Distributed Stable States for Process Networks – Algorithm, Analysis, and Experiments on Intel SCC

Devendra Rai, Lars Schor, Nikolay Stoimenov and Lothar Thiele
Computer Engineering and Networks Laboratory, ETH Zurich, 8092 Zurich, Switzerland
firstname.lastname@tik.ee.ethz.ch

ABSTRACT

Technology scaling is a common trend in current embedded systems. It has promoted the use of multi-core, multiprocessor, and distributed platforms. Such systems usually require run-time migration of distributed applications between the different nodes of the platform in order to balance the workload or to tolerate faults. Before an application can be migrated, it needs to be brought to a *stable state* such that restarting the application after migration does not violate its functional correctness. An application in a *stable state* does not change its context any further, and therefore, *stabilization* is a prerequisite for any application migration. Process networks are a common model of computation for specifying distributed applications. However, most results on the migration of process networks do not provide an algorithm to put a general process network into a *stable state*, suitable for migration. This paper proposes a technique which efficiently and correctly brings a process network executing on a distributed system to a known *stable state*. The correctness of the technique is independent of the temporal characteristics of the system and the topology of the process network. The required modifications of a process network are lightweight and preserve its original functionality. A model characterizing the timing properties of the technique is provided. The feasibility and efficiency of the proposed approach and the respective model are validated with experimental results on Intel's SCC platform.

1. INTRODUCTION

Nowadays, increasing computational demands in the embedded systems domain have required the use of distributed many-core platforms. One example is the automotive industry where a contemporary car has many driver assistant systems with tens of cameras, each of them supplying a video stream that needs to be processed in real-time.

However, the increased performance from such platforms comes at the price of increased power consumption per unit area. Such systems may experience high chip temperatures which may require that applications are migrated at runtime between different processing nodes in order to cool down parts of the chip. Moreover, load-balancing also requires runtime migration in order to optimize the performance.

Migration requires that upon detecting an event, an application can be brought to a *stable state* where all processes involved in migration have stopped their execution, collected all data packets sent to them, do not send any new data, and all local variables have been saved, including the program counters. Only when such a consistent state is reached, contexts can be saved correctly, applications (or parts of them) can be migrated and safely restarted from the point where they have been interrupted.

Bringing an application (or parts of it) to a stable state is not trivial when the application is distributed, composed of many asynchronously executing processes which do not share clocks or memory, with possibly asynchronous communication, and no prior knowledge of the amount of data being produced (or consumed) by a process in any given interval of time. Such is the case for applications specified as Kahn Process Networks (KPNs) [12]. The model is quite often used for specification and design of control and signal-processing applications which are ubiquitous today.

The paper focuses on the stabilization problem. Given a process network executing on a distributed memory system, upon the detection of an event, the process network needs to be brought to a stable state which is suitable for the migration of any of its processes. The technique should be lightweight, safe, correct, and work independently of the timing properties of the system or the topology of the process network.

The contributions of the paper are summarized as follows: 1) A technique is proposed that brings a process network executing on a distributed memory system to a stable state; 2) Timing analysis for the technique is provided; 3) Experiments are performed on a state-of-the-art multiprocessor system (Intel SCC [11]). They validate the efficiency and applicability of the technique, and the correctness of the provided timing model.

2. RELATED WORK

Lots of research results have been published on process migration techniques, see e.g. [1, 2, 4, 13], however, they usually target shared memory systems or do not provide any details on how to bring a general process network to a stable state where contexts can be saved correctly. Moreover, timing models are rarely provided or discussed. Similarly, this is the case for load-balancing literature [15, 18, 20].

A process migration technique for Polyhedral Process Networks (PPNs) has been proposed in [5]. PPNs are a restricted form of KPNs since all loop bounds, array indices, and index expressions must be affine expressions and a pro-

DAC '13, May 29 - June 07 2013, Austin, TX, USA.

cess cannot change these parameters at run-time. Therefore, the technique proposed in [5] is not applicable to general KPNs. In the proposed technique, a process execution can be stopped at any time. However, this may require re-execution of the same code after migration which is in contrast to our solution which does not require re-executions. Moreover, the approach in [5] relies on a complex middleware system that continues to run on the processing node even if the application is migrated which makes the technique unsuitable in cases the reason for migration is high temperature. In contrast, in our approach, the affected core can completely stop, after a known time, the *stabilization time*. Furthermore, the authors do not discuss memory requirements, timing properties, or the correctness of their stabilization technique.

Kernel-based approaches to do process migration usually require the usage of specific features of an operating system (OS) or modifications to the OS kernel making them non-portable, e.g. [6,16]. In this paper, we focus on solutions that work in user-space so that they do not depend on any specific OS features, guaranteeing the portability of the solution.

Checkpointing provides a means to manage the context of a migrating process, e.g., the Berkeley Labs Checkpointing and Restore (BLCR) algorithm [19]. However, checkpointing requires a fairly complex bookkeeping process where all processes must log all incoming tokens, all calculations, and all output tokens between each checkpoint. Thus, checkpointing can easily overwhelm the computational capabilities of a typical embedded system [21]. In this paper, we focus on a *lightweight* approach which avoids rolling back, but is able to bring the process network to a stable state which is ready for migration.

Chandy and Lamport [7] have proposed an approach to taking snapshots of a process network on a distributed system which is similar to the stabilization problem that we handle. However, they restrict themselves to (theoretical) systems with infinite FIFO channels and rule out the possibility that a process may block when attempting to write on a full output channel. In contrast, our technique is applicable to practical systems with bounded FIFO channels. Furthermore, we provide an implementation and a timing model which are validated with experiments.

3. MOTIVATIONAL EXAMPLES

In this section, we illustrate the challenges involved in the stabilization of a process network executing on a distributed memory system by using two simple examples.

Example 1: Decentralized Stabilization. Consider the process network shown in Fig. 1, with three processes v_1, v_2, and v_3 and the possibility that v_1 and v_2 must be stabilized. Assume that v_1 exits successfully, but v_3 is blocked when it attempts to read from the input FIFO $\mathcal{F}(v_3, 1)$ due to insufficient number of tokens. On the other hand, v_2 blocks when attempting to write to the full FIFO $\mathcal{F}(v_3, 2)$. This creates a deadlock and v_2 can never proceed to collect its context. In principle, process v_3 can be unblocked when process v_1 resumes normal operation after migration, allowing v_2 to also stabilize, and then migrate. However, such an approach serializes the stabilization of the process network, which may be unacceptable. The example shows that it should be possible to unblock processes that are performing possibly blocking operations (read/writes). Also some form of coordination is needed between processes v_1, v_2, and v_3 in order to lead them to stable states.

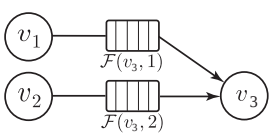

Figure 1: Example 1.

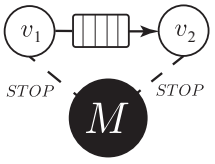

Figure 2: Example 2.

Example 2: Coordinated Stabilization. For the given discussion, assume that a separate master process is connected to all processes via event channels and it initiates the stabilization of the processes by sending a `stop` event. Consider the producer (v_1) and consumer (v_2) process network shown in Fig. 2. Once the affected processes receive the `stop` event, they can start the stabilization procedure.

Suppose that the master sends the `stop` event to both v_1 and v_2. Let v_2 receive the event before v_1. Stabilization requires that v_2 must not receive anymore data tokens from its parents, it must stop any further computation, and it must not transmit any tokens to its children. However, v_1 is not aware that v_2 is entering stabilization, and hence, v_1 keeps transmitting data tokens until it receives the `stop` event or its output channel becomes full. Process v_2 cannot go into a stable state because it does not know if v_1 has already received the `stop` event and which data token will be the last data token transmitted by v_1. If v_2 ignores any data tokens which continuously arrive over its input channels, then the execution context for v_2 cannot be calculated correctly and tokens may be lost.

The example shows that simply having a master which tries to bring all processes to a stable state is not sufficient. Due to different network latencies, processes may go into stable states at different times. Therefore, the correctness of the coordination technique needs to be time insensitive. Further, a mechanism is needed which indicates when all processes have stopped and special tokens have to be sent to child processes to notify that the last data token has arrived.

4. MODEL AND DEFINITIONS

A process network $\mathcal{N}$ is defined as the tuple $\mathcal{N} = (V, C, I, O, F, i, o, c, f)$, where V is a set of processes, C is a set of data channels, I is a set of input ports, O is a set of output ports, and F is a set of bounded first-in first-out (FIFO) buffers. The function $i : V \to \mathcal{P}(I)$ maps a process to a set of input ports, where $\mathcal{P}(S)$ denotes the power set of a set S. The function $o : V \to \mathcal{P}(O)$ maps a process to a set of output ports. Processes cannot share input or output ports, i.e., $i(v_k) \cap i(v_j) = \emptyset$ and $o(v_k) \cap o(v_j) = \emptyset$ for all $v_k, v_j \in V, v_k \neq v_j$. A process $v \in V$ reads data tokens from its input ports $i(v)$ and writes data tokens to its output ports $o(v)$. The function $c : U \to C$, where $U = \{(a, b) : a \in o(v_k), b \in i(v_j), v_k, v_j \in V, v_k \neq v_j\}$, maps pairs of output and input ports, belonging to different processes, to a data channel.

The function $par : V \to \mathcal{P}(V)$ returns the set of parent processes for a given process, and the function $ch : V \to \mathcal{P}(V)$ returns the set of child processes for a given process.

The function $f : V \times I \to F$ provides a FIFO buffer $\mathcal{F}(v, m)$ to an input port m of a process v. The buffer has a finite size denoted as $|\mathcal{F}(v, m)|$. A process attempting to write to an output port connected to an input port with a full FIFO will block until there is sufficient space available. Similarly, a process attempting to read data tokens from an input port with an empty FIFO will block until there are sufficient tokens available. Conventionally, KPNs [12] assume unbounded FIFOs, however, practical systems with finite

resources impose maximum sizes on the FIFOs [9, 10, 17]. Therefore, the notation adheres more closely to the typical implementation of a process network.

Bringing a process network $\mathcal{N}$ into a state that is ready for migration requires that each process $v \in V$ reaches a stable state which is defined as follows:

DEFINITION 1. *(STABLE STATE) A process $v \in V$ enters a stable state if it does not perform any more computations, parents $par(v)$ do not send any new data tokens, v does not send any new data tokens to its children $ch(v)$, and it has received all data tokens already sent by its parents $par(v)$.*

Once such a stable state has been reached, the context of each process can be saved and migrated. The context of a process includes all unread tokens (all not yet processed input tokens), all produced but unsent tokens, and the program state. Given this context, a process can be safely restarted.

Having the ability to bring each process to a stable state may require that the original processes are slightly modified. Such modifications should preserve the correct functionality of the network. Thus, it must be ensured that the original process network $\mathcal{N}$ is functionally equivalent to the modified process network $\mathcal{N}'$. In other words, the solution must comply with the notion of *Correctness* defined as:

DEFINITION 2. *(CORRECTNESS) Given two process networks $\mathcal{N}$ and $\mathcal{N}'$, where $\mathcal{N}'$ is a modified version of $\mathcal{N}$ such that it has mechanisms to be brought to a stable state. We say that $\mathcal{N}'$ is correct, if for any process $v' \in V'$ of $\mathcal{N}'$ which corresponds to process $v \in V$ of $\mathcal{N}$, for any vector of input sequences of data tokens In, the following relationship holds:*

$$In \xrightarrow[(v \in V)]{} Out \implies In \xrightarrow[(v' \in V')]{} Out \qquad (1)$$

where $In \xrightarrow[(v \in V)]{} Out$ means that process v produces the vector of output sequences of data tokens Out, when given with the vector of input sequences of data tokens In.

Thus, the overall problem of this paper can be summarized as: Extend the process network $\mathcal{N}$ to $\mathcal{N}'$, such that:

1. $\mathcal{N}'$ is functionally equivalent to $\mathcal{N}$;
2. $\mathcal{N}'$ can be brought into a stable state independent of computation or communication delays.

5. PROPOSED TECHNIQUE

Interrupting the normal execution of a process network is initialized and coordinated by a central authority, which can be either an external process or a process of the existing network. Without loss of generality, we assume that the central authority is an external process called the "master".

We start by defining a set of coordination events which will be used by the master for the communication with all processes. The set of coordination events $\mathcal{E}$ contains the `stop` event: a process receiving this event will *eventually* stop any computations and data transmissions to children, and must then acknowledge the reception of the event; and the `proceed` event: a process receiving this event must proceed to collect its context. The master sends the `proceed` event only when it has received all acknowledgments for the `stop` events.

In this paper we focus on stabilizing the entire process network, therefore, the master process always broadcasts the `stop` event to all processes in the network. However, the algorithm can be easily extended to stop only specific processes in the network, see Appendix B. Furthermore, a prototype implementation of the proposed stabilization technique is discussed in Appendix A.

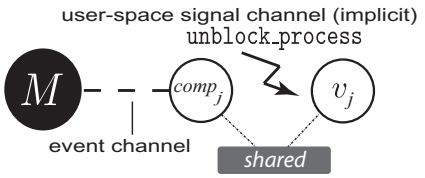

Figure 3: Process, companion process, master, event channel, and signal.

The master uses a (bidirectional) event channel $e_j \in E$ to communicate with process v_j. If the process is blocked because of reading from an empty FIFO or writing to a full FIFO, it will not be able to detect (and process) events from the master, therefore the event channel $e_j \in E$ is not directly connected to process v_j, but to a companion process $comp_j$, as shown in Fig. 3. The companion process $comp_j$ is very lightweight. It only receives and processes events from the master and makes them available to its process via a shared variable *shared*. When $comp_j$ receives the `stop` event, it sets the shared variable *shared* to `stop` and sends a signal `unblock_process` to v_j that cancels any blocking read or write of process v_j.

Process v_j checks the variable *shared* at the beginning of each communication primitive (a read or write statement) and exits normal execution if *shared* is set to `stop`. If process v_j is blocked and receives the signal `unblock_process`, it also exits normal execution, otherwise the signal has no effect. Once the process exits execution, it sets the shared variable *shared* to `done`, and then, the companion process can send an acknowledgment back to the master.

The above mechanism is only a conceptual description of our technique. The actual implementation of a separate companion process or unblocking signals will depend on the underlying platform.

5.1 Collecting Data Tokens

When a process exits normal execution, it executes a special function called $Wrapup$. Here no more data transmissions or computations are performed. First, the function waits until the process has received the `proceed` event from the master, i.e., meaning that all processes have suspended their normal computational activity. Then, it collects the process context and performs any other housekeeping steps such as returning any allocated memory.

During the collection of the context, a process must collect all tokens that are sent by its parents. If these tokens are not collected, the data tokens are "lost", which leads to incorrect behavior. This is further complicated by the fact that there might be a number of tokens arriving late due to late arrival of `stop` events in parent processes. Therefore, it must be ensured that there are no data tokens which are "in flight", i.e., written by a parent process but not yet received in the local FIFO of the child process. Otherwise, the technique would not be delay-independent.

In order to remedy this problem, in the $Wrapup$ function, each process, after receiving the `proceed` event, sends an end-of-stream (EOS) token to all its children. Thus, a process must continue to collect late arriving tokens from its input channels until the EOS marker has been received on each channel.

5.2 Bounding the Size of Contexts

For the purpose of discussion, the FIFO $\mathcal{F}_v(m)$ for an input port m of process v is divided into two FIFOs: $\mathcal{M}_v(m)$ and $\mathcal{L}_v(m)$. The size of $\mathcal{F}_v(m)$ is the sum of sizes of $\mathcal{M}_v(m)$ and $\mathcal{L}_v(m)$. Tokens move from $\mathcal{M}_v(m)$ to $\mathcal{L}_v(m)$ when there is sufficient space in $\mathcal{L}_v(m)$. The separation is made in order to reflect more closely real implementations of process networks, where $\mathcal{M}_v(m)$ refers to buffers of the interprocess communication layers and the capacity of communication links, while $\mathcal{L}_v(m)$ refers to the FIFO local to a process. The process has only the knowledge of the current status of $\mathcal{L}$ but not of $\mathcal{M}$.

We assume that the number of late-arriving tokens to each process can be bounded. This is the case for any NoC-based communication where the network capacity can be statically calculated (or at least upper bounded) by analyzing its topology, and it is the case for many communication libraries where the buffer sizes of data links are finite.

The memory space to absorb all late-arriving tokens is provided by a statically allocated set of "backup FIFOs" $\mathcal{B}$, in particular, one for each input port FIFO. The maximum number of late arriving tokens that a process v must absorb on each input port m and store in a backup FIFO is upper bounded by: $|\mathcal{B}_v(m)| = |\mathcal{M}_v(m)| + |\texttt{EOS}|$ where $|\mathcal{M}_v(m)|$ denotes the size of FIFO $\mathcal{M}_v(m)$, and $|\texttt{EOS}|$ denotes the size of the EOS marker. The bound is correct even if the local FIFO $\mathcal{L}_v(m)$ is full, since tokens in the input FIFOs are not considered as late arriving tokens and therefore not saved in the backup FIFOs. The backup FIFOs $\mathcal{B}$ are not available during the normal course of operation. Upon reception of the `proceed` event and before sending the EOS tokens, a process v_j swaps all regular FIFOs $\mathcal{L}_{v_j}(m)$ with the corresponding backup FIFOs $\mathcal{B}_{v_j}(m)$.

Consequently, an upper bound on the size of the context of process v is:

$$D_v^* = \sum_{\forall j} \{|\mathcal{L}_v(j)| + |\mathcal{B}_v(j)|\} + |\text{LN}| + |\text{LV}| \tag{2}$$

where $|\text{LN}|$ is the memory space required to store the line number of the program when v exited the normal execution, and $|\text{LV}|$ is the memory space required to store all local variables (loop indexes, unsent tokens, etc.).

Note that many existing solutions for migration do not rely on backup FIFOs but simply use a constantly running middle-ware system that will re-direct any late arriving data, no matter how late it is. However, such solutions are not always feasible if, for example, a processing node is close to reaching peak temperature and any processing activity on it needs to be stopped after a certain time.

5.3 Timing Analysis

The correct behavior of the proposed algorithm to stabilize a process network is delay-independent. However, in case that the maximum time to transmit a token between two processing nodes and the maximum time that a process is executing without calling a communication primitive are known, an upper bound on the overall stabilization time for a process network can be calculated. Such timing parameters can be obtained either with formal analysis and then the computed bounds would be hard real-time ones, or by measurements (or simulations), and then the bounds would be soft real-time ones.

In order to analyze the timing, we consider two phases of the algorithm. In the first one (denoted as $phase1$), the master (denoted as M) broadcasts the `stop` token to all processes and waits for all acknowledgments. In the second one (denoted as $phase2$), it broadcasts the `proceed` token and then each process waits until it receives an `EOS` marker on its input ports.

The maximum time between the instance when the master broadcasts the `stop` token and the instance it receives the acknowledgment from process v is composed of four time periods: (a) the maximum time $t^*_{M\to v}$ for the `stop` token to travel from the processing node of the master to the one of process v, (b) the maximum time $t^*_{read,v|write,v}$ that process v requires to perform a single read or write of a data packet of maximum size, (c) the maximum time $t^*_{c,v}$ that process v is executing without calling a communication primitive, and (d) the maximum time $t^*_{v\to M}$ for the `ack` token to travel from the processing node of process v to the one of the master. In other words, the master receives the acknowledgment from process v no later than after the following time period:

$$t^*_{phase1,v} = t^*_{M\to v} + t^*_{read,v|write,v} + t^*_{c,v} + t^*_{v\to M} \tag{3}$$

and can broadcast the `proceed` token no later than after the following time period:

$$t^*_{phase1,\mathcal{N}} = \max_{v\in V} \left\{t^*_{phase1,v}\right\}. \tag{4}$$

Afterwards, each process waits until it receives the `proceed` token, swaps all regular FIFOs with the backup FIFOs, and waits until it receives an `EOS` marker on each of its input ports. The time between the instance the master broadcasts the `proceed` token and the instance process v can start to collect its context is upper bounded either by the sequence that v receives the `proceed` token, swaps all regular FIFOs, and waits until it receives an `EOS` marker, or by the sequence that a parent of v receives the `proceed` token and then process v receives an `EOS`. Thus, $phase2$ takes no longer than the following time period:

$$\begin{aligned} t^*_{phase2,v} = \max\Bigg\{ & t^*_{M\to v} + \max_{u\in par(v)} \{t^*_{u\to v}\}, \\ & \max_{u\in par(v)} \{t^*_{M\to u} + t^*_{u\to v}\} \Bigg\} \end{aligned} \tag{5}$$

where $t^*_{u\to v}$ is the maximum time for the `EOS` marker to travel from the processing node of process u to the one of process v.

Finally, the stabilization time of process v and process network $\mathcal{N}$ is upper bounded by:

$$t^*_{stab,v} = t^*_{phase1,\mathcal{N}} + t^*_{phase2,v} \tag{6}$$

$$t^*_{stab,\mathcal{N}} = t^*_{phase1,\mathcal{N}} + \max_{v\in V} \left\{t^*_{phase2,v}\right\}. \tag{7}$$

The property of Correctness. The proposed technique requires three modifications of the process network, namely the addition of a companion process, a $Wrapup$ function, and a conditional check before proceeding with a blocking read or write. The addition of the companion process $comp_j$ to process v_j does not change the order of any tokens in any channel for any process. It only retains the information that an event $\{\texttt{stop}, \texttt{proceed}\} \in \mathcal{E}$ was dispatched from the master. The $Wrapup$ function simply stores late-arriving tokens in backup FIFOs, maintaining the relative order of arrival of tokens. Finally, the conditional check before proceeding with a blocking read or write does not interfere with computations or the tokens that are already read or need to be written. Thus, all three modifications preserve the original functionality of the process network and the correctness property is satisfied.

Table 1: Measured stabilization time $t_{stab,v}$ vs. its upper bound $t^*_{stab,v}$ for each process of the Demosaicing application. In addition, the maximum context size D_v is compared to its upper bound D^*_v.

process	$t_{phase1,v}$		$t^*_{phase1,v}$	$t_{phase2,v}$		$t^*_{phase2,v}$	$t_{stab,v}$		$t^*_{stab,v}$	D_v	D^*_v
	avg	max		avg	max		avg	max		max	
load image	0.024 s	0.13 s	0.13 s	19.46 μs	21.39 μs	29.33 μs	0.77 s	3.26 s	4.20 s	52 B	52 B
pre processing	0.042 s	0.16 s	0.16 s	40.80 μs	44.39 μs	48.12 μs	0.77 s	3.26 s	4.20 s	632472 B	632472 B
pre demosaicing	0.32 ms	0.55 ms	0.55 ms	77.07 μs	78.03 μs	79.74 μs	0.77 s	3.26 s	4.20 s	36 B	632484 B
demosaicing_0	0.41 s	1.04 s	1.05 s	60.58 μs	61.54 μs	66.91 μs	0.77 s	3.26 s	4.20 s	157704 B	163370 B
demosaicing_1	0.53 s	1.05 s	1.07 s	60.30 μs	62.63 μs	66.91 μs	0.77 s	3.26 s	4.20 s	160296 B	163370 B
demosaicing_2	0.61 s	1.04 s	1.07 s	58.70 μs	60.98 μs	66.91 μs	0.77 s	3.26 s	4.20 s	160296 B	163370 B
demosaicing_3	0.60 s	1.05 s	1.05 s	58.86 μs	61.32 μs	66.91 μs	0.77 s	3.26 s	4.20 s	157704 B	163370 B
post demosaicing	0.70 s	2.28 s	2.40 s	149.09 μs	159.04 μs	179.65 μs	0.77 s	3.26 s	4.20 s	338964 B	2538900 B
post processing	0.77 s	3.26 s	4.20 s	73.63 μs	76.26 μs	80.43 μs	0.77 s	3.26 s	4.20 s	620750 B	1255992 B
write result	0.36 ms	0.53 ms	0.57 ms	37.02 μs	37.32 μs	56.37 μs	0.77 s	3.26 s	4.20 s	368 B	623460 B
process network		3.26 s	4.20 s		159.04 μs	179.65 μs		3.26 s	4.20 s	2176.4 KB	6188.3 KB

6. EXPERIMENTS

The feasibility and efficiency of the proposed stabilization technique are validated using two representative multiprocessing benchmarks: Demosaicing and a distributed Motion-JPEG (MJPEG) decoder algorithm, detailed in Appendix C. We aim to measure the time to bring the benchmark applications into a stable state, and to compare the time with the (theoretical) upper bound described in Section 5.3. The experiments were performed on Intel's SCC platform [11], a 48-cores (24-tiles) experimental prototype of future on-chip many-core platforms detailed in Appendix C.1.

Experimental Setup. Both benchmarks are running bare-metal to avoid timing jitter due to the operating system. Cache-related timing variations are reduced by hosting one process per tile. Since the SCC implements a deterministic X-Y routing, timing variations due to router contention are reduced by carefully binding the processes onto the tiles. Inter-process communication is implemented using the iRCCE library [8]. For the timing measurements, all tiles establish a common time reference when they boot using the barrier operation available in the communication library. L2 caches and interrupts are disabled on all tiles. Data messages are at most of size 3 KB each (longer ones are split) and control tokens are of size 16 B. The master process was placed on a separate tile so that it does not interfere with the application.

In order to achieve our goal, i.e., to compare the observed stabilization time with its upper bound, we proceed in three steps: 1) Calibration experiments are performed to derive a communication model of the target platform and to obtain the characteristics of the benchmark applications. As a result of this step, we calculate the upper bounds $t^*_{phase1,v}$, $t^*_{phase2,v}$, D^*_v for each process v, and $t^*_{stab,\mathcal{N}}$ for the network, see Section 5. 2) Stabilization experiments of the benchmarks are executed to observe the actual time taken by a process v to complete $phase1$ and $phase2$, the time $t_{stab,v}$ to stabilize, and the context size D_v. 3) The observed values are compared with the bounds calculated in Step 1.

Calibration. The communication model was derived by observing the time taken to deliver a packet with size ranging from 4 B through 3 KB over hop distances ranging from one through eight. A total of 585 observations were made. The communication latency under high cross-traffic between any two processes u and v (including the master) mapped onto different tiles, was observed to be upper bounded by:

$$t^*_{u\to v} = 5.182|P| + 9935 \quad \text{(cl.cycles)} \tag{8}$$

where $|P|$ is the size of the payload in bytes. Because of the high cross-traffic, a dependency on the number of hops between the processing nodes of the communicating processes is not observed.

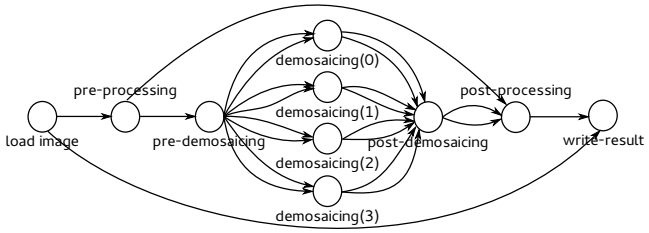

Figure 4: The Demosaicing application.

Another set of calibration experiments was performed in order to obtain the maximum computation time $t^*_{c,v}$ for each process. The Demosaicing application was executed using five RAW images of different sizes and for the MJPEG decoder, the execution time of each process was measured over each frame. The detailed results of all calibration experiments are reported in Appendix C.

Demosaicing. Demosaicing [14] is both a compute and data intensive application consisting of 10 processes, see Fig. 4. To measure the stabilization times, the experiment was repeated 20 times with different inputs and randomly varying the instants at which the master starts a `stop` token broadcast. Both the average and maximum values of the 20 runs are reported.

Table 1 compares the measured stabilization time $t_{stab,v}$ with the calculated upper bound $t^*_{stab,v}$. It can be seen that all processes did indeed stabilize before the expected time bounds. For some of the processes, the observed measurements are very close to the expected upper bounds. This means that the estimated bounds can be very accurate. The gaps between observed values and bounds are explained by the fact that $t^*_{stab,v}$ is mainly composed of the time the master waits until it receives all acknowledgments, and considers that a process can be in its longest computation section $t^*_{c,v}$. As shown in Table 1, $t^*_{c,v}$ is particularly large for the post processing process.

In addition, the maximum measured size of the context D_v is compared in Table 1 with its upper bound D^*_v calculated using Eq. (2). For some of the processes, the observed measurements are equal to the estimated upper bounds which means that equation (2) is tight. The total size of the upper bound is about three times larger than the measured maximum size. The former assumes that all FIFO channels are full when the context is calculated, but in practice, some of the channels are only partly filled.

MJPEG Decoder. The second example is a parallelized version of the MJPEG decoder application taken from the benchmark suite of the Artist Network of Excellence [3].

Table 2: Measured stabilization time $t_{stab,v}$ vs. its upper bound $t^*_{stab,v}$ for each process of the MJPEG decoder.

process	$t_{phase1,v}$		$t^*_{phase1,v}$	$t_{phase2,v}$		$t^*_{phase2,v}$	$t_{stab,v}$		$t^*_{stab,v}$
	avg	max		avg	max		avg	max	
trigger	55.5 μs	129 μs	155 μs	20.1 μs	23.7 μs	24.1 μs	401 μs	671 μs	899 μs
splitstream	102 μs	157 μs	167 μs	31.1 μs	44.8 μs	47.7 μs	412 μs	698 μs	923 μs
splitframe	48.8 μs	96.6 μs	98.9 μs	49.4 μs	76.4 μs	77.5 μs	430 μs	699 μs	952 μs
iqzigzagidct	380 μs	653 μs	875 μs	47.5 μs	74.2 μs	78.4 μs	428 μs	716 μs	953 μs
mergeframe	71.1 μs	116 μs	116 μs	43.1 μs	65.1 μs	66.9 μs	424 μs	683 μs	942 μs
mergestream	53.8 μs	107 μs	114 μs	24.7 μs	37.2 μs	37.6 μs	405 μs	687 μs	913 μs
process network		653 μs	875 μs		76.4 μs	78.4 μs		699 μs	953 μs

Table 3: Overhead in terms of execution time and binary code size for adding the ability to stabilize compared to the original implementation.

application	memory overhead	time overhead
Demosaicing	8624 B	43.01 μs($< 0.05\%$)
MJPEG decoder	7104 B	43.01 μs($< 0.05\%$)

The application consists of six processes and its structure is outlined in Fig. 5.

Similar to the first benchmark example, we compare the measured stabilization time $t_{stab,v}$ of each process v with its upper bound $t^*_{stab,v}$, see Table 2. The experiment was repeated 20 times and the average and maximum results are reported in Table 2. The results confirm the trend observed with the Demosaicing application. In particular, all processes stabilized before the expected time bounds. In many cases, the bounds are actually very accurate. For the MJPEG decoder application, the upper bound is mainly composed of the maximum execution time $t^*_{c,v}$ of the iqzigzagidct process.

Time and Memory Overheads. The time and memory overhead generated by the additional code required to accomplish process network stabilization is presented in Table 3. The time overhead is mainly due to the additional logic to check for the `stop` token from the master and related housekeeping activities. In particular, an individual checking for the `stop` token has taken on average 43.01 μs.

Summary. Using realistic applications, it has been shown that the proposed technique can bring a process network to a stable state. Performance metrics such as upper bounds on the maximum stabilization time and maximum context sizes are also presented. The maximum stabilization time is dominated by the maximum time a process can execute without calling any communication primitive, i.e., *process compute time*. It may be possible to further reduce the stabilization time by inserting additional checks for events in the process' compute segments. Finally, detailed results from experiments on the Intel SCC baremetal platform were presented, validating the ideas presented in this paper.

7. CONCLUSION

The paper presented a technique to bring a process network executing on a distributed system into a stable state, suitable for migration. The proposed technique has been shown to be lightweight, and preserves the original functionality of the network. The correctness of the technique has been shown to be independent of the temporal characteristics of the system and the topology. We have shown that if the token communication time and *process compute time* are upper bounded, then an upper bound on the overall stabilization time can be calculated. Finally, we validated the feasibility and efficiency of the proposed approach and the respective timing models with representative experiments on Intel's SCC platform.

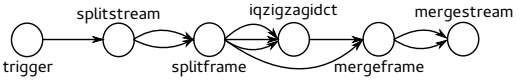

Figure 5: The MJPEG application.

8. REFERENCES

[1] A. Acquaviva et al. Assessing Task Migration Impact on Embedded Soft Real-Time Streaming Multimedia Applications. *EURASIP J. Embedded Syst.*, pages 9:1–9:15, 2008.

[2] G. M. Almeida et al. An Adaptive Message Passing MPSoC Framework. *Int'l J. of Reconfigurable Computing*, 2009.

[3] Artist. Benchmarks. `http://www.artist-embedded.org/artist/Benchmarks.html`, 2008.

[4] S. Bertozzi et al. Supporting Task Migration in Multi-Processor Systems-on-Chip: A Feasibility Study. In *Proc. DATE*, pages 15–20, 2006.

[5] E. Cannella et al. Adaptivity Support for MPSoCs Based on Process Migration in Polyhedral Process Networks. *VLSI Design*, pages 2:2–2:17, 2012.

[6] S. Chakravorty et al. Proactive Fault Tolerance in MPI Applications via Task Migration. In *Proc. HPC*, pages 485–496, 2006.

[7] K. M. Chandy. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Trans. on Computer Systems*, 3:63–75, 1985.

[8] C. Clauss et al. iRCCE: A Non-Blocking Communication Extension to the RCCE Communication Library for the Intel Single-chip Cloud Computer. Technical report, RWTH Aachen, 2011.

[9] M. Geilen and T. Basten. Kahn Process Networks and a Reactive Extension. In *Handbook of Signal Processing Systems*, pages 967–1006. Springer, 2010.

[10] W. Haid et al. Efficient Execution of Kahn Process Networks on Multi-Processor Systems Using Protothreads and Windowed FIFOs. In *Proc. ESTIMedia*, pages 35–44, 2009.

[11] J. Howard et al. A 48-Core IA-32 Message-Passing Processor with DVFS in 45nm CMOS. In *Proc. ISSCC*, pages 108–109, 2010.

[12] G. Kahn. The Semantics of a Simple Language for Parallel Programming. In *Proc. IFIP Congress*, pages 471–475, 1974.

[13] C. Lee, H. Kim, H.-W. Park, S. Kim, H. Oh, and S. Ha. A Task Remapping Technique for Reliable Multi-Core Embedded Systems. In *Proc. CODES/ISSS*, pages 307–316, 2010.

[14] X. Li. Demosaicing by successive approximation. *Trans. Img. Proc.*, 14(3):370–379, 2005.

[15] C. Lu and S.-M. Lau. A Performance Study on Load Balancing Algorithms with Task Migration. In *Proc. TENCON*, pages 357–364, 1994.

[16] D. F. Mark Claypool. Transparent Process Migration for Distributed Applications in a Beowulf Cluster. In *Proc. INC*, pages 459–466, 2002.

[17] H. Nikolov et al. Systematic and Automated Multiprocessor System Design, Programming, and Implementation. *IEEE Trans. Comput. Aided Design*, 27(3):542–555, 2008.

[18] J.-C. Ryou and J. Wong. A Task Migration Algorithm for Load Balancing in a Distributed System. In *Proc. System Sciences*, pages 1041–1048, 1989.

[19] S. Sankaran et al. The LAM/MPI Checkpoint/Restart Framework: System-Initiated Checkpointing. *Journal of High Performance Computing Applications*, 19(4):479–493, 2005.

[20] T. Suen and J. Wong. Efficient Task Migration Algorithm for Distributed Systems. *IEEE Trans. on Parallel and Distributed Systems*, 3:488–499, 1992.

[21] C. Wang et al. Hybrid Checkpointing for MPI Jobs in HPC Environments. In *Proc. ICPADS*, pages 524–533, 2010.

APPENDIX

A. PROTOTYPE IMPLEMENTATION

In this section, we illustrate a prototype implementation of the proposed stabilization technique.

A.1 Process Network Specification

We start with illustrating a high-level API for specifying process networks. A process $v \in V$ starts executing by first initializing itself at Line 2 in Algorithm 1, and then repeatedly invoking the *Fire* function at Line 4. The function can consist of any number and order of data-token read/write steps, and compute steps (e.g. branches, loops, assignments, etc.) depending on the actual functionality of the process.

Algorithm. 1: Basic structure of a process $v \in V$.

```
1: process v
2:     Init();                                   ▷ Initialization
3:     while true do                             ▷ Call Fire repeatedly
4:         Fire();                               ▷ Communication and computation
5:     end while
6: end process
```

A.2 Integrating the Stabilization Mechanism

Next, we will illustrate how the original process network specification from Algorithm 1 can be extended to integrate the stabilization mechanism. The structure of a process network should be minimally modified so that each process obtains the ability to go into the stable state and collect its context. The modifications discussed in Section 5 are incorporated into the pseudo-code shown in Algorithm 2. First, notice that Lines 10-19 are thread-safe. Line 10 checks for the `stop` event *before* starting a potentially blocking data token read or write step. If no event has been posted, the process starts the data-token read or write step. If the process is blocking on a read or write step while the `stop` event from the master is received, the signal `unblock_process` will unblock the process.

Algorithm. 2: New Structure of process v_j.

```
1: process v_j
2:     Init();                                   ▷ Initialization
3:     while !cancelled do          ▷ Until the stop event cancels further execution
4:         cancelled ← Fire();      ▷ Communication and computation
5:     end while
6:     Wrapup();   ▷ Call the Wrapup function to finish stabilization
7:     function Fire
8:         ...
9:                 ▷ Must not start read or write step if shared is stop
10:        if shared = stop then
11:            shared ← done
12:            return (cancelled ← true);
13:        else
14:            Start a blocking R/W step
15:            if unblocked by signal then
16:                shared ← done
17:                return (cancelled ← true)
18:            end if
19:        end if
20:        ...
21:    end function
22: end process
```

Unblocking upon reception of an event is easily accomplished by using user-space signals from threading libraries such as the POSIX library. Thus, the reception of the `stop` event by a process effectively cancels the currently blocked token-write or token-read operation.

A.3 The Wrapup Function

The *Wrapup* function is introduced in Section 5.1. Pseudo-code illustrating the function is given in Algorithm 3. First, in Line 2, it waits until it receives the `proceed` event from the master. It swaps all regular FIFOs $\mathcal{L}(v_j, m)$ with the corresponding backup FIFOs $\mathcal{B}(v_j, m)$ in Line 3. Afterwards, it sends an end-of-stream (EOS) token to all its children and waits in Line 5 until it receives an EOS marker from all its parents. Finally, some cleanup operations are performed to return the memory to the system.

Notice that swapping of the local FIFO $\mathcal{L}(v_j, m)$ with the backup FIFO $\mathcal{B}(v_j, m)$ preserves the correctness of the process network. This is because:

- A backup FIFO is brought online *only* after reception of the `proceed` signal. Note that a companion process transmits the acknowledgement only when the process changes the shared variable to `done`. Thereafter, the algorithm guarantees that the process will not transmit any more tokens. Therefore, in the worst case, the backup FIFO must be able to accommodate the tokens which are still in flight, which are upper bounded to $|\mathcal{M}(m)| + |\texttt{EOS}|$.
- The only token that a process transmits post-reception of the `proceed` event is the $|\texttt{EOS}|$ marker, which is accommodated in the backup FIFO.

Therefore, it can be seen that (assuming that the communication network is lossless), none of the data tokens are lost in the process of stabilization. Further, the FIFO data structure maintains the relative ordering on the tokens on each channel.

Algorithm. 3: Basic structure of the *Wrapup* function.

```
1: function Wrapup
2:     Wait-to-proceed();   ▷ Wait for proceed event from master
3:     Switch L(v_j, k) with B(v_j, k);
4:     Forward-EOS();       ▷ Forward EOS token to all children
5:     Collect-tokens();    ▷ Collect "late-arriving" tokens
6:     Cleanup();           ▷ Return memory to the system, etc.
7: end function
```

A.4 The Companion Process

The companion process $comp_j$ of process v_j is responsible for the communication of process v_j with the master process. Algorithm 4 describes the companion process $comp_j$, which is executed independently of process v_j. The assumption is that `stop` events from the master cannot overlap. In particular, the companion process waits for an event of the master. If it finds a `stop` event, it updates the *shared* variable and waits until the variable is set to `done`. Afterwards, it sends an acknowledgement to the master and waits until it receives the `proceed` event.

It is possible to optimize this structure such that not every process in the process network has a companion process, but instead a group of processes residing on one processing element share a companion process. However, such optimization is beyond the scope of the paper.

As a process v_j might be blocked because of reading from an empty FIFO or writing to a full FIFO channel, the companion process $comp_j$ has to be implemented as an additional object that is running in parallel to process v_j and just shares a single variable with process v_j. In case the platform supports multi-threading, the companion process

Algorithm. 4: Pseudo-code illustrating the functionality of the companion process $comp_j$.

```
1: process comp_j                ▷ Runs as a separate concurrent process
2: while true do
3:     Read event from event channel
4:     if found stop then
5:         shared ← stop
6:     end if
7:     Sleep on the shared variable until it is changed to done
8:     Send acknowledgment to master
9:     Read event from event channel
10:    if found proceed then
11:        shared ← proceed;
12:    end if
13: end while
14: end process
```

$comp_j$ can be implemented as an additional thread. Otherwise, one can use stack-less threads as described in section A.6. As the companion process is in a known state when the stabilization is completed, the companion process is not part of the context of process v_j, but can be re-initialized after migration.

A.5 Additional Note on Stabilization Time

It can be seen from (7) that the overall stabilization time depends on the time it takes for *phase 1* and *phase 2* to complete. The length of phase 1 is dominated by the maximum *process compute time*. The length of phase 2 is dominated by network speed, with time taken to swap FIFO being negligible. Notice from Algorithm 2, lines 14 - 17 that the a write by a process can be canceled by a signal, and therefore, the stabilization time is (largely) independent of the amount of data written by a process. The network speed accounts for a small part in the overall duration of the stabilization time, since a maximum of $|\mathcal{M}_v(m)|$ tokens need to be collected in phase 2 before the stabilization is complete.

A.5.1 |EOS| Send and Receive Times

Equation (5) does not consider the |EOS| send and receive times by a process individually. This is because the time it takes for a process v to receive the |EOS| token from its parent already covers the time it takes for v's parent to send the |EOS| tokens (plus a small communication time), and thus the use of $t^*_{u\to v}$.

A.6 Implementing Unblocking

Finally, we present an overview of how the unblocking functionality is implemented in the prototype implementation used in Section 6. There are various mechanisms that provide the ability to cancel a blocked read or write operation. For example, one could separately schedule each block of code between two communication calls using a kernel-space thread. However, this might lead to a large scheduling overhead due to the full preemption and memory protection, both undesired when scheduling such code blocks. Another option is to use stack-less threads as, for example, protothreads [1]. Protothreads have already been successfully applied to KPNs to provide lightweight scheduling [2]. The functionality of protothreads is implemented as a set of macros that enclose the communication calls. The embedding of a KPN process into a protothread process can also be automated at the software synthesis step.

In protothread, a control structure is used to store the local data of a process together with a variable that represents the line number of the process. Whenever the process exits the *Fire* function, it updates this variable either to the current line number or, if the process has reached the end of the *Fire* function, to the beginning of the *Fire* function. On the other hand, at the beginning of the *Fire* function, the line number variable of the control structure is read and the program counter jumps to this line. In order to extend the protothread library with the unblocking functionality, we change the process structure in two ways: First, we extend the PT_WAIT_UNTIL macro of protothreads with the abilities to check for the *shared* variable and to be unblocked by the unblock_process signal, as outlined in Algorithm 2. Originally, the PT_WAIT_UNTIL macro just blocked a process until the read or write is successful. Second, we enclose each communication call with the extended PT_WAIT_UNTIL macro to obtain the functionality described in Algorithm 2.

A.7 Summary of Distributed Process Network Stabilization Approach

A short summary of the approach is presented here:

1. The algorithm stabilizes the distributed process network *correctly*, i.e., no tokens are lost, and relative ordering amongst tokens in each channel is maintained. All tokens not consumed by a process v are stored in the process context in the correct order.
2. POSIX signals are used to unblock a process blocked on a full output FIFO or an empty input FIFO. The Intel SCC implementation uses a custom communication layer with the following properties:
   - Blocks a process on an empty input FIFO or a full output FIFO;
   - Unblocks the process upon receiving the *signal* from the *master*. All messages are broken into *chunks*. The maximum chunk-size is carefully selected so that the each core has *guaranteed* space to receive the messages from the master. The communication layer intersperses normal message read and write steps with checking for control messages from the master.
3. The stabilization procedure is composed of two phases, *phase 1* and *phase 2*.
   - Phase 1 brings all processes in the process network into a *known* state. The completion of phase 1 guarantees that no process in the network performs any further compute, write, or read steps. The length of phase 1 is dominated by *process compute times*.
   - Phase 2 required each process to swap local FIFO with backup FIFOs. The backup FIFOs are sized appropriately in order to accommodate all possible in-flight tokens from each of process' parents, plus the |EOS| token. Phase 2 is determined by the communication times, and hence dependent upon the network characteristics.
4. The algorithm is independent of the size of local FIFOs, and is the same as in the original process network. The size of backup FIFOs are statically determined.

B. STABILIZING INDIVIDUAL PROCESSES

Our main section introduced an approach in which the entire process network was stabilized. However, the same principles can be applied to stabilize a part of the process network. Consider a process $v \in V$, which must stabilized.

Therefore, all parents of v are informed of the stabilization of v, and as a result, process v receives the `EOS` from all its parents. The process v must also send `EOS` to all its children, so that v's children do not continue to expect tokens from the previous location of v. Once process v stabilizes, it can migrate. Subsequently, new channels must be established between the parents of v and v's children. Since the network structure is statically known, recreating channels is straightforward.

C. ADDITIONAL EXPERIMENTAL RESULTS

In addition to the results presented in Section 6, we summarize the Intel SCC platform and other experimental results in this section.

C.1 Intel Single-Chip Cloud Computer

The Intel Single-chip Cloud Computer (SCC) is a prototype of future embedded on-chip many-core platforms [3]. The processor consists of 24 tiles that are organized into a 4×6 grid and linked by a 2D mesh on-chip network. A tile contains a pair of P54C processor cores, a router, and a 16 KB block of SRAM. Each core runs at 533 MHz and each router runs at 800 MHz. The on-tile SRAM block is also called "message passing buffer" (MPB) as it enables the exchange of information between cores in the form of messages. Figure 6 schematically outlines the SCC processor.

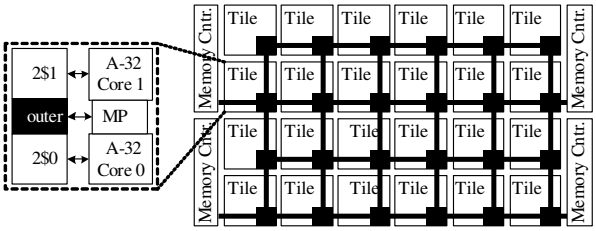

Figure 6: Schematic representation of Intel's SCC processor [3].

C.2 Additional Calibration Results

In Section 6, we have shown that the communication latency under high cross-traffic between any two processes u and v can be upper bounded. In particular, we have seen that the latency is independent of the number of hops between the processing nodes of the communicating processes. In addition, we observed the communication latency under low data traffic conditions, which is bounded by:

$$t_{u \to v}^{\text{low traffic}} = 5.182|P| + 307.4H + 531 \quad \text{(cl.cycles)} \qquad (9)$$

Here, the latency depends on the number of hops H between the nodes of the communicating processes. However, this model cannot be used as an upper bound as it considers only low data traffic in the network.

C.3 The Demosaicing Benchmark: Additional Results

Table 4 summarizes the characteristics of the Demosaicing application obtained when measuring the computation and read/write times of the individual processes. The binding reports the identification number of the SCC core, on which the process is executed. The maximum execution time corresponds to the maximum time that a process is executing without calling a communication primitive and is calculated when the Demosaicing application was executed under five RAW images of different sizes. Finally, the maximum amount of data that is transmitted per outgoing channel and iteration is reported. The values given in Table 4 have been used in Table 1 to calculate the upper bounds on the stabilization times.

Table 4: Characteristics of the Demosaicing application.

process	binding	max. execution time	port	max. data / iteration
load image	26	0.13 s	0	618 KB
pre processing	12	0.13 s	0	618 KB
pre demosaicing	14	0.4 ms	1 2 3 4 5 6 7 8	160 KB 2 B 160 KB 2 B 160 KB 2 B 160 KB 2 B
demosaicing_0	28	1.05 s	2 3	160 KB 471 KB
demosaicing_1	18	1.07 s	2 3	160 KB 471 KB
demosaicing_2	02	1.07 s	2 3	160 KB 471 KB
demosaicing_3	40	1.05 s	2 3	160 KB 471 KB
post demosaicing	30	2.4 s	8 9	618 KB 618 KB
post processing	20	4.2 s	2	618 KB
write result	22	0.5 ms		

Context Sizes: More Details.

The overall context sizes reported in Table 1 is the sum of:

- The space required to store the line number (in order to restore the context), local variables (except those which store unsent output data tokens, and unread input data tokens), denoted as S_1.
- The space required for storing unsent output data tokens, and unread input data tokens, denoted as S_2.

Therefore, S_2 can be considered as the application-dependent context storage requirement, while S_1 is largely independent of the application. The contribution due to S_1 in the overall context sizes reported in Table 1 is presented in Table 5. It is clear from Table 5 that the total size of the context is dominated by the nature of the application itself.

C.4 The MJPEG Benchmark: Additional Results

Next, the characteristics of the MJPEG decoder application are summarized in Table 6. The following values are shown: The identification number of the SCC core, on which the process is executed; the maximum time that a process is executing without calling a communication primitive; and the maximum amount of data that is transmitted per outgoing channel and iteration. The time of the longest compute segment was measured over each frame of an example video with resolution 320×240 pixels.

Table 5: Context size *exclusively* due to S_1.

process	context size for S_1
load image	48
pre processing	24
pre demosaicing	36
demosaicing_0	72
demosaicing_1	72
demosaicing_2	72
demosaicing_3	72
post demosaicing	84
post processing	152
write result	68

Table 6: Characteristics of the MJPEG decoder application.

process	binding	max. execution time	port	max. data / iteration
trigger	02	58 μs	0	4 B
splitstream	04	167 μs	0 1	4 B 10 KB
splitframe	06	98 μs	2 3 4 5	307.2 KB 64 B 4 B 8 B
iqzigzagidct	18	875 μs	3	76.8 KB
mergeframe	20	116 μs	2 3	7.68 KB 8 B
mergestream	22	114 μs		

D. REFERENCES

[1] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali. Protothreads: Simplifying Event-Driven Programming of Memory-Constrained Embedded Systems. In *Proc. SenSys*, pages 29–42, 2006.

[2] W. Haid et al. Efficient Execution of Kahn Process Networks on Multi-Processor Systems Using Protothreads and Windowed FIFOs. In *Proc. ESTIMedia*, pages 35–44, 2009.

[3] J. Howard et al. A 48-Core IA-32 Message-Passing Processor with DVFS in 45nm CMOS. In *Proc. ISSCC*, pages 108–109, 2010.