Computing Applications*, vol. 31, no. 5, pp. 391–404, 2017.

[25] W. Thies and S. Amarasinghe, "An empirical characterization of stream programs and its implications for language and compiler design," in *Proc. of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2010, pp. 365–376.

[26] H. Kim, D. De Niz, B. Andersson, M. Klein, O. Mutlu, and R. Rajkumar, "Bounding memory interference delay in cots-based multi-core systems," in *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2014, pp. 145–154.

[27] M. Paolieri, E. Quiñones, F. J. Cazorla, G. Bernat, and M. Valero, "Hardware support for wcet analysis of hard real-time multicore sys-

tems," in *2009 ACM 36th Annual International Symposium on Computer Architecture (ISCA)*, vol. 37, no. 3. ACM, 2009, pp. 57–68.

[28] D. May, *xCORE-200: The XMOS XS2 Architecture*, XMOS, 2015.

[29] M. Steuwer, T. Remmelg, and C. Dubach, "Lift: a functional data-parallel ir for high-performance gpu code generation," in *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2017, pp. 74–85.

[30] P. Metzger, M. Cole, and C. Fensch, "NUMA optimizations for algorithmic skeletons," in *Proc. of 24th International European Conference on Parallel and Distributed Computing (Euro-Par)*. Springer, 2018, pp. 590–602.

[31] T. Lutz, C. Fensch, and M. Cole, "PARTANS: An autotuning framework for stencil computation on multi-gpu systems." *ACM Transactions on Architecture and Code Optimization*, vol. 9, no. 4, pp. 59:1–59:24, 2013.

[32] D. De Sensi, M. Torquati, and M. Danelutto, "A reconfiguration algorithm for power-aware parallel applications," *ACM Transactions on Architecture and Code Optimization*, vol. 13, no. 4, pp. 43:1–43:25, 2016.

[33] *X216-512-TQ128 Datasheet*, XMOS Ltd., 09 2018.

[34] C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat, "OTAWA: an open toolbox for adaptive WCET analysis," in *Proc. of the 8th IFIP International Workshop on Software Technolgies for Embedded and Ubiquitous Systems (SEUS)*, 2010, pp. 35–46.

[35] P. Corke, *Robotics, Vision and Control: Fundamental Algorithms In MATLAB® Second, Completely Revised*. Springer, 2017, vol. 118, p. 405.

[36] V. Kianzad and S. Bhattacharyya, "Efficient techniques for clustering and scheduling onto embedded multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 17, no. 7, pp. 667–680, 2006.

[37] G. Buttazzo, E. Bini, and Y. Wu, "Partitioning real-time applications over multicore reservations," *IEEE Transactions on Industrial Informatics*, vol. 7, no. 2, pp. 302–315, May 2011.

[38] Q. Tang, T. Basten, M. Geilen, S. Stuijk, and J.-B. Wei, "Mapping of synchronous dataflow graphs on mpsocs based on parallelism enhancement," *Journal of Parallel and Distributed Computing*, vol. 101, pp. 79–91, 2017.

[39] J. Sun, N. Guan, X. Wang, C. Jin, and Y. Chi, "Real-time scheduling and analysis of synchronous openmp task systems with tied tasks," in *Proceedings of the 56th Annual Design Automation Conference 2019*. ACM, 2019, p. 94.

[40] A. Stegmeier, M. Frieb, R. Jahr, and T. Ungerer, "Algorithmic skeletons for parallelization of embedded real-time systems," in *3rd Workshop on High-Performance and Real-time Embedded Systems (HiRES)*, 2015.

[41] L. M. Pinho, V. Nélis, P. M. Yomsi, E. Quiñones, M. Bertogna, P. Burgio, A. Marongiu, C. Scordino, P. Gai, M. Ramponi, and M. Mardiak, "P-SOCRATES: A parallel software framework for time-critical many-core systems," *Microprocessors and Microsystems*, vol. 39, no. 8, pp. 1190–1203, 2015.

[42] M. A. Serrano, A. Melani, R. Vargas, A. Marongiu, M. Bertogna, and E. Quiñones, "Timing characterization of OpenMP4 tasking model," in *Proc. of the IEEE International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, 2015, pp. 157–166.