SPES

Software Plattform Embedded Systems 2020

GEFÖRDERT VOM

Bundesministerium für Bildung und Forschung

Deliverable ZP-D5.1.A-a: Konzept für die Erweiterung bestehender paralleler Programmiermodelle um Determinismus in Echtzeitsystemen

Version: 1.1

Projektbezeichnung	SPES 2020	
Verantwortlich	Tobias Schüle	
QS-Verantwortlich		
Erstellt am	12.10.2009	
Zuletzt geändert	06.03.2012 10:33	
Freigabestatus		Vertraulich für Partner:
		Projektöffentlich
	X	Öffentlich
Bearbeitungszustand		in Bearbeitung
		vorgelegt
	X	fertig gestellt

Weitere Produktinformationen

Erzeugung	Tobias Schüle
Mitwirkend	Urs Gleim

Änderungsverzeichnis

Änderung			Geänderte Kapitel	Beschreibung der Änderung	Autor	Zustand
Nr.	Datum	Version				
1	12.10.2009	1.0	Alle	Initiale Produkterstellung	T. Schüle	
2	06.03.2012	1.1	Deckblatt	Kleinere Korrekturen	T. Schüle	

Kurzfassung

In modernen Embedded Systems kommen zunehmend Multi-Core-Prozessoren zum Einsatz, bei denen mehrere unabhängige Prozessorkerne auf einem Chip untergebracht sind. Multi-Core-Prozessoren bieten höhere Rechenleistungen als konventionelle Prozessoren bei gleichzeitig reduziertem Energieverbrauch. Um von den Vorteilen von Multi-Core-Prozessoren profitieren zu können, müssen darauf auszuführende Anwendungen explizit parallelisiert werden. Dies erfordert insbesondere für die Entwicklung sicherheitskritischer, hoch performanter softwarebasierter Embedded Systems geeignete Programmiermodelle und -werkzeuge. In diesem Dokument wird ein Programmiermodell und eine Sprache für die Entwicklung von parallelen Embedded Systems vorgestellt.

Inhalt

1 Einordnung und Kurzbeschreibung

1.1 Motivation und Einordnung

Bei dem in der Praxis dominierenden Programmiermodell mittels Threads wird Parallelität explizit durch den Programmierer spezifiziert. Die Synchronisation mehrerer Threads bei Zugriffen auf gemeinsame Ressourcen erfolgt ebenfalls explizit mittels Konstrukten für den gegenseitigen Ausschluss. Die Programmierung mit Threads ist jedoch häufig ein mühsames und fehlerträchtiges Unterfangen. Ein weiterer Nachteil von Threads ist ihr potentieller Nichtdeterminismus, der die Vorhersagbarkeit des Verhaltens und damit auch die Fehlersuche erheblich erschwert. Insbesondere in eingebetteten Systemen, die üblicherweise Echtzeitanforderungen unterliegen, spielt ein Höchstmaß an Vorhersagbarkeit eine wichtige Rolle, um mögliche Fehler vor der Auslieferung eines Systems entdecken zu können.

Ein weiterer Aspekt, dem mit dem Einsatz von Multi-Core-Prozessoren eine entscheidende Rolle zukommt, ist die Wiederverwendung existierender Software. In der Vergangenheit konnten bestehende Programme ohne Anpassungen von steigenden Taktfrequenzen und architektonischen Verbesserungen der Hardware profitieren. Da die meisten Programme jedoch sequentiell sind, können sie nicht von der Rechenleistung moderner Multi-Core-Prozessoren profitieren. Dies erfordert eine in der Regel aufwändige Refaktorisierung, die mit hohen Kosten verbunden ist. Aus industrieller Perspektive sollte der Übergang von Single-Core- zu Multi-Core-Prozessoren die Wiederverwendung bestehender Software unterstützen und so wenige Änderungen wie möglich nach sich ziehen.

Zur Lösung dieser Probleme wurde ein Konzept für die Entwicklung paralleler Embedded Systems entwickelt. Es bildet den Kern des Deliverables ZP-D5.1.A und dient als Grundlage für die Deliverables ZP-D5.2.C (Migrationspfade zur Parallelisierung bestehender Anwendungen), ZP-D5.2.D (Erfahrungen zur globalen Scheduling-Analyse) und ZP-D5.3.C (Konzepte und Experimente zur Vorhersage von W/B/A Case Execution Times für ausgewählte Parallelarchitekturen).

1.2 Management Summary

Das im Rahmen der ZP-Subtask 5.1.1 entwickelte Konzept basiert auf dem Einsatz von Koordinierungssprachen, die eine Entkopplung von den eigentlichen Berechnungen eines Systems und der Kommunikation zwischen den Modulen des Systems vorsehen. Auf diese Weise lässt sich bestehende Software wiederverwenden, wobei die parallele Ausführung der Module durch die Koordinierungssprache geregelt ist. Um die praktische Umsetzbarkeit dieses Ansatzes unter Beweis zu stellen, wurde eine an die Programmiersprache C/C++ angelehnte Koordinierungssprache definiert, die sich gut für die Verarbeitung kontinuierlicher Datenströme, beispielsweise in der digitalen Signalverarbeitung, eignet.

Außerdem wurden Algorithmen für die Übersetzung von Programmen der Koordinierungssprache in datenflussgesteuerte Prozessnetzwerke entworfen. Ein wesentlicher Vorteil von Prozessnetzwerken ist ihr Determinismus, der – wie bereits Abschnitt 1.1 erwähnt – die Sicherstellung der Korrektheit eines Systems erleichtert und eine Vorhersagbarkeit des Verhaltens erlaubt. In vielen Fällen entstehen jedoch bei der parallelen Ausführung bestehende Programmteile Konflikte beim Zugriff auf gemeinsame Ressourcen. Zur Vermeidung solcher Konflikte wurde ein Mechanismus eingeführt, der unter Berücksichtigung der Abhängigkeiten zwischen den betroffenen Programmteilen gewährleistet, dass zu keinem Zeitpunkt auf eine Ressource mehrfach schreibend zugegriffen wird. Die Übersetzung in Prozessnetzwerke stellt zudem sicher, dass im Falle eines Konflikts die betroffenen Programmteile immer in einer vorgegebenen Reihenfolge ausgeführt werden.

Die Methoden und Algorithmen wurden implementiert und anhand einiger Beispiele aus der industriellen Praxis getestet. Die Implementierung umfasst neben einem Compiler ein Laufzeitsystem für das Scheduling und die Speicherverwaltung. Die experimentellen Ergebnisse zeigen, dass die entwickelten Verfahren so effizient sind wie eine manuelle Parallelisierung mit Hilfe geeigneter Bibliotheken. Durch den Einsatz von Koordinierungssprachen ist der Aufwand für die Parallelisierung wesentlich geringer, wodurch Kosten reduziert und die Wartbarkeit erhöht werden.

1.3 Überblick

Eine detaillierte Beschreibung der Konzepte, Verfahren und experimentellen Ergebnisse findet sich in [1] (siehe Anlage zu diesem Dokument). Aus diesem Grund wird an dieser Stelle auf eine tiefergehende Behandlung verzichtet.

2 Literaturverzeichnis

[1] Tobias Schüle, “A Coordination Language for Programming Embedded Multi-Core Systems“, in *International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*. Hiroshima, Japan: IEEE Computer Society, 2009.

A Coordination Language for Programming Embedded Multi-Core Systems

Tobias Schuele
Corporate Technology, Software Architecture and Platforms (GTF SA&P)
Siemens AG
Munich, Germany
tobias.schuele@siemens.com

***Abstract*—We propose an implicitly parallel language for the coordination of sequential processes and present a translation scheme to dataflow process networks. Our language is well-suited for programming embedded systems and allows to benefit from the computational power offered by modern multi-core processors. Moreover, it facilitates the reuse of legacy code and can deal with side effects that may arise between functions written in the guest language by taking into account dependencies due to shared resources.**

I. Introduction

With the emergence of multi-core processors, parallel programming is becoming mainstream. However, this raises several challenges for software engineering. Firstly, programming with threads, which are still widely used in parallel programming, is an intricate and error-prone task [1]. Typical pitfalls in thread-based programming are violations of atomicity constraints and deadlocks (see [2] for a comprehensive study on real-world concurrency bugs). Due to the nondeterministic nature of threads, such bugs are hard to find by means of testing and might remain undetected for a long time [1]. For this reason, deterministic programming models have attained considerable interest in embedded system design, where high demands are put on correctness and reliability. Furthermore, predictability of nonfunctional aspects such as the worst-case execution time is crucial in embedded systems, since they are usually subject to real-time constraints [3].

Secondly, the shift from single-core to multi-core processors has significant impact on software reuse. In the past decades, software could benefit from steadily increasing processor clock rates and architectural improvements of the hardware without having to be rewritten. However, as most software is highly sequential, it is not able to utilize multiples processor cores. To benefit from the computational power of multi-core and future many-core processors, existing software needs to be parallelized, which requires extensive refactoring at significant costs [4]. From an industrial perspective, the migration of software from single-core to multi-core processors should therefore be as smooth as possible. Additionally, the resulting code should be easy to maintain and independent from the underlying hardware architecture, in particular the number of processor cores, to ensure scalability.

A promising approach to deal with these problems is the use of coordination languages [5], [6]. In the coordination paradigm, programming parallel or distributed systems is seen as the combination of two distinct activities: the computation part and the coordination part. The computation part comprises a number of single-threaded processes involved in manipulating data, whereas the coordination part is responsible for the communication and cooperation between the processes. Hence, coordination is the glue that binds separate activities into an ensemble [5]. There are a number of criteria for classifying coordination languages. For example, both parts might be integrated into a single language, or they might be separated into two distinct languages. Moreover, one distinguishes between data-driven and control-driven coordination languages [6].

This work has been funded by the German Federal Ministry of Education and Research (BMBF) as part of the alliance project “Software Platform Embedded Systems (SPES) 2020”.

In this paper, we present a data-driven coordination language called FluenC for programs written in C/C++. FluenC is well-suited for applications that process continuous streams of data, e.g. digital signal processing applications in embedded systems. It facilitates the reuse of legacy code and reduces the effort for refactoring. Even though we separate the coordination part from the computation part, the effort required to deal with two languages is low, since both languages have a similar syntax. In contrast to C/C++, however, FluenC is a dataflow language with imperative constructs such as loops (see [7] for a survey on dataflow programming languages). As a major advantage of the dataflow model of computation [8], parallelism is implicitly given and – except for side effects between C/C++ functions – only limited by the data dependencies inherent to a program.

Programs in our language are translated to dataflow process networks [9] that have a deterministic semantics, provided that the called C/C++ functions do not have any side effects. However, the restriction to pure (referentially transparent) functions is often neither necessary nor possible without rewriting large parts of a given program. Consider, for example, an application that writes several records to two distinct files. We certainly want the records to be written in the correct order to each file, but we might not be interested in the order in which the files are accessed. FluenC allows the programmer to deal with side effects by taking into account dependencies due to shared resources. The translation to dataflow process networks ensures that functions accessing a shared resource are always executed in a predefined order. In this way, the number of possible schedules is reduced, which simplifies testing and debugging.

An approach closely related to our work is Granular Lucid (GLU), a coarse-grain dataflow language for programming conventional parallel computers [10]. GLU supports the integration of functions and data types from foreign languages such as C, but lacks mechanisms to deal with side effects. StreamIt is a language and compilation infrastructure designed for large streaming applications [11]. In contrast to our approach, applications are described in a structural manner and functions are assumed to be side effect free. The Ptolemy system is a framework for the design of embedded real-time systems [12]. It supports a variety of computation models, including different forms of process networks, but focuses on modeling and simulation.

The rest of this paper is organized as follows: In the next section, we present the basic concepts of FluenC. Then, we show how FluenC programs can be translated to dataflow process networks (Section III). In Section IV, we present experimental results for the execution of the resulting networks on multi-core processors. Finally, we conclude with a summary and directions for future work in Section V.

II. The FluenC Language

A. An Introductory Example

To demonstrate the basic concepts and language constructs of FluenC, let us consider a simple example. Suppose, we want to compute the sum of two C functions `f` and `g` that are applied to a stream of integers. The values are read from the input stream and written to the output stream using the C functions `read` and `write`, respectively. Fig. 1 shows an implementation in FluenC. First, we declare the external functions using the **`extern`** keyword. As in C, the code to be executed upon startup is contained in the function `main`. However, due to the functional character of FluenC, the body of a function has to be an expression instead of a sequence of statements. In our example, the function `write` is applied to the result of `f(x)+g(x)`, where `x` is the value obtained by the call to the function `read`. As usual in dataflow and functional languages, assignment of values to variables is accomplished using **`let`** expressions.

Since `f` and `g` are independent of each other, they can be executed in parallel, which reduces the *latency* of the system. Additionally, the functions can be executed in a pipelined fashion, which increases the *throughput*. Since pipelining is a frequently encountered pattern in stream-based programs, FluenC provides an operator for describing pipelines in a more intuitive way than using function application. Using this operator, the body of `main` might be written as follows:

```
(let x=read() in f(x)+g(x)) |> write;
```

The pipeline operator reflects the order in which pipelines are usually depicted, namely from left to right.

Suppose that the execution time of `g` is much higher then the execution time of the other functions and the + operator. In other words, suppose that the execution time of `g` limits the throughput. What can we do to increase the throughput? If `g` is referentially transparent, i.e., if `g` neither maintains a state nor has any side effects, multiple instances of `g` might be executed in parallel. For that purpose, we may put the keyword **`parallel`** in front of the function declaration (or a block of function declarations). This tells the compiler that it might execute multiple instances of the function(s) in parallel in order to increase the throughput.

B. Resources

In the previous example we assumed that the input stream and the output stream are independent of each other. Hence, the functions `read` and `write` do not interfere and can be executed in parallel. In certain cases, however, several functions might share a resource, which requires synchronization to avoid race conditions. FluenC does not provide any explicit synchronization mechanisms such as semaphores or monitors. Conflicts are resolved at compile time by taking into account data dependencies, the evaluation order of an expression, and the resources a function accesses (reading or writing).

We do not make any assumptions about the physical nature or the implementation of a resource. Resources in FluenC are abstract entities that may represent various kinds of objects, e.g. peripheral devices, files, communication channels, or shared memory areas. A resource may even subsume several objects. As an example, Fig. 2 depicts a feedback control system that continuously reads data from a sensor, processes the data, and controls an actuator. These three steps are implemented by the functions `read`, `process`, and `write`, respectively. Suppose that we do not want a new value to be read from the sensor before the previous one has been processed and the actuator has been adjusted. Then, a new iteration of the control loop must not be started before the previous one is completed. This can be achieved by modeling the sensor and the actuator as well as the environment as a single resource R. In addition, we have to specify the resources accessed by the functions `read` and `write`:

```
resource R;
double read() reads R;
void write(double) writes R;
```

The translation of FluenC programs to dataflow process networks guarantees that functions accessing a shared resource are executed in a round-robin manner. Hence, a new sensor value will not be read until the previous one has been processed.

```
extern "C" {
  int read();
  void write(int);
  int f(int);
  int g(int);
}

void main() {
  write(let x=read() in f(x)+g(x));
}
```

Figure 1. FluenC program that processes a stream of integers

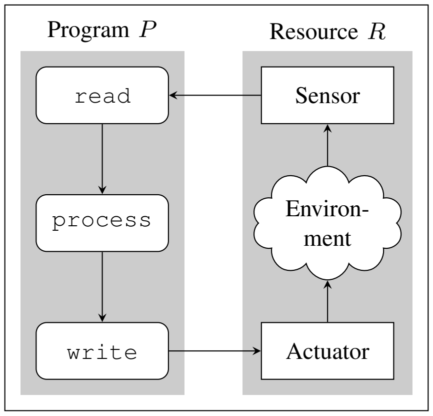


Figure 2. System consisting of a program and an associated resource

In the sequel, we will refer to a set of functions and the resources they access as a *mesh*, which is defined as follows:

Definition 1: A mesh is a tuple $(\mathcal{R}, \mathcal{F}, \mathsf{par}, \mathsf{ar}, \mathsf{rd}, \mathsf{wr})$, where

- $\mathcal{R}$ is a finite set of resources,
- $\mathcal{F}$ is a finite set of functions,
- $\mathsf{par}\colon \mathcal{F} \to \mathbb{B}$ associates with every function a Boolean value that specifies whether multiple instances of the function can be executed in parallel,
- $\mathsf{ar}\colon \mathcal{F} \to \mathbb{N}$ associates with every function its arity,
- $\mathsf{rd}\colon \mathcal{F} \to 2^{\mathcal{R}}$ associates with every function the set of resources from which the function reads, and
- $\mathsf{wr}\colon \mathcal{F} \to 2^{\mathcal{R}}$ associates with every function the set of resources to which the function writes.

C. Synchronization

In contrast to explicit locking mechanisms, which can be applied to small critical sections, synchronization in FluenC is performed at function level. On the one hand, this may lead to a waste of parallelism, since not all parts of a function may actually depend on a resource. On the other hand, fine-grained synchronization mechanisms such as mutexes are often rather inefficient, since they cause a time consuming context switch by the operating system in case a thread blocks. For this reason, many libraries and tools like Intel's Threading Building Blocks (TBB)[1] [13] and OpenMP[2] [14], [15] try to avoid locks and perform synchronization at the level of lightweight tasks. FluenC follows a similar approach: Instead of immediately scheduling a function, which may block soon after due to missing data or busy resources, it is not scheduled until all of its parameters are available and all resources on which the function depends are free.

As mentioned previously, the evaluation order of expressions is used to resolve conflicts between functions that access one or more shared resources. In case of a conflict, the involved subexpressions are evaluated one after the other in the order of their occurrence (from left to right). Otherwise, they may be executed in parallel. Consider, for example, the expression `f()+g()+g()+f()+h()` and suppose that `f` writes to a resource, which is read by `g` (`h` does not access the resource). Then, both calls to `g` are delayed until the first call to `f` is completed, and the second call to `f` is delayed until both calls to `g` are completed. Since `g` only reads from the resource, both calls to `g` can be executed in parallel. Moreover, `h` can be executed in parallel to both `f` and `g`. Fig. 3 shows a possible schedule on dual-core processor.

[1]`http://www.threadingbuildingblocks.org`
[2]`http://www.openmp.org`

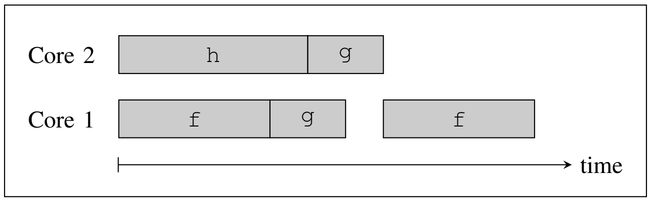


Figure 3. Schedule on a dual-core processor

D. Expressions

In the following, we will describe the syntax and semantics of expressions in FluenC. For the sake of simplicity, we will restrict ourselves to a subset, which contains the most important constructs.[3]

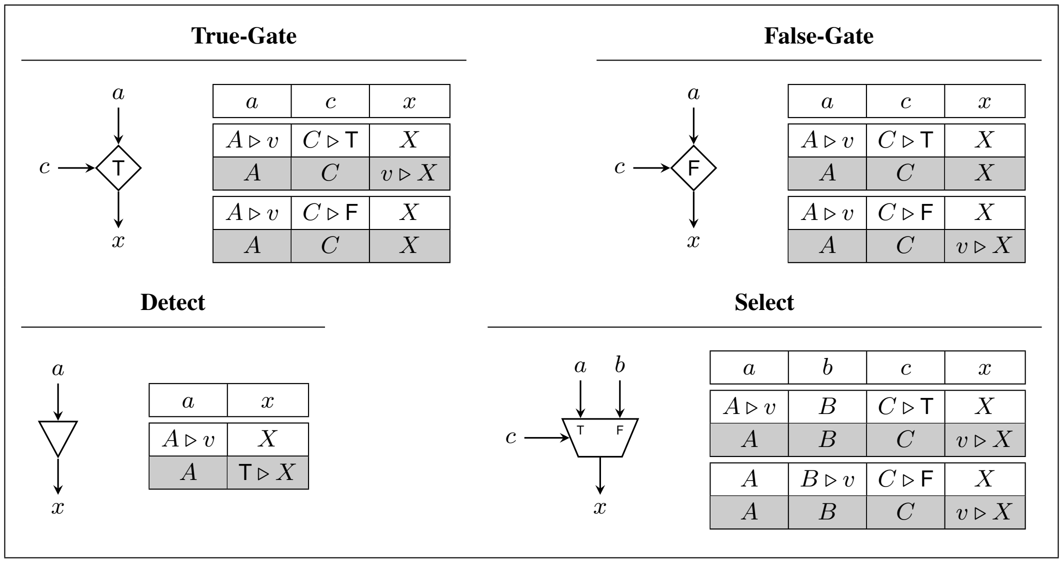
Definition 2: Given a mesh $M = (\mathcal{R}, \mathcal{F}, \mathsf{par}, \mathsf{ar}, \mathsf{rd}, \mathsf{wr})$ and a set of variables $\mathcal{V}$, the set of expressions Expr is the least set that satisfies the following rules:

- $\{\texttt{true}, \texttt{false}\} \subseteq \mathsf{Expr}$
- $\mathbb{Z} \subseteq \mathsf{Expr}$
- $\mathcal{V} \subseteq \mathsf{Expr}$
- $e \;\texttt{|>}\; f \in \mathsf{Expr}$, provided that $e \in \mathsf{Expr}$, $f \in \mathcal{F}$, and $\mathsf{ar}(f) = 1$
- $f(e_0, \ldots, e_{n-1}) \in \mathsf{Expr}$, provided that $f \in \mathcal{F}$, $\mathsf{ar}(f) = n$, and $e_i \in \mathsf{Expr}$ for all $0 \leq i < n$
- $\texttt{!}\,e \in \mathsf{Expr}$, provided that $e \in \mathsf{Expr}$
- $e_0 \diamond e_1 \in \mathsf{Expr}$, provided that $e_0, e_1 \in \mathsf{Expr}$ and $\diamond \in \{\texttt{\&\&}, \texttt{||}\}$
- $e_0 \circ e_1 \in \mathsf{Expr}$, provided that $e_0, e_1 \in \mathsf{Expr}$ and $\circ \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{\%}\}$
- $e_0 \bowtie e_1 \in \mathsf{Expr}$, provided that $e_0, e_1 \in \mathsf{Expr}$ and $\bowtie \in \{\texttt{==}, \texttt{!=}, \texttt{<}, \texttt{>}, \texttt{<=}, \texttt{>=}\}$
- $e\texttt{@} \in \mathsf{Expr}$, provided that $e \in \mathsf{Expr}$
- $e_0\texttt{,}\, e_1 \in \mathsf{Expr}$, provided that $e_0, e_1 \in \mathsf{Expr}$
- $e_0 \;\texttt{?}\; e_1 \;\texttt{:}\; e_2 \in \mathsf{Expr}$, provided that $e_0, e_1, e_2 \in \mathsf{Expr}$
- $\textbf{\texttt{if}}\ (e_0)\ e_1\ \textbf{\texttt{else}}\ e_2 \in \mathsf{Expr}$, provided that $e_0, e_1, e_2 \in \mathsf{Expr}$
- $\textbf{\texttt{while}}\ (e_0)\ \{\textbf{\texttt{next}}(v_1) = e_1;\ \ldots;\ \textbf{\texttt{next}}(v_{n-1}) = e_{n-1};\}\ \textbf{\texttt{finally}}\ e_n \in \mathsf{Expr}$, provided that $e_0, \ldots, e_n \in \mathsf{Expr}$ and $v_1, \ldots, v_{n-1} \in \mathcal{V}$
- $\textbf{\texttt{let}}\ \{v_0 = e_0;\ \ldots;\ v_{n-1} = e_{n-1};\}\ \textbf{\texttt{in}}\ e_n \in \mathsf{Expr}$, provided that $e_0, \ldots, e_n \in \mathsf{Expr}$ and $v_0, \ldots, v_{n-1} \in \mathcal{V}$

The pipeline operator `|>` was already described in Section II-A. The Boolean operators `&&` and `||` have the same meaning as in C: The second operand is only evaluated if the first operand does not already determine the result. The `@` operator, which is not available in C, serves as a barrier. Any functions following a barrier are not executed until all functions preceding the barrier are completed. For example, let `f`, `g`, and `h` be functions that do not share any resources and may thus be executed in parallel. Given the expression `(f()+g())@+h()`, the execution of `h` is delayed until both `f` and `g` are completed. Barriers are particularly useful for debugging, since they allow the programmer to control the execution order at a more fine-grained level than using resources.

The comma operator has the same meaning as in C, i.e., both operands are evaluated, but only the result of the right operand is returned. Hence, $e_0\texttt{,}\,e_1$ is equivalent to e_1, provided that e_0 does not have any side effects. The value of a conditional expression $e_0 \;\texttt{?}\; e_1 \;\texttt{:}\; e_2$ is equal to the value of e_1 if e_0 evaluates to true. Otherwise, it is equal to the value of e_2. The conditional operator is eager in the sense that both e_1 and e_2 are evaluated

[3]Except for pointers, FluenC supports most of the language constructs available in C, including arrays, structures, enumerations, different kinds of loops, bitvector operations, compound literals, etc.



True-Gate

a	c	x
$A \triangleright v$	$C \triangleright \mathsf{T}$	X
A	C	$v \triangleright X$
$A \triangleright v$	$C \triangleright \mathsf{F}$	X
A	C	X

False-Gate

a	c	x
$A \triangleright v$	$C \triangleright \mathsf{T}$	X
A	C	X
$A \triangleright v$	$C \triangleright \mathsf{F}$	X
A	C	$v \triangleright X$

Detect

a	x
$A \triangleright v$	X
A	$\mathsf{T} \triangleright X$

Select

a	b	c	x
$A \triangleright v$	B	$C \triangleright \mathsf{T}$	X
A	B	C	$v \triangleright X$
A	$B \triangleright v$	$C \triangleright \mathsf{F}$	X
A	B	C	$v \triangleright X$

Figure 4. Actors and their semantics for controlling the dataflow in a process network

(the unused result is discarded). The **`if`** construct can be viewed as a lazy variant of the conditional operator: e_1 is only evaluated if e_0 holds, and e_2 is only evaluated if e_0 does not hold. The syntax of the **`while`** loop was adopted from the dataflow language Id [16]. The loop body consists of a series of statements, where each binding occurrence of an identifier v_i is prefixed by the keyword **`next`**, denoting the value of v_i in the next iteration. Execution of the loop body terminates when the loop condition e_0 evaluates to false, in which case the value of e_n is returned. Finally, the **`let`** statement binds a series of values $e_0, \ldots, e_{n-1}$ to a series of variables $v_0, \ldots, v_{n-1}$ and returns the value of e_n.

Definition 3: A program is a triple $(e, M, \mathcal{V})$, where $e \in \mathsf{Expr}$ is an expression over a mesh M and a set of variables $\mathcal{V}$.

The presented subset of FluenC contains some syntactic sugar. In order to simplify the translation of FluenC programs to dataflow process networks, we make use of the following equivalences:

- e `|>` $f \equiv f(e)$
- e_0 `&&` $e_1 \equiv$ **`if`** (e_0) e_1 **`else false`**
- e_0 `||` $e_1 \equiv$ **`if`** (e_0) **`true else`** e_1
- e_0`,`$e_1 \equiv$ **`false`** `?` e_0 `:` e_1

According to these equivalences, every program can be easily translated to a program that neither contains the pipeline operator, the Boolean operators `&&` and `||`, nor the comma operator.

III. Translation of FluenC Programs to Dataflow Process Networks

A. Foundations

Dataflow process networks (DPNs) [9] are a special case of Kahn process networks (KPNs) [17]. Process networks have found many applications, e.g. in digital signal processing and distributed computing. A KPN consists of a set of (sequential) processes that communicate with each other through unidirectional FIFO channels. Processes read and write data elements (tokens) from and to the channels, where each element is read (written) exactly once. KPNs are deterministic under the following conditions [9], [17]: writing to a channel is nonblocking, while reading from a channel is blocking, i.e., a process that reads from an empty channel will stall and can only continue when the channel contains at least one token. Moreover, processes are not allowed to test an input channel for emptiness, and each process must be deterministic. KPNs lend themselves well for modeling parallel systems, since their behavior does not depend on computation or communication delays.

However, the blocking read condition may cause considerable overhead for context switching in case a process wants to read from an empty input channel. DPNs avoid this overhead by replacing processes with actors that are executed (fire) according to a predefined set of rules [9]. In general, an actor fires iff each input channel contains at least one token. Fig. 4 shows some more complex actors, which are used to control the dataflow in a DPN (see [18]–[20] for a description of different kinds of actors in dataflow computing). The semantics of the actors is given by the associated tables, where nonshaded (shaded) lines represent the contents of the input and output channels before (after) firing. An entry $A \triangleright v$ $(v \triangleright X)$ denotes concatenation of a possibly empty sequence A (X) and a value v. Intuitively, the true-gate (false-gate) passes the token at input a to output x if c contains a token with the Boolean value true (false). Otherwise, the token is discarded. The detect actor produces for each incoming token a new token carrying the value true. Depending on the token at input c, the select actor passes a token either from input a or input b to output x (no token is consumed from the other input).

B. Dependency Analysis

For the translation of FluenC programs to DPNs, we have to take into account dependencies between functions that access one or more shared resources. For that purpose, we first perform a dependency analysis that determines which functions can be executed in parallel. The obtained information is then used in a second step for the construction of the DPN. The dependency analysis is based on combined control/dataflow graphs (CDFGs), which represent data dependencies between function calls and their execution order in case of conflicts.

Definition 4: A CDFG is a tuple $(N, N_B, \mathsf{fn}, E_C, E_D)$, where

- N is a set of nodes,
- $N_B \subseteq N$ is a set of nodes that immediately precede a barrier,
- $\mathsf{fn}\colon N \to \mathcal{F}$ associates with every node a function from a given set $\mathcal{F}$,
- $E_C \subseteq N \times N$ is a set of control flow edges, and
- $E_D \subseteq N \times N$ is a set of dataflow edges.

The set of predecessors of a node $n \in N$ w.r.t. a set of edges E is denoted by $\mathsf{pre}(n, E)$. Two nodes $n_0, n_k \in N$ are data-dependent, written $n_0 \rightsquigarrow n_k$, iff there exists a sequence of nodes $n_0, \ldots, n_k$ such that $(n_i, n_{i+1}) \in E_D$ for all $0 \leq i < k$.

The construction of a CDFG for a FluenC program is accomplished according to the templates shown in Figs. 5 and 6, where dataflow edges are represented by solid lines and control flow edges by dashed lines (a line leaving a gate or a select actor at the right-hand side is a copy of the control input).[4] The first template shows that control is fed back from the last function of a program to the first one, which reflects the fact that we consider nonterminating systems. For a function application $f(e_0, \ldots, e_{n-1})$, the arguments are evaluated from left to right, i.e., starting with e_0, before f is called. The template for a barrier expression `e@` indicates that the most recently encountered functions are added to the set N_B. Note that for an expression **`if`** (e_0) e_1 **`else`** e_2 the control flow forks after e_0, meaning that either e_1 or e_2 is executed, but not both. In case of a **`while`** loop, control is fed back after the last **`next`** statement to the loop condition (the circles to the left of the select actors indicate that the FIFO channels associated with the control inputs initially contain the value false). As an example, Fig. 7 shows a simple FluenC program and the corresponding CDFG.

The next step is to determine which functions can be executed in parallel for a given mesh $M = (\mathcal{R}, \mathcal{F}, \mathsf{par}, \mathsf{ar}, \mathsf{rd}, \mathsf{wr})$. To deal with barriers and functions that cannot be executed in parallel, we modify $\mathcal{R}$ as follows: For each $f \in \mathcal{F}$ such that $\mathsf{par}(f)$ does not hold, we create a resource r_f, which is written by f. Additionally, we create a resource r_b, which is read by every function $f \in \mathcal{F}$ and written whenever a barrier is encountered. Hence, all functions preceding a barrier must be completed before execution continues after the barrier. To sum up, we obtain a mesh $M' = (\mathcal{R}', \mathcal{F}, \mathsf{par}, \mathsf{ar}, \mathsf{rd}', \mathsf{wr}')$ such that the following holds:

$$\begin{aligned} \mathcal{R}' &= \mathcal{R} \cup \{r_f \mid f \in \mathcal{F} \land \neg\mathsf{par}(f)\} \cup \{r_b\} \\ \mathsf{rd}'(f) &= \mathsf{rd}(f) \cup \{r_b\} \\ \mathsf{wr}'(f) &= \begin{cases} \mathsf{wr}(f) \cup \{r_f\} & \text{if } \neg\mathsf{par}(f) \\ \mathsf{wr}(f) & \text{otherwise} \end{cases} \end{aligned}$$

For the computation of the dependencies, we employ a variant of the *gen-kill* framework [21], which is used for various kinds of analyses in compilers. In this framework, the desired information is obtained by setting up and solving a system of equations, each consisting of a *generate* and a *kill* part (in addition to an input part, which collects information from related equations). To distinguish between read and write accesses, we split the generate part into a read and a write part. Then, we obtain for each node $n \in N$ of a CDFG $(N, N_B, \mathsf{fn}, E_C, E_D)$ an equation of the following form:

$$\mathsf{OUT}(n) = ((\mathsf{IN}_C(n) \cup \mathsf{RD}(n)) \setminus \mathsf{KILL}(n)) \cup \mathsf{WR}(n)$$

The solution to such an equation is a subset of the set $\mathcal{D} = \mathcal{R}' \times \mathbb{B} \times N$. A triple $(r, b, n) \in \mathcal{D}$ means that resource r was last read ($b = \mathsf{false}$) or written ($b = \mathsf{true}$) by node n.

Let us first consider the input part. Given a node $n \in N$, the set of incoming triples $\mathsf{IN}_C(n)$ depends on the predecessors of n w.r.t. the set of control edges E_C:

$$\mathsf{IN}_C(n) = \bigcup\nolimits_{n' \in \mathsf{pre}(n, E_C)} \mathsf{OUT}(n')$$

The generate part consists of the functions RD and WR that specify which resources are read and written, respectively:

$$\begin{aligned} \mathsf{RD}(n) &= \{(r, \mathsf{false}, n) \in \mathcal{D} \mid r \in \mathsf{rd}'(\mathsf{fn}(n))\} \\ \mathsf{WR}(n) &= \{(r, \mathsf{true}, n) \in \mathcal{D} \mid r \in \mathsf{wr}'(\mathsf{fn}(n)) \lor \\ &\qquad\qquad\qquad\qquad r = r_b \land n \in N_B\} \end{aligned}$$

Note that $\mathsf{WR}(n)$ contains the barrier resource r_b if node n serves as a barrier, i.e., if $n \in N_B$. In this case, there is a dependency between n and the nodes succeeding n, since all functions read from r_b. Finally, KILL is used to remove all triples whose resources are overwritten by the function associated with n:

$$\mathsf{KILL}(n) = \{(r, \_, \_) \in \mathcal{D} \mid r \in \mathsf{wr}'(\mathsf{fn}(n))\}$$

Since we are interested in the minimal set of dependencies, we compute the least solution to the equation system. This can be done in polynomial time by iteratively evaluating $\mathsf{OUT}(n)$ until a fixpoint is reached, starting with $\mathsf{OUT}(n) = \{\}$ for all $n \in N$ [21]. Termination of the fixpoint iteration is guaranteed by the fact that $\mathsf{OUT}(n)$ does not occur complemented in any of the equations.

It remains to show how the solution to an equation system is used to determine which functions depend on each other. In dataflow analysis, one distinguishes between three types of dependencies:

- true dependencies (read-after-write, RAW)
- anti-dependencies (write-after-read, WAR)
- output dependencies (write-after-write, WAW)

The latter two types of dependencies are also referred to as false dependencies and can be subsumed by a single type of dependency, which we denote by WAX. The following definitions give us for each node $n \in N$ the set of nodes on which n depends:

$$\begin{aligned} \mathsf{RAW}(n) &= \{n' \in N \mid \exists r \in \mathsf{rd}'(\mathsf{fn}(n)).(r, \mathsf{true}, n') \in \mathsf{IN}_C(n)\} \\ \mathsf{WAX}(n) &= \{n' \in N \mid \exists r \in \mathsf{wr}'(\mathsf{fn}(n)).\exists b \in \mathbb{B}. \\ &\qquad (r, b, n') \in \mathsf{IN}_C(n)\} \end{aligned}$$

However, even if there is a dependency between two nodes n_0 and n_k according to the above definitions, they need not necessarily be synchronized. In many cases, we have $n_0 \rightsquigarrow n_k$ and hence, n_0 and n_k will never be executed in parallel. To reduce the effort for synchronization and thus the size of the resulting DPN, we have to take into account data dependencies. The equation system

$$\mathsf{DEP}(n) = \{n\} \cup \mathsf{IN}_D(n)$$

gives us for each node $n \in N$ the set of nodes on which n depends (including dependencies due to shared resources), where

$$\mathsf{IN}_D(n) = \bigcup\nolimits_{n' \in \mathsf{pre}(n, E_D)} \mathsf{DEP}(n').$$

[4]Since a CDFG only encodes the control and data dependencies between function calls, the translation abstracts from all other actors shown in the templates. These are later used for the construction of the DPN.

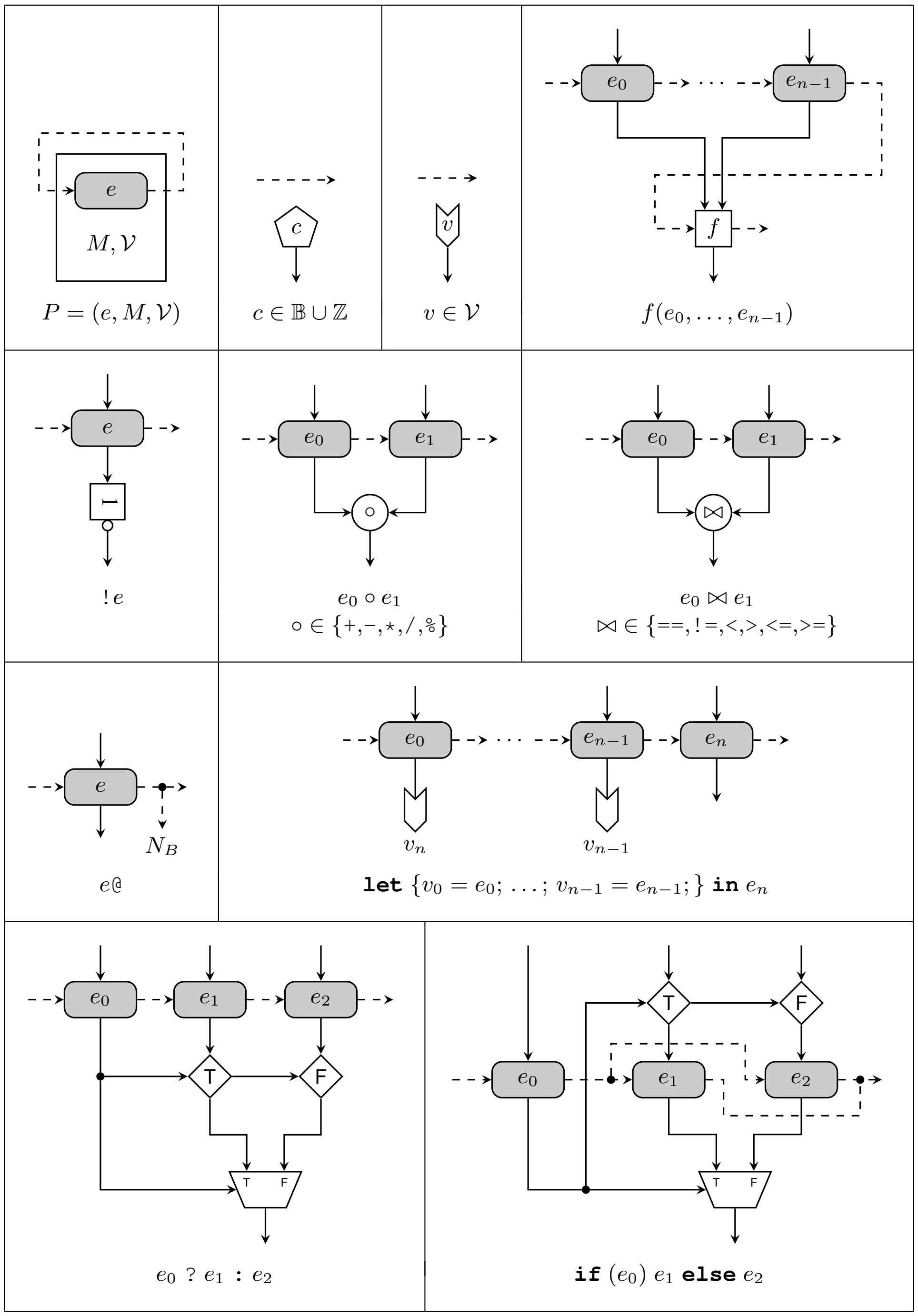
Thus, n_0 and n_k need only be synchronized if $n_0 \in \mathsf{SYNC}(n_k)$, where

$$\mathsf{SYNC}(n) = (\mathsf{RAW}(n) \cup \mathsf{WAX}(n)) \setminus \mathsf{IN}_D(n).$$

Table I shows the results of the dependency analysis for the CDFG depicted in Fig. 7 (the barrier resource r_b was omitted for better readability).

C. Construction of the Dataflow Process Network

The construction of the DPN is again based on the templates shown in Figs. 5 and 6. Additionally, we have to incorporate the results from the dependency analysis in case of function calls. Suppose, for example, that n is the node corresponding to a function call $f(e_0, \ldots, e_{n-1})$. If there exists a node n' such that $n' \in \mathsf{SYNC}(n)$, we add a detect actor to the output of the function associated with n' and extend f by a Boolean parameter e_n, which



e
$M, \mathcal{V}$
$P = (e, M, \mathcal{V})$
c
$c \in \mathbb{B} \cup \mathbb{Z}$
v
$v \in \mathcal{V}$
e_0
e_{n-1}
f
$f(e_0, \ldots, e_{n-1})$
$!e$
$e_0 \circ e_1$
$\circ \in \{+,-,\star,/,\%\}$
$e_0 \bowtie e_1$
$\bowtie \in \{==,!=,<,>,<=,>=\}$
N_B
$e@$
e_n
v_n
v_{n-1}
let $\{v_0 = e_0; \ldots; v_{n-1} = e_{n-1};\}$ **in** e_n
T
F
e_0 ? e_1 : e_2
if (e_0) e_1 **else** e_2

Figure 5. Templates for the translation of expressions to dataflow process networks

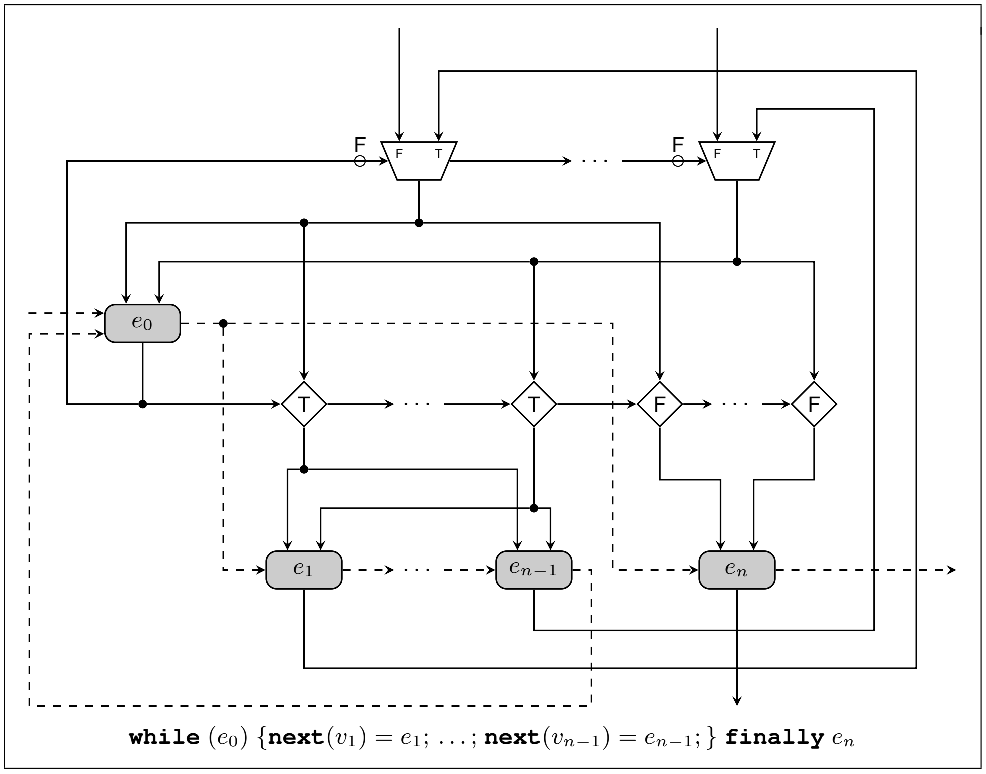

F
F
T
F
F
T
e_0
T
T
F
F
e_1
e_{n-1}
e_n
while (e_0) $\{$**next**$(v_1) = e_1; \ldots;$ **next**$(v_{n-1}) = e_{n-1};\}$ **finally** e_n

Figure 6. Template for the translation of loops to dataflow process networks

```
resource R;

extern parallel "C" {
  int get(void) reads R writes R;
  int f(int) reads R;
  int g(int) writes R;
  int h(void) reads R;
  void put(int) writes R;
}

void main() {
  ((let {x=get(); y=get();} in
    if (x!=0) f(x) else g(y))+h())
  |> put;
}
```

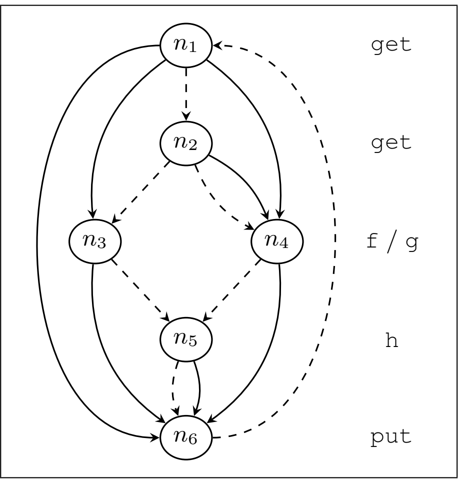


Figure 7. A FluenC program and the corresponding control/dataflow graph

Table I
RESULTS OF THE DEPENDENCY ANALYSIS FOR THE CONTROL/DATAFLOW GRAPH SHOWN IN FIGURE 7

n	$\mathsf{OUT}(n)$	$\mathsf{RAW}(n)$	$\mathsf{WAX}(n)$	$\mathsf{DEP}(n)$	$\mathsf{SYNC}(n)$
n_1	$\{(R, \mathsf{true}, n_1)\}$	$\{n_6\}$	$\{n_6\}$	$\{n_1\}$	$\{n_6\}$
n_2	$\{(R, \mathsf{true}, n_2)\}$	$\{n_1\}$	$\{n_1\}$	$\{n_2\}$	$\{n_1\}$
n_3	$\{(R, \mathsf{true}, n_2), (R, \mathsf{false}, n_3)\}$	$\{n_2\}$	$\{\}$	$\{n_1, n_3\}$	$\{n_2\}$
n_4	$\{(R, \mathsf{true}, n_4)\}$	$\{\}$	$\{n_2\}$	$\{n_1, n_2, n_4\}$	$\{\}$
n_5	$\{(R, \mathsf{true}, n_2), (R, \mathsf{false}, n_3), (R, \mathsf{true}, n_4), (R, \mathsf{false}, n_5)\}$	$\{n_2, n_4\}$	$\{\}$	$\{n_5\}$	$\{n_2, n_4\}$
n_6	$\{(R, \mathsf{true}, n_6)\}$	$\{\}$	$\{n_2, n_3, n_4, n_5\}$	$\{n_1, n_2, n_3, n_4, n_5, n_6\}$	$\{\}$

is connected to the output of the detect actor.[5] In this way, it is guaranteed that f is not executed until the function associated with n' is completed (e_n need not be evaluated by f — it simply serves as a trigger for the execution of f). If n depends on multiple nodes, we create for each node a detect actor and compute the conjunction of their outputs. Note that the output of a detect actor (or a conjunction) is treated as an ordinary value. This means that it must pass a gate (select) actor if it enters (leaves) a branch of an **`if`** expression. Otherwise, the DPN may block or compute wrong results due to superfluous tokens. Dependencies entering or leaving a **`while`** loop are treated analogously. As an example for the construction of a DPN, Fig. 8 shows the DPN for the program of Fig. 7 (shaded actors are used for synchronization).

To run a FluenC program on a multi-core processor, the resulting DPN can either be interpreted at runtime or used for code generation. As can be seen from Fig. 8, the degree of parallelism may thereby depend on the input data: If the result of the comparison operation is true, `f` and `h` may be executed in parallel. Otherwise, `h` must wait until `g` is completed. If a function has been declared to be

[5]If the function associated with n' does not return a result, i.e., if it is declared to be **`void`**, we extend it to return dummy value.

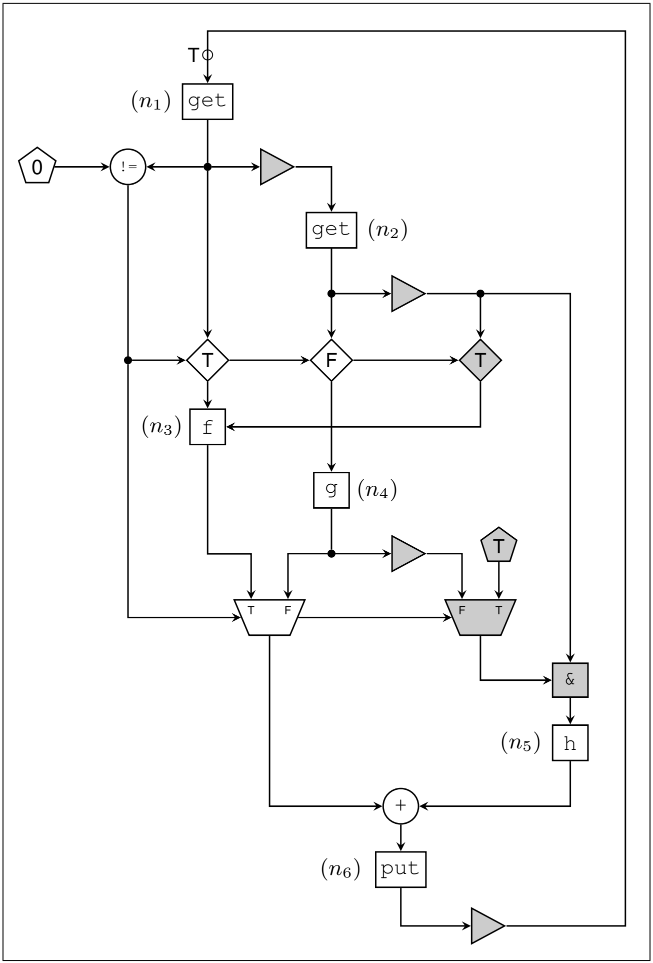


Figure 8. Dataflow process network for the program shown in Figure 7

parallel, multiple calls may be executed simultaneously, provided that the incoming FIFO channels contain sufficient data.

IV. Experimental Results

We have developed a compiler that translates FluenC programs to C/C++ code. The front-end of the compiler constructs a DPN, which is used by the back-end for code generation. Each actor is thereby mapped to a lightweight task, which is scheduled as soon as its operands are available. The generated code can be executed in a completely distributed manner, i.e., there are no global data structures that may result in a bottleneck.

Additionally, we have implemented a number of benchmarks to evaluate our approach. Besides the FluenC implementation, each benchmark has also been implemented using Intel's TBB. To obtain comparable results, the code generated by the FluenC compiler also uses the task scheduler provided by TBB. All experiments were performed on a system with two quad-core Xeon processors (2.83 GHz) and 6 GB RAM running Linux.

Table II shows the results. For each benchmark, we measured the speedup with respect to a sequential implementation. Additionally, we determined the efficiency (speedup divided by the number of cores). The first benchmark, *MergeSort*, sorts a 256 KB large array of characters. The results for *bzip2* were obtained by compressing a 256 MB large file. The next benchmark is an implementation of a video decoder according to the H.261 standard. Finally, *ImgRec* is part of an image recognition application developed at Siemens for industrial automation.

On a single core, the parallel implementations using TBB and FluenC are virtually as fast as the sequential ones, which indicates that the overhead for task management and scheduling is negligible. Note that the FluenC implementation of *MergeSort* scales much better than the one using TBB (98% vs. 66% efficiency on eight cores). This is due to the fact that the former implicitly exploits pipelining, whereas the latter is based on a divide and conquer algorithm. In principle, one could also make use of pipelining in the TBB implementation. However, since TBB only supports linear (string-like) pipelines, the tree-like structure of *MergeSort* has to be linearized, which complicates the implementation and increases the latency.[6] For *bzip2*, *H261*, and *ImgRec* both implementations perform comparably well.

[6]For the other benchmarks, linearization was either not necessary or possible with reasonable effort.

Table II
Experimental Results

Benchmark	Impl.	# Cores (Speedup/Efficiency [%])			
		1	2	4	8
MergeSort	TBB	1.00/100	1.90/ 95	3.35/ 84	5.31/ 66
	FluenC	0.99/ 99	1.97/ 99	3.93/ 98	7.83/ 98
bzip2	TBB	1.00/100	1.95/ 97	3.72/ 93	6.13/ 77
	FluenC	1.00/100	2.00/100	3.38/ 85	5.94/ 74
H261	TBB	0.98/ 98	1.95/ 97	3.85/ 96	5.53/ 69
	FluenC	1.00/100	1.96/ 98	3.88/ 97	5.67/ 71
ImgRec	TBB	1.00/100	1.97/ 99	3.83/ 96	6.59/ 82
	FluenC	1.00/100	1.97/ 99	3.83/ 95	6.80/ 85

Besides performance, implementation effort and maintainability play an important role in the development of parallel applications. Our experience has shown that the parallelization of C programs at task level using FluenC is often much easier compared to the use of libraries such as TBB, unless the system can be modeled as a single linear pipeline. One reason for this is that the FluenC compiler relieves the programmer from the burden of task management and lets him/her focus on architectural aspects. The amount of code for parallelization is thereby significantly reduced. For example, the FluenC implementation of *ImgRec* is about 40% smaller than the TBB implementation.

V. Summary and Conclusion

We presented a coordination language called FluenC for parallel programming of embedded multi-core systems. FluenC is based on the dataflow model of computation and lends itself well for stream-based applications. As a major advantage from an industrial perspective, it facilitates the reuse of sequential code: Functions written in C/C++ can be directly called from FluenC programs without the need for extensive refactoring. We believe that the similarity to C/C++ stimulates its acceptance, since it reduces the effort for becoming acquainted with a new language. Nevertheless, the concepts of FluenC are largely independent from the guest language and can also be applied to other languages such as Java.

FluenC programs can be translated to dataflow process networks that have a deterministic semantics, provided that the actors of a network are deterministic and do not share any global data. In certain cases, however, the requirement that an actor may operate on local data only is too strict, particularly in the presence of legacy code. FluenC enables the programmer to deal with external functions that have side effects using the notion of resources. Our translation procedure ensures that functions accessing the same resources are always executed in a predefined order, which simplifies testing and debugging.

FluenC primarily targets parallelism at task level. However, in order to utilize future processors with dozens or more cores, one has to exploit parallelism at all levels. We plan to extend FluenC by operations that leverage data-level parallelism, e.g. using the map-reduce pattern [22]. Moreover, we are working on techniques for analyzing the worst-case execution time of FluenC programs, which is required in the design of real-time systems to guarantee that certain timing constraints are met.

References

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

[2] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes – a comprehensive study on real world concurrency bug characteristics," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Seattle, WA, USA: ACM, March 2008, pp. 329–339.

[3] A. Jantsch and I. Sander, "Models of computation and languages for embedded system design," *IEE Proceedings*, vol. 152, no. 2, pp. 114–129, March 2005.

[4] V. Pankratius, C. Schaefer, A. Jannesari, and W. Tichy, "Software engineering for multicore systems – an experience report," in *International Workshop on Multicore Software Engineering*. Leipzig, Germany: ACM, 2008, pp. 53–60.

[5] D. Gelernter and N. Carriero, "Coordination languages and their significance," *Communications of the ACM*, vol. 35, no. 2, pp. 97–107, February 1992.

[6] G. Papadopoulos and F. Arbab, "Coordination models and languages," *Advances in Computers*, vol. 46, pp. 330–401, 1998.

[7] W. Johnston, J. Hanna, and R. Millar, "Advances in dataflow programming languages," *ACM Computing Surveys*, vol. 36, no. 1, pp. 1–34, March 2004.

[8] W. Najjar, E. Lee, and G. Gao, "Advances in the dataflow computational model," *Parallel Computing*, vol. 25, no. 13-14, pp. 1907–1929, December 1999.

[9] E. Lee and T. Parks, "Dataflow process networks," *Proceedings of the IEEE*, vol. 83, no. 5, pp. 773–801, May 1995.

[10] R. Jagannathan, "Coarse-grain dataflow programming of conventional parallel computers," in *Advanced Topics in Dataflow Computing and Multithreading*, G. Gao, L. Bic, and J.-L. Gaudiot, Eds. IEEE Computer Society, 1995, pp. 113–129.

[11] W. Thies, M. Karczmarek, and S. Amarasinghe, "StreamIt: A language for streaming applications," in *International Conference on Compiler Construction (CC)*, ser. LNCS, R. Horspool, Ed., vol. 2304. Grenoble, France: Springer, 2002, pp. 179–196.

[12] E. Lee, "Overview of the Ptolemy project," University of California at Berkeley, Tech. Rep. UCB/ERL M03/25, July 2003.

[13] J. Reinders, *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly, 2007.

[14] B. Chapman, G. Jost, and R. van der Pas, *Using OpenMP: Portable Shared Memory Parallel Programming*. MIT Press, 2008.

[15] *OpenMP Application Program Interface (Version 3.0)*, OpenMP Architecture Review Board, 2008.

[16] R. S. Nikhil, "ID Language Reference Manual (Version 90.1)," Massachusetts Institute of Technology, Computation Structures Group Memo 284-2, July 1991.

[17] G. Kahn, "The semantics of a simple language for parallel programming," in *Information Processing*, J. Rosenfeld, Ed. Stockholm, Sweden: North Holland, 1974, pp. 471–475.

[18] J. Dennis, "First version of a data-flow procedure language," in *Programming Symposium*, ser. LNCS, B. Robinet, Ed., vol. 19. Paris, France: Springer, 1974, pp. 362–376.

[19] Arvind and K. Gostelow, "The U-interpreter," *IEEE Computer*, vol. 15, no. 2, pp. 42–49, 1982.

[20] A. Davis and R. Keller, "Data flow program graphs," *IEEE Computer*, vol. 15, no. 2, pp. 26–41, February 1982.

[21] A. Aho, M. Lam, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd ed. Addison-Wesley, 2006.

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