HAL
open science

Efficient Compilation of Esterel for Multi-core Execution

Simon Yuan, Li Hsien Yoong, Partha S. Roop

To cite this version:

Simon Yuan, Li Hsien Yoong, Partha S. Roop. Efficient Compilation of Esterel for Multi-core Execution. [Research Report] RR-8056, INRIA. 2012. hal-00728149

HAL Id: hal-00728149

https://inria.hal.science/hal-00728149v1

Submitted on 5 Sep 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Efficient Compilation of Esterel for Multi-core Execution

Simon Yuan , Li Hsien Yoong , Partha S Roop

RESEARCH REPORT
N° 8056
September 5, 2012
Project-Teams POP ART

ISSN 0249-6399 ISRN INRIA/RR--8056--FR+ENG

Efficient Compilation of Esterel for Multi-core Execution

Simon Yuan *, Li Hsien Yoong †, Partha S Roop ‡

Project-Teams POP ART

Research Report n° 8056 — September 5, 2012 — 49 pages

Abstract: Recent advances in processor technology have lead to affordable multi-core processors, which could even be used in embedded applications. However, many embedded applications are safety-critical and require suitable abstractions such as the synchronous abstraction. Esterel is one such language belonging to the synchronous family and has been used extensively in the design of safety critical systems. While several compilation techniques of Esterel have been proposed, these are unsuitable for multi-cores due to the inherently sequential approach of *compiling away* the concurrency. We overcome this limitation by proposing two distinct approaches that distribute Esterel threads evenly across multi-core architectures. The first approach statically distributes threads based on the computation intensity approximated by the number of instructions generated from each thread. The second approach distributes threads dynamically using a thread queue that dispatches a thread whenever a core becomes idle. We have performed extensive benchmarking over large Esterel programs to illustrate that, using the static approach, achieving throughput with parallel execution of Esterel is benchmark dependent. However, the dynamic approach not only benefits data-dominated Esterel programs, but also large control-dominated ones. In particular, gains in performance of 36% and 93% were attained for a large control-dominated program using a dual-core and quad-core Microblaze processor, respectively.

Key-words: Multi-core, synchronous languages, Esterel, compilation.

* Department of ECE, University of Auckland, New Zealand `iyua002@aucklanduni.ac.nz`
† Department of ECE, University of Auckland, New Zealand `lyoo002@aucklanduni.ac.nz`
‡ Department of ECE, University of Auckland, New Zealand `p.roop@auckland.ac.nz`. INRIA Pop Art external collaborator funded by the AFMES associated team.

RESEARCH CENTRE GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Compilation efficace d'Esterel pour exécution sur multi-cœurs

Résumé : Les récentes avancées technologiques des processeurs ont rendu abordables les processeurs multi-cœurs qui ont pu être utilisés dans les systèmes embarqués. Cependant, beaucoup de systèmes embarqués ont des contraintes de sécurité qui demandent des abstractions adaptées, comme les abstractions de synchronisation. Esterel est un langage de la famille des langages synchrones, largement utilisé dans l'élaboration de système avec des contraintes de sécurité. Plusieurs techniques de compilation ont été proposées pour Esterel, mais elles ne sont pas adaptables pour les processeurs multi-cœurs en raison du caractère séquentiel intrinsèque à la compilation, loin de la simultanéité. Cette limitation est franchie grâce à deux approches distinctes qui distribuent les threads Esterel uniformément sur l'architecture multi-cœurs. La première approche distribue statiquement les threads issues de l'intensité calculée approximée par le nombre d'instructions générées par chaque thread. La seconde approche distribue dynamiquement les threads en utilisant une file qui envoie les threads lorsqu'un cœur est inactif. Nous avons réalisé d'importants benchmarking sur un grand nombre de programmes Esterel pour illustrer que la réalisation de débit avec l'exécution en parallèle statique de Esterel est dépendante du benchmark. En revanche, l'approche dynamique ne profite pas qu'aux programmes Esterel à dominance de donnée, mais aussi bien à ceux à large dominance de contrôle. En particulier, des gains de performance de 36% et de 93% ont été atteints pour des programmes Esterel à dominance de contrôle en utilisant respectivement un processeur dual-core et un processeur quad-core Microblaze.

Mots-clés : Multi-cœur, programmation synchrone, Esterel, compilation.

Although Esterel is inherently concurrent, Esterel compilers have typically removed the currency away in their implementations and execute sequentially. Parallelizing Esterel programs for better performance is notoriously difficult due to the synchronous semantics of the language. In particular, causality issues introduced by instantaneous broadcast communication put significant effort on the compiler to correctly handle the communication. Hence, there are very minimal attempts to achieve this objective [7, 3, 11]. This paper proceeds to present techniques for implementing the concurrency in Esterel with true parallelism (see [12] for the distinction between concurrency and parallelism).

Implementing parallelism for Esterel in hardware is simpler than software as the synchronous approach is a convenient paradigm for digital hardware design. A global clock is used to advance sequential states, while propagating signals via parallel combinational paths. In software, however, this approach is less natural, as it is difficult to efficiently implement concurrent threads with lock-step synchronization using an OS. This traditional approach of implementing the concurrency with an OS is highly inefficient as large Esterel programs may easily consist of hundreds of threads. If each of these threads are implemented separately on an OS, context-switching cost will be prohibitively high. This is due to each thread having to be scheduled at least once in each clock cycle, and possibly more if inter-thread communication is required. Consequently, software implementations of Esterel have typically compiled the explicit concurrency to yield statically scheduled sequential code that can run directly on a single-core processor [6, 10, 14], including STARPro introduced in [19, 20].

Recent advancements in microelectronics, however, have led to the increasing use of multi-core processors to achieve better power-performance tradeoff. As the power consumption of a processor increase by a power of two with respect to its hardware clock frequency, heat and power consumption have become increasingly more difficult to manage. In order to keep up with the demand for more computation power, an alternative approach is to increase the number of core processing units with little to no increase in the processor clock speed. This is possible thanks to the ever advancing silicon processing technologies keeping up with the rate described by Moore's law [13].

To take advantage of multi-core architectures, execution of Esterel has to be distributed. Unfortunately, the static schedule generated by most Esterel compilers is poorly-suited for multi-core processors. This paper will present two approaches to overcome this limitation by compiling Esterel in a manner that preserves the concurrency at the source level so that the resulting code can run in parallel on the available cores.

The task of compiling Esterel for efficient parallel execution is challenging due to the following:

1. Frequent thread synchronization that is required at the boundary of each clock cycle.
2. Instantaneous communication that occurs within a *tick* between multiple threads.
3. Distribution of threads to processor cores is intertwined with problem 1 and 2.

These factors limit the amount of parallel execution that can actually take place, irrespective of the number of cores available. Problem 1 and 2 deal with thread scheduling such that the semantics of instantaneous broadcast is strictly obeyed while executing threads in parallel. The implementation of thread scheduling also heavily influences the way threads are distributed as described in problem 3. Despite thread scheduling being tightly coupled to thread distribution, the techniques introduced in this paper is able to divide thread scheduling and distribution into problems that can be solved independently. The key is to resolve the statuses of signals at runtime so that threads can be scheduled dynamically. The dynamic scheduling approach gives more

freedom to the thread distribution problem by allowing threads to execute in any arbitrary order. Two approaches for thread distribution will be presented: an approach that statically distribute to cores based on a greedy heuristic, and a dynamic approach where threads are distributed to idle cores at run-time using a FIFO queue.

The remaining of the paper will proceed by introducing Esterel through a small example in Section 1. We will describe dynamic scheduling using the run-time signal resolution technique and its correctness in Section 2. Then, the implementation of run-time signal resolution and the intermediate format used will be explained in Section 3. The static thread distribution approach and the dynamic approach will be described in Section 4 and 5 respectively. Evaluation of the effectiveness of these two approaches will be presented in Section 6, and then finally concluded in Section 7.

1 An Esterel Example

The *ParallelData* example shown in Fig. 1, is a parallel data processing pipeline consisting of three threads, demarcated by the || operator. These threads communicate via the local signals `S` and `S2`. The first thread begins by waiting for the `Start` signal. Once received, the program starts to process the data by calling the `processData1` procedure with `result` passed to the procedure by reference (the arguments in the first pair of parentheses are passed by reference while the second pair pass by value). The output of the procedure is then stored in `result` and sent using signal `S` to the second thread for further processing. The second thread waits for `S` using the `await` statement. Note that the data sent from the first thread is buffered and retrieved in the next *tick* using the `pre` keyword. This results in a software pipeline: while the second thread works on the data received from the first, the first thread continues to produce new data in parallel. As soon as the second thread completes its processing, the final result is sent to the third thread using the signal `S2`. If the `CheckStatus` input is activated, the body of the `every` statement in the third thread will start to execute. It takes the data from the second thread, does a self-diagnostic test with the given data, and then indicates whether the process is working normally. The program can be stopped at any time by activating the `Stop` signal.

Fig. 2 shows a timeline of an example execution trace given a set of inputs. Tick boundaries are represented as the vertical lines and time progress along the horizontal line towards the right hand side. No inputs were given in the first *tick* as `ParallelData` does not react to any input in the first *tick*. In the second *tick*, `Start` becomes present and detected by the `await Start` statement on line 11. The first thread enters the loop, calls `processData1`, emits `S` and finally stops at the `pause` statement on line 15. The second thread does not react to `S` as it reacts to the *previous* status of the `S`, which was absent in the first *tick*. The third thread remains awaiting for `CheckStatus` on line 24.

From the third *tick* onwards, `processData2`'s buffer is filled with data produced by `processData1` in the first thread in the previous tick. As `S` became present in the second `tick`, the `await pre(S)` statement detects the presence of `S`, calls `processData2` with the data embedded in `S` and finally emits `S2`. Both `S` and `S2` are emitted in the third *tick*.

On the forth tick, `CheckStatus` becomes present. The first two threads continue to process the data and emit `S` and `S2`. The third thread detects `CheckStatus` and calls `selfDiagnose` on the data embedded in `S2` on line 25. The `selfDiagnose` host function detects an error in the data, returns false and finally emits `Error` on line 28.

On the fifth tick, `Stop` becomes present. The `Stop` *conditional* node detects the signal and takes the present branch. Finally, the program terminates in this *tick*.

```
1 module ParallelData:
2 input Start, CheckStatus, Stop;
3 output Good, Error;
4 type Data;
5 procedure processData1(Data)();
6 procedure processData2(Data)(Data);
7 function selfDiagnose(Data) : boolean;
8 abort
9   signal S : Data, S2 : Data in
10  var result : Data, final : Data in
11    await Start;
12    loop
13      call processData1(result)();
14      emit S(result);
15      pause;
16    end loop
17    ||
18    loop
19      await pre(S);
20      call processData2(final)(pre(?S));
21      emit S2(final);
22    end loop
23    ||
24    every CheckStatus do
25      if selfDiagnose(?S2) then
26        emit Good;
27      else
28        emit Error;
29      end if
30    end every
31  end var
32  end signal
33 when Stop
34 end module
```

Figure 1: The `ParallelData` example written in Esterel

ticks ⟶	`Start`		`CheckStatus`	`Stop`	**input**
	`S`	`S`	`S`		
		`S2`	`S2`		**output**
			`Error`		

Figure 2: A reaction timeline of the `ParallelData` example

2 The run-time signal resolution approach

Static scheduling techniques [8, 14, 10] rely on analyzing the dependencies between threads at compile-time. Attempting to parallelize a statically scheduled Esterel program would yield little performance gain due to the *sequentialization*. Enforcing a strong execution order would defeat any performance gain through parallel execution. To maximize the number of threads that can execute in parallel, a scheduling technique must satisfy the following:

- The ability to execute as many threads as possible, only enforcing an execution order at the points of communication.
- Threads are able to freely execute in any arbitrary order at any other time. This would allow load distribution be decoupled into an independent problem.

Given the requirements above, one can quickly arrive at the conclusion that, a key to effective parallelization lies in the ability to resolve signal dependencies at run-time, instead of having threads statically sequentialized.

The run-time signal resolution approach works on the principle that a signal is not read until its status is known. If the status of the signal is still unknown when a thread attempts to read a signal, the scheduler freezes the thread and picks another thread to start execution. Using this approach, the threads can be executed in any arbitrary order, and an execution order is only enforced at the points of communication.

A signal is *resolved* when it is emitted or its emission is ruled out. Detecting the presence of a signal is straightforward; it takes only one emission of the signal to confirm the status. Unlike detecting for presence of a signal, detecting the absence of a signal requires concerted effort from all threads who may potentially emit to the same signal. These threads, called potential emitters, must collectively agree on the absence of the signal in order to resolve the signal as absent. To schedule threads at run-time, a thread must be suspended when a signal being accessed is still unresolved. While the thread is suspended, its potential emitters have a window of opportunity to execute. As each potential emitter execute, it may removes itself as a potential emitter when the emission of the signal has been ruled out. If the unresolved signal is not emitted while the potential emitters execute, the number of potential emitters will eventually reduce to zero and the signal can be eventually concluded as absent.

With the run-time signal resolution mechanism, the order threads execute become irrelevant. The scheduler only has to ensure that any suspended thread is eventually rescheduled by executing each thread in a round-robin fashion. A high-level representation of such cyclic executive is presented in Fig. 3. The scheduler itself forms the skeleton of *reactive function*. It is called once every *tick*, and returns the termination status of the *tick*.

The *reactive function*, on line 4, starts by initializing the number of potential emitters for each signal. The number of potential emitters of a signal can be calculated by counting the number of threads that can potentially emit to the signal. Then the root thread of the program is added to a set of threads T as the candidates to be scheduled. The cyclic executive on line 8 is used to continuously schedule a threads until T becomes empty. Within the cyclic executive, the first thread t is selected and removed from T for execution. A thread t is a sequential composition of Esterel statements represented as a function t`()`. It would return to the cyclic executive for one of the following reasons:

- t has reached a *tick* boundary.
- t has terminated normally, by exception or preemption.
- t has been blocked due to an unresolved signal.

```
1  T denotes a vector of active threads as candidates to be scheduled
2  t denotes a sequential composition of Esterel statements represented as a function
3  function reactive_function
4      foreach signal s shared by threads in T do
5          n_s := the number of potential emitters of s
6      end
7      add the root thread to T
8      while T ≠ φ do
9          select and remove the first thread t ∈ T
10         term_code := t()
11         if term_code = ∞ then
12             add t to the back of T
13         end
14     end
15     return term_code
16 end
```

Figure 3: The high-level representation of the cyclic executive for run-time signal resolution

The termination status represented by *term_code* indicates the reason for t's return. Threads in T are removed as they are selected for execution. However, t may be added back to T if it has been blocked due to an unresolved signal. Then, t has to be rescheduled by pushing t to the back of T. Eventually, when all threads are removed from T, the *reactive function* completes a *tick* by returning the termination code of the last executed thread. Due to reasons that will be explained later, the last executed thread will always be the root thread. A thread t interacts with the cyclic executive in Fig. 3 in four kinds of ways during execution:

Forks into more threads — saves itself in a buffer f, then add all child threads of the fork to T. To ensure all child threads terminate synchronously, each child thread has to check the the number of remaining child threads that are alive before it terminates. The last child thread to terminate is responsible to resume and reschedule its parent thread by adding f to T.

Attempts to read a signal — returns ∞ if the attempt to read failed due to unresolved signal due to $n_s \neq 0$ (some potential emitters may still emit). Otherwise, the thread proceeds to read the signal and continue execution normally.

Emits a signal — emits the signal s and reset the number of potential emitters by $n_s := 0$. Note that n_s is initialized by the cyclic executive on line 4 in Fig. 3.

Rules out emission of a signal — removes itself as a potential emitter by decrementing $n_s := n_s - 1$.

The hierarchical structure of threads are preserved by the first kind of behaviour described above. During the execution of t, if it forks into more threads, the address of the parent t is saved to f prior to suspending it and adding its child threads to T. Suspension of the parent is done implicitly by not adding it to T until all its child threads complete. The last child thread to terminate will resume its parent thread by adding f to T.

For example, the following sketch Esterel program

```
1 [
2     T3
3     ||
4     T4
5 ]
6 ||
7 T2
```

consists of five threads represented by `T0` (root thread), `T1` (parent of two threads), `T2`, `T3` and `T4`. As `T0` and `T1` are implicit in this example, they do not appear in the code. `T0`, being the root thread, is added to the vector T first by the cyclic executive in Fig. 3, i.e., $T = \{T0\}$. Initially, `T0` is the only thread in T. Then, `T0` is selected and removed from T for execution. T becomes empty momentarily before `T0` spawns `T1` and `T2` by adding these two thread to T and suspends `T0` by adding it to a buffer F, i.e., $T = \{T1, T2\}$ and $F = \{T0\}$. When `T1` is removed from T for execution, `T1` immediately spawns `T3` and `T4` followed by adding itself to F, i.e., $T = \{T2, T3, T4\}$ and $F = \{T0, T1\}$. Assuming `T2` terminates normally, `T0`, `T1`, `T3` and `T4` remain alive with `T0` and `T1` being suspended, i.e., $T = \{T3, T4\}$ and $F = \{T0, T1\}$. When `T3` and `T4` terminate in their respective order, `T4` removes `T1` from F and adds `T0` to T in order to resume its parent thread, i.e., $T = \{T1\}$ and $F = \{T0\}$. Then, when `T1` executes, as the last child thread of `T0` to terminate, it resumes `T0` by removing `T0` from F and adds it to T, i.e., $T = \{T0\}$ and $F = \phi$. Finally, `T0` terminates immediately when resumed, the cyclic executive terminates as the loop condition $T \neq \phi$ on line 8 in Fig. 3 no longer holds.

The suspension and resumption of forked parent threads implement the normal synchronous termination of threads. This ensures that the parent threads will always terminate after their child threads. Subsequently, the last thread to execute in the cyclic executive will always be the root thread.

The run-time signal resolution approach is correct, so long as the Esterel program is known a priori to be causal. The definition of a causal program and the formalization of this statement is given in Definition 1 and Lemma 1 respectively.

Definition 1. *A program without any instantaneous cyclic dependencies in its control-flow is defined as causal [5]. This implies that the statuses of all signals in a causal program can be constructively derived from the facts established about other signals without having to make any assumptions regarding their statuses.*

Lemma 1. *Let C be a causal Esterel program. Further, let T be the set of parallel threads in C, and S be the set of signals shared (emitted and tested) between two or more threads in T. The cyclic executive will resolve at least one signal in S in each iteration.*

Proof. The proof for Lemma 1 proceeds by first presenting a number of axioms regarding the scheduling algorithm described earlier:

Axiom 1. *The scheduling algorithm schedules threads in T in a round-robin fashion within a cyclic executive in each tick. Since C is causal, there will not be any dependency cycles between threads that would cause the program to deadlock. This implies that all threads will eventually get scheduled.*

Axiom 2. *At the start of each scheduling cycle corresponding to a new tick, all signals in S that are potentially tested are prevented from doing so until the statuses of the signals have been conclusively determined. This ensures that signals can never be tested prematurely. Signals can never be tested before they are emitted or before all their potential emitters have been ruled out, irrespective of the order that threads get scheduled.*

Axiom 3. *Each scheduled thread completes by returning a termination code. Threads that are blocked due to an unresolved signal will return a termination code of ∞, or a finite positive integer otherwise. Threads that return a termination code of ∞ will be rescheduled in the next iteration of the cyclic executive.*

Based on these axioms, the proof can now be done by induction. Since C is causal, the execution of threads in T will result in at least one of the following outcomes during a *tick*:

1. At least one shared signal is resolved to be present due to its emission, or absent when all its potential emitters collectively agree that the signal can no longer be emitted in that *tick*. In either case, $|S|$ will decrement by the number of resolved signals.
2. The thread gets blocked from reading a signal due to that signal being unresolved. That thread will be rescheduled in the next iteration of the cyclic executive. $|S|$ remains unchanged in this case.
3. The thread gets removed from T because it neither emits nor tests a signal in S. Consequently, $|S|$ remains unchanged.

Base case

Consider the case where $S = \{s_1\}$ is shared among a set of threads, $T = \{t_0, t_1, \ldots, t_M\}$. There are two possible scenarios:

- If t_i ($i \in \{0, 1, \ldots, M\}$) only tests but does not emit s_1, $|S|$ remains the same (Outcome 2).
- If t_i is a potential emitter of s_1, then by Axiom 2, there are two possibilities:
  1. Signal s_1 is emitted.
  2. Thread t_i rules itself out as a potential emitter of s_1.

  If s_1 gets emitted, all signals are resolved and $|S|$ becomes zero. Otherwise, each t_i will rule itself out as a potential emitter, as the cyclic executive schedules them consequtively within the first iteration. Hence, after one iteration, s_1 will either be emitted, or have all its potential emitters ruled out. Therefore, the number of unresolved signals will decrease to zero after at most one iteration of the cyclic executive.

Hypothesis

Assume that Lemma 1 holds for the case where $S = \{s_1, s_2, \ldots, s_j\}$.

Inductive step

Now, consider the case where $S = \{s_1, s_2, \ldots, s_j, s_{j+1}\}$ is shared among a set of threads, T. After j iterations of the cyclic executive, at least j signals would have been resolved, leaving at most one unresolved signal, say, s_u. Then, in the $(j+1)$th iteration of the cyclic executive, the execution of thread t_i will again result in one of two scenarios as before:

- Thread t_i gets blocked while attempting to read s_u (Outcome 2).
- Thread t_i, by Axiom 2, either:
  1. emits s_u; or
  2. rules itself out as a potential emitter of s_u.

If s_u gets emitted, all signals are resolved and $|S|$ becomes zero. Otherwise, each t_i will rule itself out as a potential emitter.

At the end of the $(j+1)$th iteration, s_u will either be emitted, or have all its potential emitters ruled out. It cannot happen that s_u is still unresolved, since the statuses of all other signals in S are already known. The first scenario can only perpetuate after the $(j+1)$th iteration if the test of s_u itself determines its resolution (a contradiction of the causal assumption). Therefore, the cyclic executive will always resolve at least one signal in S in each iteration. □

Theorem 1. *For any execution order of all threads $t_i \in T$ $(i \in \mathbb{N})$ in each* tick*, the number of signals in S that are unresolved will decrease monotonically, and eventually converge to zero after at most $|S|$ iterations of the cyclic executive used for dynamic scheduling.*

Lemma 1 can now be used to prove Theorem 1.

Proof. This will again be done by induction.

Base case

For the case where T consists of only one thread, there is no need for any signal resolution. Hence, we begin with the base case, where $T = \{t_0, t_1\}$.

- If the execution of either t_0 or t_1 results in Outcome 3 at any time, the proof is degenerate, as it implies that $|S|$ has either become zero, or the remaining thread in T will decrement $|S|$ to zero without getting blocked any further.
- Otherwise, assume $S = \{s_1, s_2, \ldots, s_N\}$. Then, based on Lemma 1, t_0 and t_1 will either resolve one or more signals in S (Outcome 1) before getting blocked, or will get blocked without resolving any signal (Outcome 2) in each iteration of the cyclic executive. However, both threads will not get blocked in the same iteration without resolving any signal, since C is causal. In this case, $|S|$ will decrement by an integer value between $[1, N]$ in each iteration. Therefore, the number of unresolved signals will decrease monotonically and eventually converge to zero after at most $|S|$ iterations of the cyclic executive.

Hypothesis

Assume that the theorem holds for the case where $T = \{t_0, t_1, \ldots, t_k\}$, with the number of iterations required to reduce $|S|$ to zero being less or equal to N.

Inductive step

Then, consider the case where $T = \{t_0, t_1, \ldots, t_k, t_{k+1}\}$:

- If the execution of t_{k+1} immediately results in Outcome 3, the number of iterations of the cyclic executive required to resolve all signals in S will still be the same as in the case where $T = \{t_0, t_1, \ldots, t_k\}$, since t_{k+1} does not affect any signal in S.
- Otherwise, two scenarios are possible:
  1. t_{k+1} only tests for signals in S, and is not a potential emitter of any signal in S. As before, since t_{k+1} does not affect any signal in S, the number of iterations of the cyclic executive required to resolve all signals in S will still be the same, as in the case where $T = \{t_0, t_1, \ldots, t_k\}$. In particular, if all signals in S can be resolved within N iterations for the set of threads $\{t_0, t_1, \ldots, t_k\}$, they will also be resolved within N iterations for the set $\{t_0, t_1, \ldots, t_k, t_{k+1}\}$.

2. t_{k+1} is a potential emitter of one or more signals in S. The addition of more potential emitters for a given signal can only result in that signal being potentially resolved in fewer iterations, but never more. Consequently, if at most N iterations were originally required to reduce $|S|$ to zero when $T = \{t_0, t_1, \ldots, t_k\}$, the inclusion of t_{k+1} to T will also only at most require N iterations.

Since both the base and inductive cases have been proven, Theorem 1 will hold for any causal Esterel program with two or more threads. □

Due to theorem 1 and Axiom 2, the semantics of instantaneous broadcast is preserved. The resulting behaviour of a dynamically scheduled program would be completely the same as a statically scheduled program. In the following section, the actual implementation of the run-time signal resolution algorithm will be described.

3 Implementation of run-time signal resolution

Implementing the run-time signal resolution algorithm involves two stages of the compilation process. One stage is during the construction of the intermediate format, and the second is the code generation stage. This section will proceed by describing the GRC intermediate format [14] in the next subsection.

3.1 The GRC intermediate format

The graph code (GRC) format represents an Esterel program with an acyclic directed graph. It consists of two parts: (1) a hierarchical state graph (HSG), and (2) a concurrent control-flow graph (CCFG). As the discussions of the compilation techniques in this paper do not involve the HSG, from here on, the term GRC will always be referred to the CCFG part of the representation.

The construction process of GRC introduces a unique distinction between the initial behaviour and the resumption behaviour of each Esterel statement, called the *surface* and *depth* behaviour respectively. We elaborate these terms with the following:

- The *surface* behaviour describes the micro-steps performed in the first *tick* of any composition of Esterel statements.
- The *depth* behaviour describes the micro-steps performed in the subsequent *ticks* of any composition of Esterel statements.

We will first describe the types of GRC nodes that exist in GRC followed by its control-flow. A list of GRC nodes is illustrated along the top of Fig. 4. These nodes are described as the following:

***Fork* and *Join*:** Create and destroy threads respectively.

***Test*:** The graphical representation of the `present` or the `if` statement.

***Action*:** Performs an action such as variable assignment, emitting a signal and calling a procedure.

***Switch*:** Encodes the state of a thread. Each branch under this node represents a *tick*.

***Enter*:** Sets the state encoded in a *switch* node of a thread.

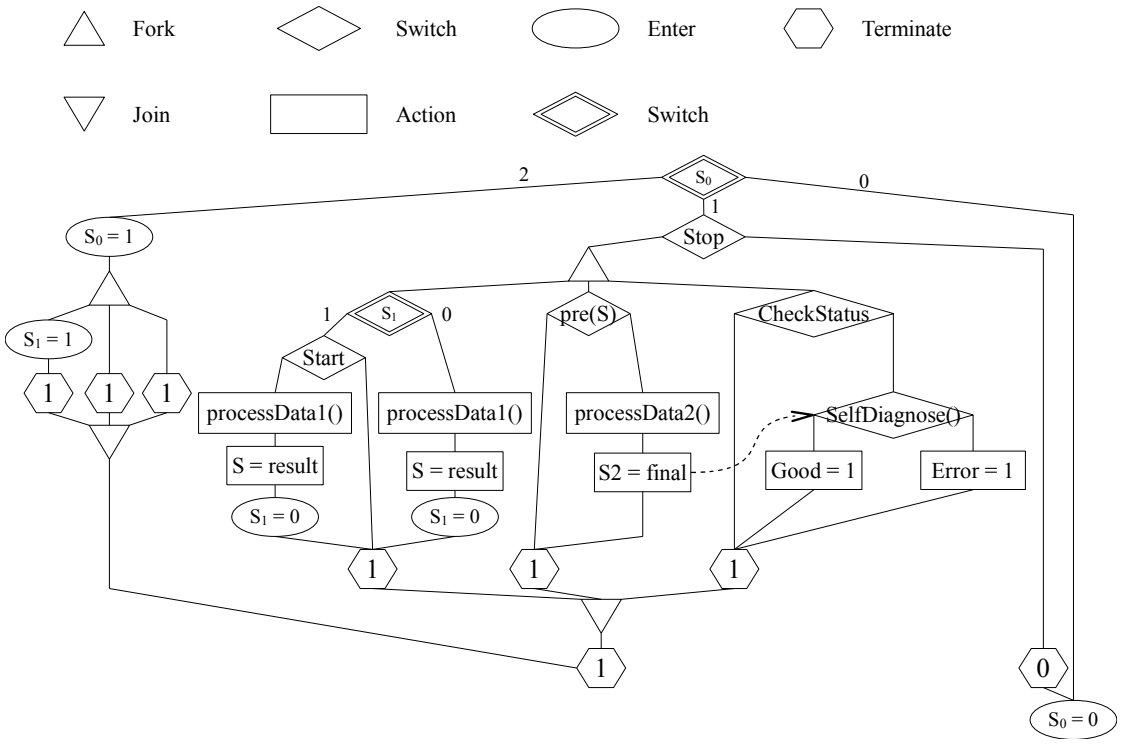

Figure 4: The `ParallelData` example represented in the Graph Code Format

***Terminate*:** This node, introduced in [10], is a variant of the *sync* node introduced in [14]. A *terminate* node encodes the completion status of a thread. A completion code of zero denotes a normal termination; a value of one indicates the thread has been paused for a *tick*; and a value greater than one indicates the thread has been terminated by an exception or preemption. This clever encoding scheme (courtesy of Berry [4]) succinctly captures the *meet at the rendezvous* behaviour of concurrent statements and the *winner-take-all* behaviour of simultaneously thrown exceptions and preemption. Within a pair of *fork* and *join* nodes, *terminate* nodes immediately precede the *join* node and dictate the continuation context the *join* node must take when threads terminate. This is the primary mechanism for compilation of nested exception and preemption. The hierarchical nesting of the exception and preemption constructs determine the priorities of their reactions when triggered simultaneously. The higher the priority the larger the value a *terminate* node returns. The highest value will take precedence when *terminate* nodes merge at a *join* node.

Nodes in GRC are connected by *control arcs* (solid lines) and *data dependency arcs* (dashed line) to describe the control-flow and communication between nodes respectively. A GRC graph is traversed from top to bottom once every *tick*. There are no loops in GRC as the repetition is achieved by controlling of the states of the program in the subsequent *tick* using *switch* and *enter* nodes. *Switch* nodes are later translated into state variables that are assigned by *enter* nodes.

As an example, the GRC shown in Fig. 4 represents the `ParallelData` Esterel module. We will assume the same input trace as Fig. 2 to describe the control-flow. The program starts from

the *switch* node at the top. The nodes residing on the left branch of the top *switch* node define the *surface* behaviour of the program. The program immediately pauses for one *tick* due to the await statements. Two *enter* nodes are inserted to setup the states of the subsequent *tick* by assigning $S_0 = 1$ and $S_1 = 1$.

In the second *tick*, the top *switch* node reads the value of S_0 and selects the middle branch. `Start` is now present and the program follows the present branch of the *test* `Start` node. S_1 is set to 0 to ensure that `Start` will no longer be tested in the subsequent *ticks*. Then, `S` is emitted before the first *tick* ends.

From the third *tick* onwards, `processData2`'s buffer, implicitly created by the `pre` operator, is filled with data produced by `processData1` in the first thread in the previous tick. The `pre(S2)` *test* node in the second thread detects the presence of `S` and selects the present branch. Then, `S2` is emitted before the third *tick* ends. Both the first and the second thread now repeat these execution paths indefinitely unless stopped by the `Stop` signal.

On the forth tick, `CheckStatus` becomes present. The first two threads continue to process the data and emit `S` and `S2`. The third thread detects `CheckStatus`, follows the present branch, then calls `selfDiagnose`. The procedure takes the data emitted via `S2` by the second thread and analyzes the data and discovers an anomaly. The thread takes the else branch of the `selfDiagnose` *conditional* node and emits `Error`.

On the fifth tick, `Stop` becomes present, which preempts the program due to the `abort–when Stop` statements. The program follows the present branch and exits the program via the *enter* node at the bottom. This *enter* sets $S_0 = 0$ and forces the program to forever take the right branch of the top *switch* node in the subsequent *ticks*. This effectively terminates the program as the program will no longer react unless reset.

3.1.1 Extensions to GRC

To schedule Esterel threads, an Esterel program is represented in a variant of GRC [14]. GRC features a unique way of determining execution path using termination codes from each statement in Esterel. The run-time signal resolution approach elegantly piggy-backs on this special encoding scheme of the termination codes by partially capture scheduling information within the intermediate format. GRC represent synchronous termination of threads by encoding the termination code of each thread in *termination* nodes at the end of each thread. The *termination* nodes provide a mechanism for the program to react differently based on the values stored within the *termination* nodes. The termination mechanism in GRC is augmented with an additional termination code for resolving signals at run-time. Hence, the use of GRC would minimizes the effort required to implement a mechanism for resolving signals at run-time and does not require inserting control logic into the graph for run-time signal resolution.

To support resolving signals at run-time, the GRC format has been augmented with two additional nodes. We will illustrate the two additional nodes using the the `ParallelData` example presented in Fig. 1 and its corresponding representation in GRC augmented with run-time signal resolution nodes is presented in Fig.5.

The program starts by testing S_0, which is initially set to 2, using the *switch* node at the top of the graph. The *switch* node determines the current state of the program. Here, branch 2 of S_0 represents the initial state of the program. A *guard* (`S2?`) node is inserted in places where a signal must be protected from being read prematurely. A *resolution* (`S!`) node is inserted at suitable points to remove a thread from the set of potential emitters for `S2`. For this example, a *guard* node has been inserted immediately before the *test* node of `selfDiagnose(?S2)`, while *resolution* nodes have been inserted immediately after the *emit* ($S2 = final$) node, and on all paths that would not lead to `S` being emitted. The *resolution* nodes inserted immediately below

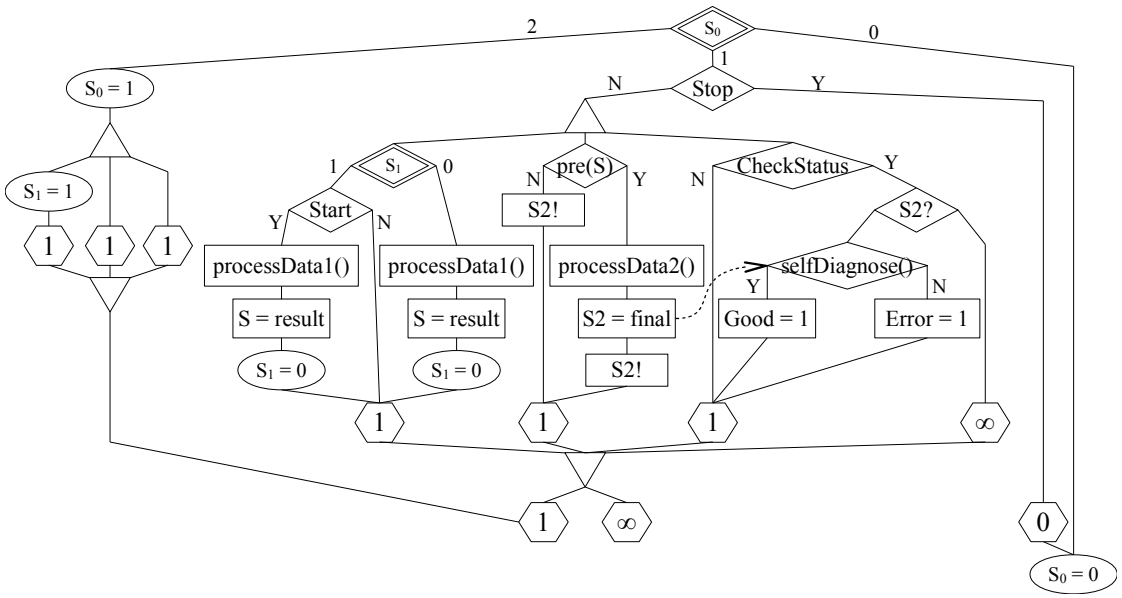

Figure 5: The `ParallelData` example represented in the Graph Code Format

signal emissions *immediately declares the signal as present*. However, other *resolution* nodes *removes the thread from the set of potential emitters*.

Synchronizing threads to *tick* boundaries is achieved explicitly in GRC as part of the control-flow by forking and joining threads at the start and the end of of each *tick* respectively. This behaviour is exemplified by the *fork-join* pair in each *tick* when concurrent statements execute in Fig. 5. In the `ParallelData` example, both the initial *tick* (the left branch under the top *switch* node) and the subsequent *ticks* (the middle branch under the top *switch* node) start by forking into two threads and joining them at the bottom of the GRC graph. This explicit *tick* synchronization mechanism at the intermediate representation level saves the cyclic executive in Fig. 3 from having to handle *tick* synchronization. The cyclic executive requires only the value of the termination code when the program reaches the bottom of the GRC graph to determine whether a *tick* has elapsed based on the following:

- A value of ∞ indicates some threads are still alive but blocked due to unresolved signals. These threads must be scheduled for execution in the next iteration of the cyclic executive.
- A value of zero indicates the program has terminated normally.
- A value of one indicates the program has reached a *tick* boundary. During a *tick*, the internal states (represented by the *switch* nodes) of the threads in the program are assigned for resumption in the subsequent *tick*. At the end of a *tick*, the *reactive function* returns to its caller. Calling the *reactive function* again starts a new *tick* and the previously paused threads will resume from the states assigned by the previous *tick*.

Implementing run-time signal resolution at the GRC representation level requires inserting *guard* nodes and *resolution* nodes at appropriate places. While inserting *guard* nodes is simple, inserting *resolution* nodes is more involved. A *guard* node is inserted before a *test* node whenever the *test* node reads a signal that is potentially emitted from at least one other thread. In contrast, inserting *resolution* nodes involves an algorithm that will be described in the next subsection.

3.2 The insertion algorithm for signal resolution nodes

During the compilation process, the compiler inserts a *guard* node before a *test* node to prevent the signal from being tested prematurely, i.e., when the signal status is yet unresolved. To get past a *guard* node, the status of the signal must be resolved. It takes only *one* potential emitter that emits the signal in order to confirm the presence of the signal. However, determining absence of a signal requires confirmation from *all* potential emitters. To further complicate the matter, threads often communicate back and forth. Such threads cannot wait till the *tick* boundary to determine status of the shared signals if those signals are *guarded*. Those threads would deadlock due to indefinitely waiting for each other to resolve the statuses of the signals. For this reason, the compiler must insert *resolution nodes* for each potential emitter to ensure they keep the signal readers informed about the status of the shared signal.

To insert *resolution* nodes, the insertion algorithm has to find the closest *common parent* node. A *common parent* is a *conditional* node under which the signal producer and consumer are attached. The highlighted *switch* node at the top of Fig. 6 is an example of a *common parent* of the *emit S* (i.e., `S = 1`) node and the *test* node for `S`. The highlighted node at the bottom is the signal producer, and the highlighted node in the middle is the consumer. The producer and the consumer execute concurrently under the *common parent* node as they are in two different branches of the *fork* node. Hence, a dependency exists in the example in Fig. 6 that needs to have guarded access to the shared signal.

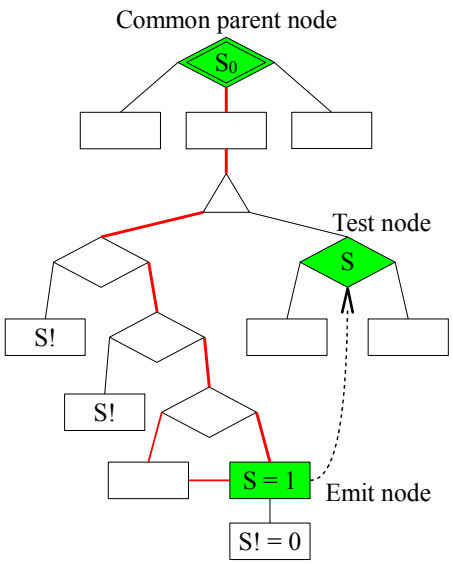

Figure 6: Illustration of places to insert *resolution* nodes

Finding the *common parent* node is only the first step to insert *resolution* nodes. Not all paths from the *common parent* node would lead to the *emit* node. As soon as the potential emitter deviates from the paths to the *emit* node, it must go through a *resolution* node. For example, the paths to the *emit* node are highlighted in red in Fig. 6. From the *common parent* onward, any path deviating from the path in red needs to have a *resolution* node inserted. The intuition is to find all the *conditional* nodes on the path from the *emit* node to the *common parent* node, and then insert *resolution* nodes under the branches of those *conditional* nodes except the branch that leads to the *emit* node. This is the basis for the insertion algorithm in

Fig. 7.

```
 1 Let C denote a set of nodes in a causal program
 2 procedure insert_resolution_nodes
 3     foreach node t ∈ C do
 4         if t is a test node then
 5             foreach data predecessor e of t do
 6                 insert_nodes(t,e)
 7             end
 8         end
 9     end
10 end
 1 procedure insert_nodes(t,e)
 2     insert a resolution node under e
 3     P := path_to_node(e)
 4     Q := path_to_node(t)
 5     c := common_parent_node(P,Q)
 6     n := e
 7     foreach conditional node p ∈ P do
 8         if p = c then  return
 9         mark_path_to_node(n,p,e)
10         n := p
11         foreach branch b under p do
12             if b is not on the path to e and
               a resolution node r for e has not been
               inserted under b then
13                 insert r at the top of b
14             end
15         end
16     end
17 end
```

Figure 7: Algorithm for inserting signal *resolution* nodes

This algorithm starts with `insert_resolution_node` in Fig. 7. The auxiliary functions used by `insert_nodes` are presented separately in Fig. 8. the algorithm is initiated with the `insert_resolution_nodes` procedure. On line 3, the algorithm traverses through the GRC of a program searching for *test* nodes. On line 5, the procedure follows each *data predecessor* e of the *test* node t and passes both t and e to the `insert_nodes` procedure. The term *data predecessor* refers to those nodes that have data dependency arcs leading to other nodes.

Within `insert_nodes`, it immediately inserts a *resolution* node under the e. The *resolution* (i.e., `S! = 0`) node inserted under e propagates the presence of the signal, allowing the signal to be consumed. An example of this node is the node labeled with `S!` attached immediately below the highlighted *emit* node in Fig. 6. The `path_to_node` function on Lines 3 and 4 create two vectors of nodes on the paths to the *emit* node (the P vector) and the *test* node (the Q vector) respectively. The `common_parent_node` function on line 5 finds the closest *common parent* node of the *emit* node and the *test* node by comparing P and Q. Once the *common parent* node is found, `common_parent_node` returns to the `insert_nodes` procedure. Line 6 and the loop on line 7 start searching from the *emit* node towards the *common parent* node to find additional

```
P denotes a vector of conditional nodes on the path to a node from the root node
function path_to_node(e)
    n := e
    while control predecessors of n > 0 do
        n := the first control predecessor of n
        if n is not a fork node then
            add n to P
        end
    end
    return P
end
function common_parent_node(P,Q)
    foreach p ∈ P do
        foreach q ∈ Q do
            if p = q then  return p
        end
    end
end
function mark_path_to_node(n,c,e)
    if n = c then  return true
    foreach control predecessor p of n do
        if mark_path_to_node(p,c,e) then
            mark the branch under c that n belongs to as a path to e
        end
    end
    return false
end
```

Figure 8: Auxiliary functions for the insertion algorithm

places to insert *resolution* (i.e., `S!`) nodes. Line 8 checks whether the *common parent* node has been reached. The procedure then returns to `insert_resolution_nodes` and move on to the next data predecessor; otherwise, the search continues. The `mark_path_to_node` function on line 9, presented in Fig. 8, recursively traverses each path of the given node 'n' upward. It stops traversing as soon as the target node c is reached, and marks the branch under c that n belongs to as a path that will eventually reach e. The `insert_nodes` procedure calls `mark_path_to_node` on a segment of the path between the *common parent* node and the *emit* node at a time, starting from the *emit* node to the nearest *conditional* node in vector P. The segment being searched is changed by line 10 in Fig. 7. The segments between c and e that are marked by `mark_path_to_node` are illustrated with the paths highlighted in red in Fig. 6. Lines 11 ∼15 then insert *resolution* nodes under each conditional node for each branch that deviates from the path to the *emit* node.

Running through the insertion algorithm on the example in Fig. 6 would result in *resolution* node being added under each brach of each *conditional* node on the path between the *common parent* node and the *emit* node. The algorithm would correctly identify the path marked in red, and avoid inserting *resolution* nodes on this path. However, the insertion algorithm assumes that all dependencies are valid. It would not work if the dependency is false. An example of both a valid and a false dependency is illustrated in Fig. 9.

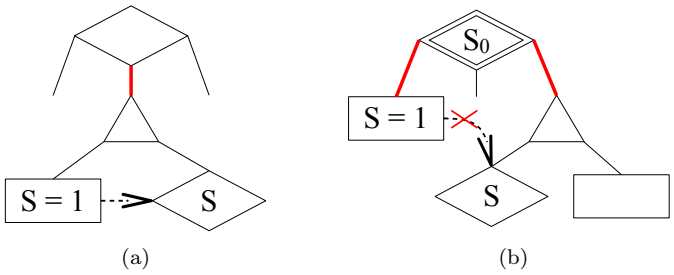

Figure 9: An example of (a) a valid and (b) false dependency

Fig. 9(b) shows an example of false dependency. In this example, the *emit* and the *test* nodes are located in different *ticks* as they are located on different branches of the *switch* node. The signal is emitted and tested in different *ticks*. Due to the compiler plainly inserting dependency between nodes by tracing the data successors or predecessors of a node, some of the matched dependencies are not real dependencies. This will create problems for dynamic signal resolution in two ways: (1) it creates extra scheduling overhead, or worse, (2) potentially produce incorrect behaviours due to attempts to resolve signals due to false dependencies. Hence, the algorithm in Fig. 10 is applied prior to the insertion algorithm to remove any false dependencies.

Determining whether a dependency is valid is very simple. The algorithm would only need to find the closest *common parent* of the *emit-test* node pair that are linked by a data dependency arc. Then, it determines whether the paths to each of the node of the pair reside on the same branch under the *common parent* node. If the pair reside on the same branch like the highlighted branch in the example in Fig. 9(a), the dependency is valid. If the dependency is invalid, the pair of nodes would reside on different branches of the *common parent* node, as illustrated in Fig. 9(b). This is precisely what the algorithm in Fig. 10 does.

The algorithm for removing false dependencies starts by looking for *emit* nodes that has at least one data successor. It then finds the closest *common parent* node on lines 6∼8. The branches that lead to the pair of nodes are compared to determine whether they reside on the

```
1  C denotes a set of nodes in a causal program
2  procedure remove_false_dependencies
3      foreach emit node e ∈ C do
4          if data successors of e > 0 then
5              foreach data successor t of e do
6                  N := path_to_node(e)
7                  M := path_to_node(t)
8                  c := common_parent_node(N,M)
9                  p := the branch under c that leads to e
10                 q := the branch under c that leads to t
11                 if e and t are in the same thread or p ≠ q then
12                     break the dependency between e and t
13                 end
14             end
15         end
16     end
17 end
```

Figure 10: Algorithm for removing false dependencies

same branch on line 11. In addition to that condition, line 11 also checks if the pair of nodes reside in the same thread. If either of these conditions is true, the dependency between the nodes would be removed.

3.3 Generating code with run-time signal resolution

The second stage of the implementation for run-time signal resolution takes place during code generation. We will first describe the implementation for sequential execution before we extend this scheduling scheme for parallel execution in Section 4 and 5.

To avoid complicating the discussion of the implementation with the full details of an actual generated code, we will illustrate a sketch of an Esterel program using the example in Fig. 11. The `sketch` example consists of four threads communicating via the local signal `S`. The thread at the top is the only reader of `S` while the other three are potential emitters to `S`. The GRC representation of this example is presented in Fig. 12 to reveal a little more detail of each thread. The GRC representation consists of a *fork* node spawning four threads. The left three branches of the *fork* node in Fig. 12 lead to the potential emitters of `S` while the consumer of `S` is located on the right most branch of the *fork* node. The *test* node in each potential emitter represents some condition that decides whether `S` is emitted. A *resolution* node has been inserted under each *emit S* node to inform the consumer about the presence of `S`. Additional *resolution* node has been inserted on the non-emitting side of the *test* nodes inside the potential emitters to decrement the number of potential emitters to `S`. These *resolution* nodes have completely covered all paths within their respective threads such that these threads would not be able to terminate without first hitting a *resolution* node. To protect against premature access to `S`, a *guard* has been inserted above the '*test S*' node. Assuming the compiler unwisely schedules the consumer thread before the potential emitters, the signal would be unresolved when this thread reaches the *guard* node. It then takes the right branch of the *guard* node and reaches a *termination* node with a value of ∞. This special termination code *falls through* the *join* node when threads join at the *join* node and the program exits the graph with a value of ∞. Due to this, the cyclic executive

```
1  module sketch:
2  input I;
3  signal S in
4      % ...
5      present S then
6          % ...
7      end
8      ||
9      present I then
10         emit S
11     end
12     ||
13     % potentially emit S ...
14     ||
15     % potentially emit S ...
16 end
17 end module
```

Figure 11: An sketch of an example Esterel program

detects the special termination code and reschedules the consumer thread in the next iteration.

When the compiler generates code from the GRC representation, the *guard* nodes are translated into counting semaphores around the shared signals. Each unique shared signal in the program has an associated counting semaphore. The counting semaphores are initialized to the number of potential emitters of their corresponding signal at the beginning of each *tick*. The counting semaphores are decremented by the *resolution* nodes in the potential emitters. The *guard* nodes are used for blocking the program when it attempts to read a signal with a non-zero counting semaphore. This locking and unlocking mechanism is exemplified by the generated code of the `sketch` example in Fig. 13.

The generated code implements the `sketch` Esterel module in a *reactive function* called `esterel_module`. The *reactive function* is called once every *tick*. The generated code presents the four threads of the `sketch` example as four functions. The actual implementation of the compiler would inline these functions into the *reactive function*. However, threads are shown as functions to aid the presentation of the program structure in the generated code.

The first thing the *reactive function* does is to initialize the program counters of the four threads. The program counters are used for resuming the threads when they are blocked by a signal *guard*. Following that, the signal *guard* is initialized by assigning the lock `lock_S` with the number of potential emitters on line 3. The name of the lock suggests that it is used for protecting the signal `S`. The cyclic executive is implemented by the loop on lines 5∼11. On every iteration, the loop condition checks the termination code. The loop will continue as long as the value of the termination code equals infinity. Within the loop body, threads execute one after the other in a sequential order. The returned termination code from each thread is combined at the end of each iteration. The combination of termination codes implements the *join* node in Fig. 12, and a value of infinity from a thread would always overide the termination code from other threads. The combined value is then checked by the loop condition on line 11 to *reschedule* the *blocked* threads. The implementation of the cyclic executive bears some resemblance of its abstract form in Fig. 3 of Section 2. The difference is that the abstract algorithm removes a thread from the vector when a thread dies; whereas the implementation of the algorithm in Fig. 13 controls the threads via their program counters. In the implementation, a thread would not execute if its program counter is zero and it would immediately return to the cyclic executive

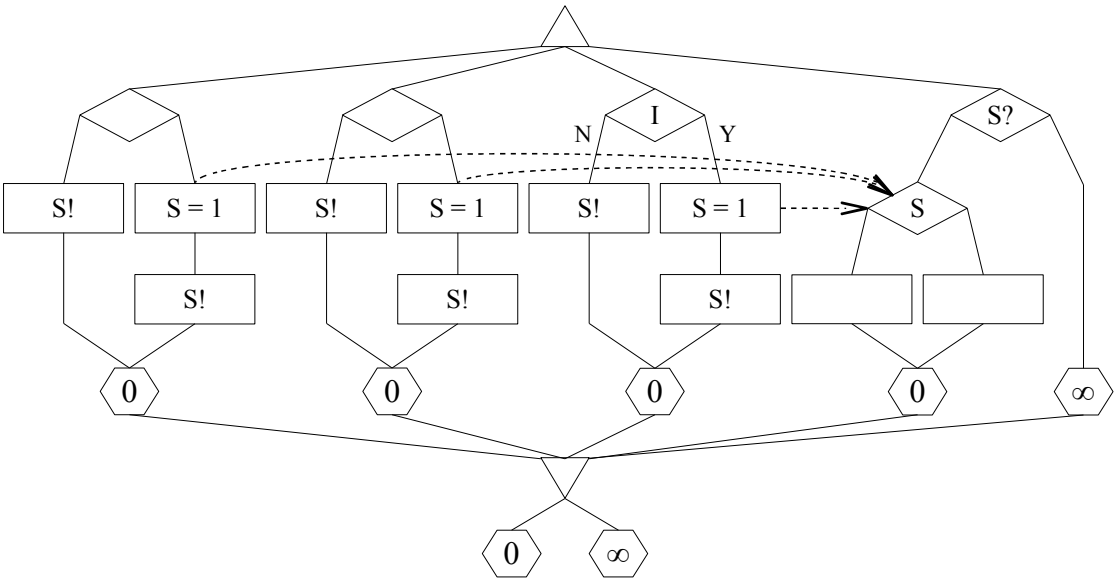

Figure 12: The GRC representation of the `sketch` example

in `esterel_module`.

To illustrate how signal locking and unlocking is performed, consider the consumer thread that has been deliberately scheduled before the potential emitters. The actual implementation of the compiler attempts with its best effort to schedule potential emitters before their consumers. In Fig. 13, when `thread_1` executes, `lock_S` has still not been released. This causes `thread_1` to store the program counter for resumption in the next iteration, and returns a value of infinity (lines 18∼20). Returning to `esterel_module`, the next thread `thread_2` gets scheduled. Depending on the status of the input signal `I`, `S` may or may not be emitted. If `S` is emitted, the code generated from the *resolution* node unlocks the *guard* node by setting the counting semaphore to zero. As soon as the signal guard is released, the dependency of the first thread is resolved. The first thread can then be freely rescheduled at any time within the *tick*. However, if the input signal `I` is absent, line 33 which corresponds to the *resolution* node on the else branch is executed and decrements the counting semaphore. With one potential emitter terminated and two remaining, the value of the counting semaphore drops to two and the signal guard stay locked. Assuming both `thread_3` and `thread_4` do not emit `S`, the counting semaphore is decremented twice by these two threads and becomes zero. When `thread_1` gets rescheduled on the second iteration of the cyclic executive, it resumes at the address pointed by its previous program counter. Then, it successfully get through the signal guard this time. Since the cyclic executive merely iterates through a static list of threads, threads that have already completed their *local ticks* are not removed from the list. For this reason, program counters are used to keep those threads at the completion state. Whenever a completed thread is rescheduled, it immediately returns.

3.4 Preservation of the scheduling algorithm

In Section 2, a sketch scheduling algorithm using run-time signal resolution has been presented. The preservation of the algorithm in Fig.3 are achieved by the following implementation of the

```
1 int esterel_module() {
2     thread_1_pc = thread_2_pc = thread_3_pc thread_4_pc = 0;
3     lock_S = 3; // 3 potential emitters
4     // The fork node spawns four threads
5     do {
6         term_1 = thread_1();
7         term_2 = thread_2();
8         term_3 = thread_3();
9         term_4 = thread_4();
10        term_0 = term_1 | term_2 | term_3 | term_4;
11    } while (term_0 == INIFINITY);
12 }
13 int thread_1() {
14    if (thread_1_pc)
15        goto *thread_1_pc;
16    // ...
17 RESUME:
18    if (lock_S) {
19        thread_1_pc = &&RESUME;
20        return INIFINITY;
21    } else {
22        // Inside guarded body
23    }
24    return 0;
25 }
26 int thread_2() {
27    if (thread_2_pc)
28        goto *thread_2_pc;
29    if (signal.I) {
30        signal.S = 1;
31        lock_S = 0;
32    } else
33        lock_S--;
34 END:
35    thread_2_pc = &&END;
36    return 0;
37 }
38 int thread_3() {
39    if (thread_3_pc)
40        goto *thread_3_pc;
41    // ..
42    lock_S--;
43    // ...
44 END:
45    thread_3_pc = &&END;
46    return 0;
47 }
48 int thread_4() {
49    if (thread_4_pc)
50        goto *thread_4_pc;
51    // ...
52    lock_S--;
53    // ...
54 END:
55    thread_4_pc = &&END;
56    return 0;
57 }
```

Figure 13: The generated code of the `sketch` example with run-time signal resolution

axioms presented in Section 2:

Axiom 1: The cyclic executive is implemented as a while-loop that encloses a set of threads ordered sequentially. Each thread is executed once in each iteration until the combined termination code from all threads result in a non-infinite value. If a thread does not communicate with any other thread, that thread will not interact with the signal locks in the generated code. When a thread terminates, the program counter remains at the end of the thread and would immediately return to the cyclic executive if rescheduled (lines 35, 45 and 55 in Fig. 13).

Axiom 2 Premature access to a protected signal is prevented by marking a signal as present due to emission, or absent when all potential emitters collectively agree that the signal can no longer be emitted in that *tick*. The number of unresolved signals is decremented by 1. This behaviour is achieved by resetting or decrementing the counting semaphore in the generated code, every time this condition holds (lines 31, 33, 42 and 52 in Fig. 13).

Axiom 3 A thread gets blocked from reading an unresolved signal is a candidate to be scheduled in the next iteration of the cyclic executive. In the generated code, the blocked thread saves its program counter and returns to the cyclic executive with a termination code of ∞ (line 20 in Fig. 13). The cyclic executive detects the blocked thread and the threads within the cyclic executive are rescheduled.

Hence, guarding shared signals with counting semaphores is a precise implementation of the run-time signal resolution algorithm sketch presented in Fig. 2. Given a causal Esterel program, every thread will have the opportunity to progress over each iteration of the cyclic executive. As threads progress, the number of unresolved signals decrease monotonically. Eventually, the number of unresolved signals converge to zero.

The previous sections and this section have discussed the dynamic scheduling approach using run-time signal resolution, provided a proof of its correctness, and the associated implementation. While the dynamic scheduling approach allows Esterel threads to execute in any order for any number of times, the macro behaviour of the program remain the same as a statically scheduled Esterel program. This flexibility elegantly decouples scheduling from thread distribution for executing Esterel programs on a multi-core. Thus, thread distribution can be solved as an independent research topic. The next two sections propose a static and a dynamic approach to distribute Esterel programs.

4 Static load distribution

The static distribution approach is the an attempt in this research to study the feasibility of executing Esterel on a multi-core. Threads are statically grouped together based on the amount of computation within each thread approximated by a heuristic guided algorithm. Each group is distributed to a dedicated core and no thread will be migrated between cores at run-time. We will describe the partitioning algorithm in the next subsection, followed by the implementation for the partitioned programs in section 4.2.

4.1 The static load distribution algorithm

The process of grouping threads together, called partitioning, takes place after the GRC of the program has been constructed. The GRC representation of the program is analyzed by a distribution heuristics to partition the nodes under the parallel branches of a *fork* node. When

the compiler generates the code from the GRC, each partition is enclosed in an cyclic executive such that threads within a partition is scheduled indenpendently of other partitions. As each processor execute its assigned partition in parallel, they synchronize at the each *tick* boundary. However, the barrier synchronization of the partitions at *tick* boundaries imposes some rules over how threads should be partitioned:

The atomicity of threads: A thread should not be divided to execute in multiple partitions. The whole thread should be allocated to one partition. Failing to follow this rule would incur performance penalties due to resolving control dependencies in addition to data dependencies.

The hierarchic ordering of threads: If a program $(p\|q)$ is distributed to partitions P1 and P2 respectively, any child thread forking from p should not be allocated to P2, and vice versa for q. Failing to follow this rule would incur performance penalties due to synchronization for starting forks in addition to synchronous joins.

Based on these rules, the distribution algorithm only distributes the threads spawned by the top level *fork* nodes in the root thread. An example of a partitioned program is illustrated in Fig. 14. The GRC representation shows the `ParallelData` example partitioned to execute on two CPUs. When the program starts, the root thread executes on the first CPU. As soon as the program reaches a *fork* node, the nodes shaded in gray are spawned on the second CPU, while the non-shaded nodes are spawned on the first CPU.

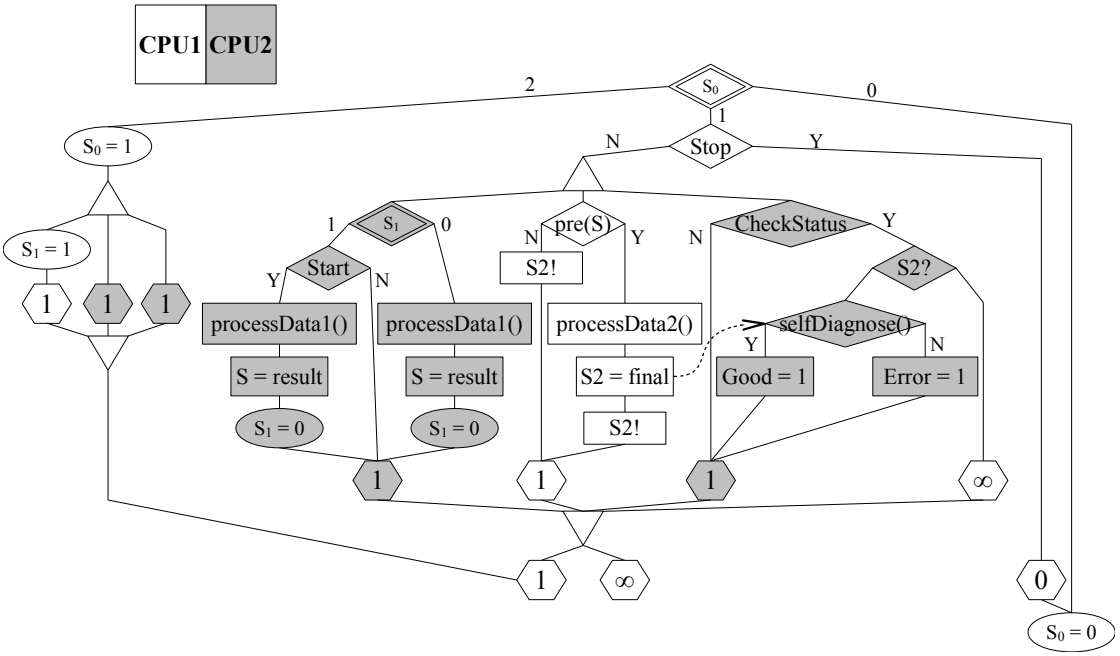

Figure 14: The partitioned GRC of the `ParallelData` example

Base on the amount of code the compiler would generate for each type of node in GRC, a relative cost is assigned to each type of node. The cost of a node is approximated by the number of instructions required for that node and normalized against the node with the least cost. For example, an *emit* node would in general have the lowest cost. It require only an assignment

operation in the generated code. Hence, an *emit* node would have a cost of one unit. For a *fork* node, an operation is required to initialize each thread it spawns. The cost of a *fork* is then approximately the number of threads it spawns. That is, for spawning four threads, the cost of the *fork* node would be four units. That is four times the cost of an *emit* node. The cost of a host procedure call is difficult to determine without exhaustive timing analysis. For nodes that call host procedures, the compiler makes an approximation of the cost by summing up the number of instructions of the host procedures. Other types of nodes have the same cost of one unit. Then, the partitioning algorithm computes the total cost of each thread by summing up the cost of each node within each thread under the top level *fork* node. Only top level *fork* nodes in the root thread are partitioned due to the rules described earlier.

After the cost of each node (represented by c in the algorithm presented in Fig. 15) in the GRC has been approximated, the `partition_dfs` algorithm in Fig. 15 starts partitioning using a depth-first search through the GRC. The algorithm distinguishes only *fork* and *join* nodes from the other types of nodes. More specifically, the algorithm looks for the top level *fork* and *join* nodes in the root thread. Other types of nodes are treated as ordinary nodes and their cost is added to the total cost of the thread being traversed. The identification of the top level *fork* is done on line 10. If a top level *fork* node, r, is found, lines 11 and 17 keep track of r until the costs of all branches under r have been estimated. Any *fork* node nested under r will cause the condition on line 10 to fail and the nested *fork* nodes are treated as ordinary nodes.

When a top level *fork* node is found, the loop on line 12 starts traversing the threads under r on a branch-by-branch (thread-by-thread) basis and assigns the reference of the root *fork* node to r. In the $r \neq 0$ state, the algorithm recursively traverses down each branch (thread) under r until it is stopped by the condition on line 8 when the corresponding *join* node of r has been reached. Before traversing through each thread under r, line 13 initialize a new cost c with zero for the thread about to be visited and adds c to C. Then, the algorithm recursively calls `partition_dfs` on the first node of each thread. For each node `partition_dfs` visits, the ordinary nodes would fail both conditions on line 8 and 10 and start from line 23. As C is not empty in the $r \neq 0$ state, the cost c_{node} of each *node* is added to the total cost c of the current thread being traversed on line 24. Then, the algorithm continues to recursively call `partition_dfs` on each control successor of *node* on line 26.

When all branches (threads) under r have been visited, the algorithm goes back to the $r = 0$ state on line 17 and cost is not calculated until the next top level *fork* node is reached. The reason cost is not calculated in the $r = 0$ state is due to the root thread being the only thread executing during this state. Then, the vector C, which represents the costs for each branch under r, is sorted in ascending order on line 18 before passing to the `load_balance` procedure. The `load_balance` procedure sorts the partitions based on their current associated costs. It then distributes each branch in C to the partition with the least cost in P. This is repeated until all branches in C have been distributed. Once `load_balance` returns to `partition_dfs`, C is cleared and remains empty until a *fork* node is encountered.

4.2 Generating code from GRC

To execute the partitioned code in parallel, the code may be generated either as POSIX threads that require an OS to run, or as custom code for a dual-core Xilinx Microblaze platform that can run without an OS. The number of thread partitions that is generated will depend on the number of processor cores that the execution platform offers. These thread partitions are created only once throughout the lifetime of the program. Both the OS and non-OS implementations are based on the same principle that will be described next.

Generating code from GRC for execution in parallel is an incremental step from the imple-

```
1  r denotes the top level fork in the root thread
2  c denotes the cost of the current branch being traversed
3  c_n denotes the cost of the current node being visited
4  C denotes a vector of costs associated with each branch under a fork
5  procedure partition_dfs(node)
6      if node is visited then return
7      add node to the visited set
8      if node = the corresponding join of r and r ≠ 0 then
9          return
10     else if node = fork and r = 0 then
11         r = node
12         foreach control successor s of node do
13             c := 0
14             add c into C
15             partition_dfs(s)
16         end
17         r := 0
18         sort C in ascending order
19         load_balance(C)
20         empty C
21         return
22     end
23     if C ≠ ∅ then
24         c := c + c_node
25     end
26     foreach control successor s of node do
27         partition_dfs(s)
28     end
29 end
```

```
1  p denotes the total cost of the partition
2  P denotes a vector of total costs associated with each partition
3  procedure load_balance(C)
4      foreach branch cost c ∈ C do
5          sort P in ascending order
6          assign the branch corresponding to c to the first partition p ∈ P
7          p := p + c
8      end
9  end
```

Figure 15: The distribution algorithm

mentation of a dynamically scheduled Esterel program, described in Section 3.3. We will use the same `sketch`[1] example presented in Fig. 11 to highlight the incremental changes. In 16, the `sketch` example has been divided into two partitions to executes on two processors. The shaded nodes execute on CPU2 and the non-shaded nodes execute on CPU1. The generated code corresponding to this partitioned GRC is presented in Fig. 17. Compared to the sequentially

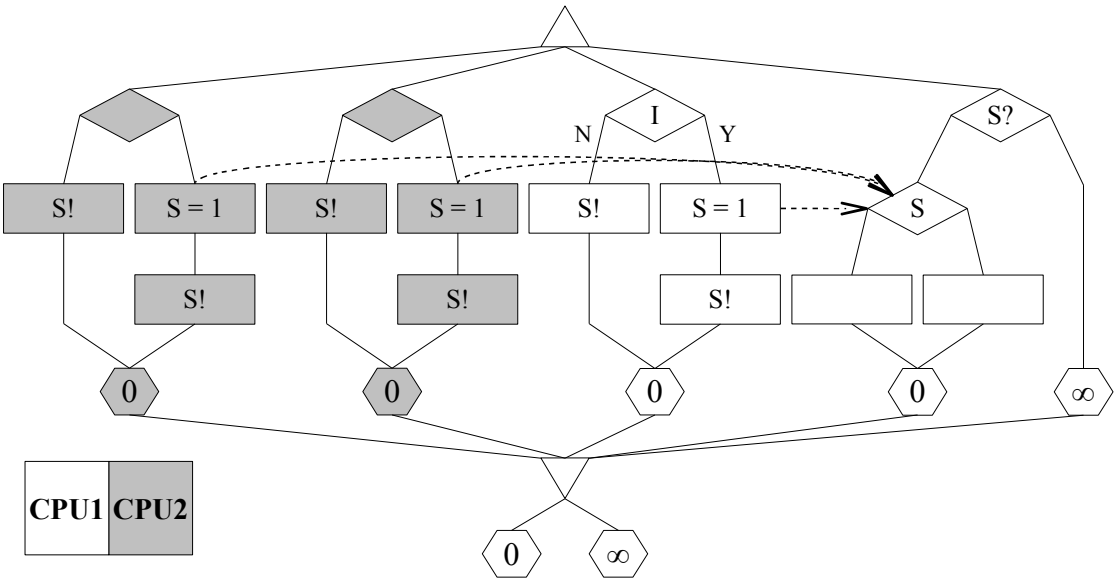

Figure 16: The partitioned GRC of the `sketch` example

executed version in Fig. 13, there are a few additions in Fig. 17:

- Coordination and synchronization between processors have been added.
- There are more than one cyclic executive such that each partition executes within its own cyclic executive.

Initially, only the first processor executes the root thread. The first processor therefore, is appointed as the *master* processor. The master processor is responsible for coordinating the *slave* processors by instructing them to start executing when the root thread forks. To allow multiple processors executing from the same binary, the *reactive function* of an Esterel program is translated into a *reentrant* function. The *reactive function* is then called from each processor. Subsequently, processors could share the same address space and communication between processors can be implemented using shared memory.

The *reactive function*, `esterel_module`, requires only one argument to indicate whether it is executing on a master processor or one of the slave processors. The slave processors are immediately blocked on line 4 until the master processor instructs them to start. When the master processor reaches the fork on line 11, it instructs the slave processors to begin execution by sending the starting addresses of the partitions allocated to the slave processors. For the master processor, it continues at the label `CPU0`, while the slave processor begins from the label

[1]Note that we present a more abstracted example than the example in Fig. 14 so as to be able to present the generated code (see Fig.17) within a one page limit.

```
1  int esterel_module(int isSlaveCPU) {
2      if (isSlaveCPU) {
3      WAIT_FORK:
4          wait_fork(pc); // Blocking
5          goto *pc;
6      }
7      thread_1_pc = 0;
8      thread_2_pc = 0;
9      lock_S = 3; // Three potential emitters
10     // ...
11     fork_cpu1(&&CPU1);
12     goto CPU0;
13 CPU1:
14     term_p1 = 0;
15     do {
16         term_1 = thread_1();
17         term_2 = thread_2();
18         term_p1 = term_1 | term_2;
19     } while (term_p1 == INIFINITY);
20     join_cpu1();
21     goto WAIT_FORK;
22 CPU0:
23     term_p0 = 0;
24     do {
25         term_3 = thread_3(); // Definitions of thread_3() and thread_4()
26         term_4 = thread_4(); // are omitted to conserve space
27         term_p0 = term_3 | term_4;
28     } while (term_p0 == INIFINITY);
29     join_all(); // Blocking
30     term_0 = term_p0 | term_p1;
31     // ...
32 }
33 int thread_1() {
34     if (thread_1_pc)
35         goto *thread_1_pc;
36     // ...
37 RESUME:
38     if (lock_S) {
39         thread_1_pc = &&RESUME;
40         return INIFINITY;
41     } else {
42         // Inside the guarded body
43     }
44     // ...
45 }
46 int thread_2() {
47     if (thread_2_pc)
48         goto *thread_2_pc;
49     // ..
50     lock_S = 0;
51     // ...
52 END:
53     thread_1_pc = &&END;
54 }
```

Figure 17: The generated code of the partitioned `sketch` example

`CPU1`. Both processors initialize their corresponding termination codes before the loop for the cyclic executive is entered.

The cyclic executive of each partition runs independently of each other at their own pace. Like the sequential version in Fig. 13, the termination code of each thread have to be combined when threads join. However, combining is a two step process in Fig. 17: once at the end of each partition on lines 18 and 27, and a second time after processors join on line 30. To synchronously join the two processors, a rendezvous is set up on line 29. This line blocks the master processor until all slave processors are ready to join. In this case, the single slave processor calls `join_cpu1` on line 20 to indicate that it is ready to join.

To illustrate how the semantics of instantaneous broadcast is preserved when threads execute in parallel (on different cores), the consumer thread of the signal `S` has been deliberately scheduled before the potential emitters of `S`. More importantly, the example in Fig. 17 also demonstrates the elegance of run-time signal resolution working across processors. When the master processor sends the starting address to the slave processor on line 11, the slave processor has a head-start of a few instructions while the master processors is being held back by the overhead of coordinating the slave processor. The master processor starts executing the third thread shortly after the slave processor started executing executing the first thread. At this point, `lock_S` is locked when the first thread attempts to read `S` on line 18 and the first thread returns with a termination code of infinity. Assuming `S` is not emitted by either the third or the forth thread, `lock_S` gets decremented twice. Let us also assume the second thread does not emit `S` when the slave processor execute it and `lock_S` becomes zero. The cyclic executive on line 19 detects that the first thread has been locked and reschedules it. Since `lock_S` is now released, the first thread is able to successfully terminate and join with other threads.

We summarize the static load distribution approach by highlighting the following salient features:

- Unresolved signals are protected from premature access by global signal locks in the generated code.
- Scheduling of threads within partitions are managed by multiple cyclic executives executing on each processor.
- Forking and joining of threads that are allocated to different processors are coordinated by the master processor. The slave processors wait until the master processor instructs them to start.
- All processors execute from the same compiled binary and have different starting addresses within the binary.

While the static load distribution algorithm attempts to produce balanced load with the best effort, the estimated load can still vary significantly at run-time due to parts of the program react to inputs from the environment. As environment cannot typically be pre-determined at compile time, the load at run-time cannot in general be precisely calculated statically. In order to tackle this problem, the dynamic load distribution approach is proposed and discussed next.

5 Dynamic load distribution

Traditionally, Esterel compilers sequentialize threads using a topological sort [10]. This approach provides little room for executing threads in parallel on a multi-core architecture and parallelizing threads in a balanced manner is difficult. This is due to the challenge of accurately estimating

the load on each core, and the difficulty of modeling environment non-determinism that makes static load distribution an approximation at best.

The dynamic load distribution approach refines the cyclic executive in Fig. 3 of Section 2 by converting the *reactive function* into a distributed reentrant function. The reentrancy nature of the function allows it to be safely called from multiple cores simultaneously.

The most salient feature of the dynamic load distribution approach is the ability to freely move a thread from one core to another when a thread is rescheduled. The two distribution rules of the static approach described in Section 4 do not apply to the dynamic approach. The key to the difference is that the cyclic executive is globally shared by all cores using the dynamic approach in contrast to the distributed cyclic executives used by the static approach. Using the static approach, distributing threads without keeping the hierarchical structure of the threads intact would require performance costly synchronization between cores for starting a fork in addition to joining threads. In contrast, due to the global vector T (line 1 in Fig. 18) being collectively managed by all cores, distribution of threads requires little overhead and is highly dynamic. This means threads can be distributed in any arbitrary way, as long as a parent thread is suspended when it is forked and one of its child threads resumes it when child threads join. See Section 2 for details of how the hierarchical thread structure is preserved.

The algorithm presented in Fig. 18 takes advantage of the flexibility of dynamic load distribution by distributing a thread to a core whenever a core becomes idle. In Fig. 18, the distributed version of the *reactive function* requires an argument to identify whether the function is called from the *master* processor. The master processor is only responsible for starting each *tick* at the start of the root thread and return from the root thread at the end of each *tick*. While the master processor starts executing the root thread on line 7, the slave processors remain idle on lines 9∼11 busy-waiting for the thread vector T (global to all cores) to become populated. As soon as the root thread forks, all processors enter the cyclic executive on line 14. As the processors execute each thread in T, they interact with the cyclic executive in the following ways:

Forks into more threads — saves the executing thread in a buffer f, then add all child threads of the fork to the back of T. To ensure all child threads terminate synchronously, each child thread has to check the the number of remaining child threads alive before terminating. The last child thread to terminate is responsible to resume and reschedule its parent thread by adding f to the back of T.

Attempts to read a signal — returns ∞ if the attempt to read failed due to unresolved signal when $n_s \neq 0$. Otherwise, the thread proceeds to read the signal and continue execution normally.

Emits a signal — emits the signal s and reset the number of potential emitters by letting '$n_s := 0$'.

Rules out emission of a signal — removes the thread as a potential emitter to s by decrementing $n_s := n_s - 1$.

As long as T is not empty, the processors will continue to execute within the cyclic executive and threads are able to execute on any processor and in any order. This highly dynamic execution scheme highlights the elegance and flexibility of the dynamic scheduling and the new load distribution approach. Due to the dynamicity, which processor will be the last processor to exit the cyclic executive cannot be pre-determined. To ensure that the root thread is always executed on the master processor, line 21 is required to look for the master processor. The slaves processors will enter idle state waiting for the next fork to occur in the root thread while the master processor executes the root thread.

```
1  T denotes a vector of active threads as candidates to be scheduled
2  function reactive_function(isMaster)
3      if isMaster = true then
4          foreach signal s shared by threads in T do
5              n_s := the number of potential emitters of s
6          end
7          start executing the root thread
8      else
9          while T = φ do
10             do nothing
11         end
12     end
13     forever
14         while T ≠ φ do
15             select and remove the first thread t ∈ T
16             term_code := t()
17             if term_code = ∞ then
18                 add t to the back of T
19             end
20         end
21         if isMaster = true then
22             continue to execute the root thread
23         else
24             while T = φ do
25                 do nothing
26             end
27         end
28     end
29 end
```

Figure 18: The reentrant version of the high level cyclic executive with run-time signal resolution

The next subsection will use a small example to visualize how the distributed *reactive functions* execute in parallel.

5.1 High level representation of dynamic thread distribution

The implementation of the distributed version of the *reactive function* consists of two aspects: (1) thread distribution and (2) run-time signal resolution. We will first describe the implementation of thread distribution using the `sketch` example and describe the implementation of run-time signal resolution in the next subsection. The GRC representation of the example is the exactly the same as Fig. 12 and it is reproduced in Fig. 19 with colours to aid the discussion of thread distribution. The thread vector T in Fig. 18 is implemented as a thread queue using a ring-buffer.

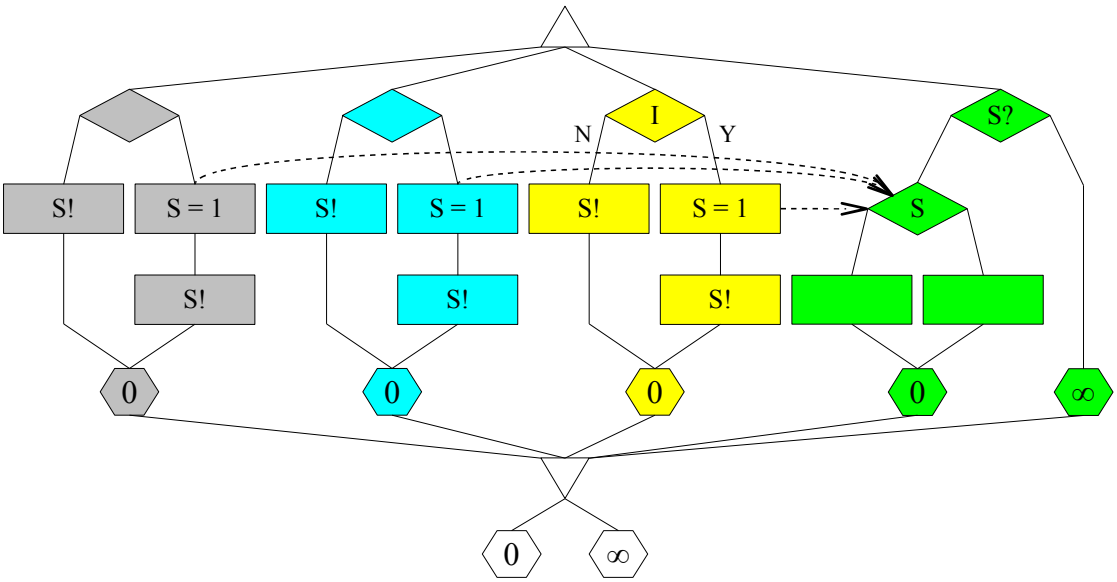

Figure 19: The GRC representation of the `sketch` example

The ring-buffer is initially empty and only the first processor is executing the root thread. The second processor remains idle awaiting threads to be inserted into the ring-buffer.

Let us assume the `sketch` example is executing on a dual-core processor. The initial state is depicted in Fig. 20(a). The white segmented ring in Fig. 20(a) represents the ring-buffer while the two blocks depicted at the center of the ring-buffer represent the statuses of the processor cores. The gray `CPU2` label indicates that the second processor is currently idle. When the root thread reaches the *fork* node in Fig. 19, it spawns four threads by adding the those threads to the ring-buffer and saves the address of the root thread in a separate buffer. The buffer containing the root thread is depicted as the square labeled `T0` in Fig. 20(b). The populated ring-buffer consists of `T1` (green), `T2` (yellow), `T3` (cyan) and `T4` (gray). The colour of each thread correspond to the nodes in Fig. 19 shaded with the same colour.

While the first processor completes the fork process, the second processor fetches `T1` from the ring-buffer and starts executing. Shortly after the second processor started executing, the first processor fetches `T2` from the ring-buffer and starts executing in parallel. As the two processors execute, two threads remain in the ring-buffer, as illustrated by Fig. 20(c). Assuming `S` is not emitted by `T2`, `S` remains locked and `T1` is unable to proceed as it has been blocked from reading

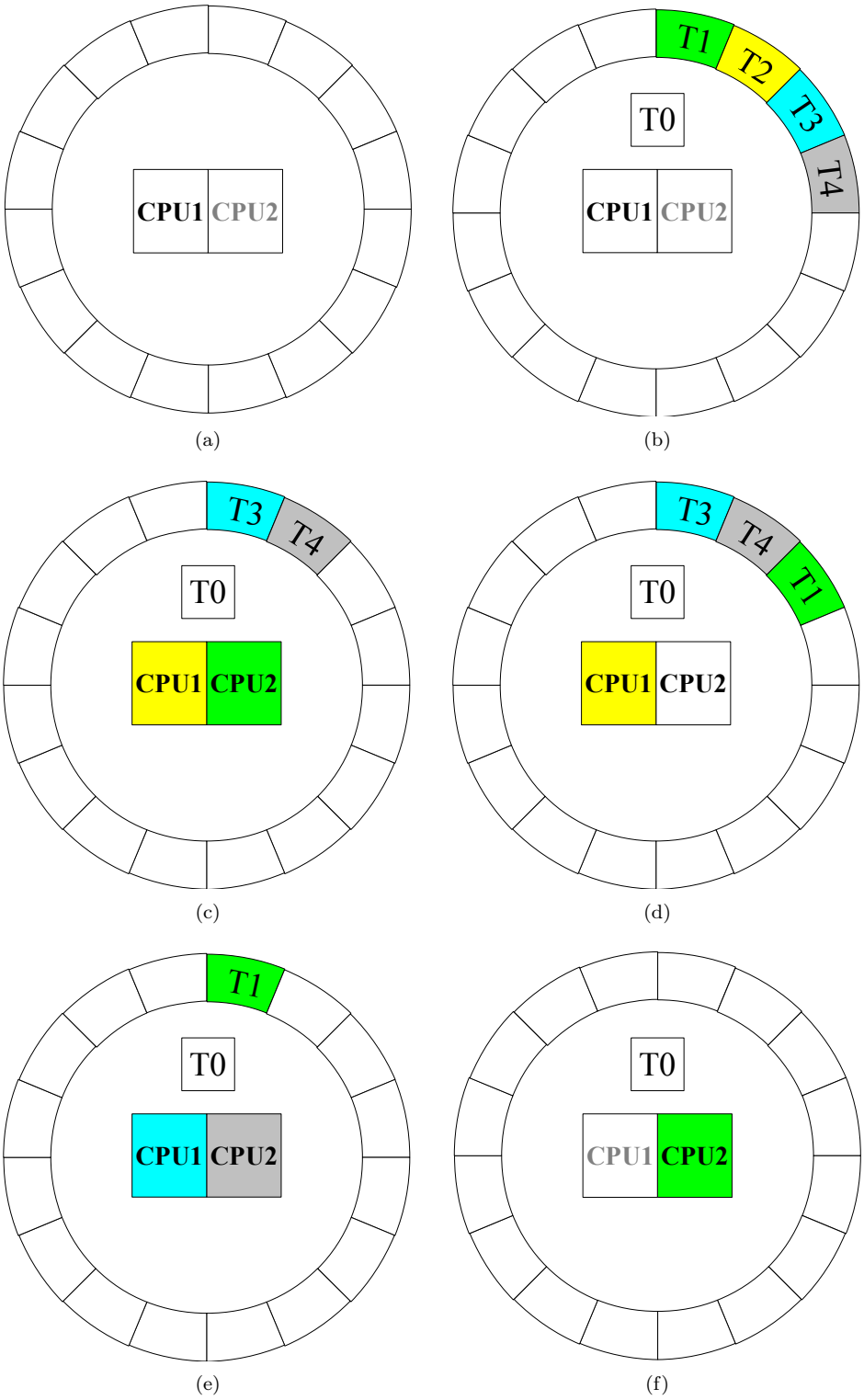

Figure 20: The time-line of the state of the thread queue and processors

`S`. The cyclic executive on the second processor detects the blocked thread and adds `T1` to the back of the ring-buffer.

The state of the ring-buffer is now represented by Fig. 20(d). The ring-buffer consists of `T3`, `T4` and `T1` and the second processor is about to fetch another thread from the ring-buffer. Then, the second processor fetches `T3` followed by the first processor fetching `T4` from the ring-buffer. The ring-buffer now has only `T1` as the two processors begin to execute `T3` and `T4`, as illustrated in Fig. 20(e). Assuming `S` is not emitted in either `T3` or `T4`, `T4` terminates normally followed by `T3` shortly after. Since the second processor completed `T4` before the first processor, it fetches `T1` from the ring-buffer and starts to execute. As the ring-buffer is now empty, the first processor has nothing to fetch and becomes idle. In Fig. 20(f), the second processor is the only active processor executing `T1`. With all the potential emitters completed without emitting `S`, `T1` is now unblocked and finishes normally.

When the second processor executes the join, it recovers the address of the root thread from the buffer saved during the fork and sends the address to the first processor. Then, the first processor is waken up by the second processor and resumes the root thread. Once the first processor takes over the control, the second processor becomes idle again until the next fork. Finally, the ring-buffer and the processors enter the same state depicted by Fig. 20(a).

The next subsection will describe the implementation of dynamic thread distribution and run-time signal resolution.

5.2 Implementation of dynamic thread distribution

Implementing an efficient thread queue is essential in order to not impede any performance gain. Typical operations on the queue involves appending to the tail of the queue and fetching from the head on a regular basis. Thus, implementing the queue as a ring-buffer would be more efficient than a linear buffer. Appending to the queue requires the following micro-steps:

- check if the queue is full
- place a thread after the item currently pointed by the tail pointer
- update the tail pointer

And popping a thread from the queue requires the following micro-steps:

- check if the queue is empty
- retrieve the thread pointed by the head pointer
- update the head pointer

Each micro-step would take a few instructions to implement in software. As a typical Esterel program frequently creates and destroys threads, accessing the thread queue on a regular basis would generate considerable overhead.

To minimize the overhead of managing the thread queue, the queue has been implemented in hardware. The hardware allows efficient mutual exclusive reading and writing to the queue. Simultaneous access to the queue will result in some cores being briefly blocked up to n clock cycles, where n equals to the number of cores accessing the queue simultaneously. The counting semaphore used for implementing run-time signal resolution is also implemented in hardware. The hardware based counting semaphore also works as a hardware mutex for atomic access to shared variables. Both hardware units are accessed through the memory bus. The interaction with the hardware is best illustrated using the example in Fig. 21 generated from the `sketch` example in Fig. 19.

```
1  int esterel_module(int coreID) {
2      if (coreID)
3          goto *FIFO_BASE;
4      else {
5          MUTEX_SETKEY = &lock_S;
6          MUTEX_LOCK = 3; // Three potential emitters
7      }
8      _thread_0_count = 4;
9      FIFO_BASE = &&THREAD_1;
10     FIFO_BASE = &&THREAD_2;
11     FIFO_BASE = &&THREAD_3;
12     FIFO_BASE = &&THREAD_4;
13     goto *FIFO_BASE;
14 THREAD_1:
15     // ...
16 RESUME:
17     MUTEX_SETKEY = &lock_S;
18     if (MUTEX_UNLOCK) {
19         FIFO_BASE = &&RESUME;
20         goto *FIFO_BASE;
21     } else {
22         // Inside guarded body
23     }
24     // ...
25     MUTEX_SETKEY = &_thread_0_count;
26     MUTEX_LOCK = 1;
27     _thread_0_count--;
28     MUTEX_UNLOCK = &_thread_0_count;
29     if (_thread_0_count)
30         goto *FIFO_BASE;
31     else
32         goto THREAD_0;
33 THREAD_2:
34     // ..
35     MUTEX_UNLOCK = &lock_S;
36     // ..
37     MUTEX_SETKEY = &_thread_0_count;
38     MUTEX_LOCK = 1;
39     _thread_0_count--;
40     MUTEX_UNLOCK = &_thread_0_count;
41     if (_thread_0_count)
42         goto *FIFO_BASE;
43     else
44         goto THREAD_0;
45 THREAD_3:
46     // Omitted to conserve space
47 THREAD_4:
48     // Omitted to conserve space
49 THREAD_0:
50     // ...
51     if (coreID == 0)
52         // ...
53     else {
54         FIFO_BASE = &&THREAD_0;
55         goto *FIFO_BASE;
56     }
57 }
```

Figure 21: The `sketch` example illustrating thread queue access in the generated code

0	THREAD_1
1	THREAD_2
2	THREAD_3
3	THREAD_4
N	...

(a)

&lock_S	3
	0
	0
	0
...	0

(b)

Figure 22: Content of the (a) thread queue; and (b) mutex

In the generated code, `FIFO_BASE`, `MUTEX_SETKEY`, `MUTEX_LOCK` and `MUTEX_UNLOCK` are memory mapped registers. These registers provide access to the thread queue and the mutex in the hardware. A thread can be added to the thread queue by writing the starting address of the thread to `FIFO_BASE`. A thread can be retrieved from the thread queue by reading `FIFO_BASE`. Reading this register effectively removes a thread from the queue. Similarly, the mutex hardware can be accessed by reading or writing to its memory mapped registers. The mutex hardware is implemented as a map that stores key-value pairs representing pairs of lock and its semaphore count. The mutex has three registers for setting the protected memory address, setting the semaphore count, decrementing and retrieving the semaphore count. The first register is `MUTEX_SETKEY`. It is a write-only register for setting the memory address to be protected. The second register is `MUTEX_LOCK`. It is also a write-only register. It initializes the counting semaphore to the value written to the register. The third register is `MUTEX_UNLOCK`. It is a read-write register that decrements the semaphore count when a memory address is written to it. The current value of the counting semaphore can be obtained by reading the `MUTEX_UNLOCK` register.

For example, the content of the hardware thread queue and the mutex are presented in Fig. 22. When the master processor reaches line 13 in Fig. 21, the thread queue contains four threads and the hardware mutex holds a single lock. The address of `lock_S` stored in the mutex maps to a value of three. This value is either decremented by the potential emitters (line 35 in Fig. 21) or reset to zero when the signal is emitted. Resetting a semaphore count works he same way as initializing it with a value, such as line 6 in Fig. 21. The size of the hardware queue and the mutex is parameterizable and are limited by the hardware resources available.

Let us assume the `sketch` example is executing on a dual-core processor and follow the same execution trace as Fig. 20. Both cores start executing by calling the *reactive function* – `esterel_module`. Each core is distinguished by its unique ID, which is passed as an argument. The first core has the ID zero and acts as the master processor while the second core becomes the slave. While the first core initializes the counting semaphore of `S`, the second core is immediately blocked upon reading the `FIFO_BASE` register on line 3 due to an empty thread queue. The first core then starts executing the root thread from line 9. This line, which corresponds to the *fork* node in Fig. 19, begins by initializing the thread count and add the starting address of each thread to the thread queue. The implementation of the fork process differs slightly to the high level representation of thread distribution in Fig. 20. Instead of saving the address of the root thread into a separate buffer, a thread count is initialized with the number of child threads being spawned. Since the continuation address when the root thread resumes is known at compile-time, the address is simply referenced by the label on line 49. However, the thread count is required to track the number of active threads spawned by the fork.

As soon as the a thread is added to the queue, the second core becomes unblocked and attempts to fetch a thread from the queue on line 3. As the second core is accessing the queue in parallel to the first core, one of them will be blocked by the other for a clock cycle. Assuming the second core gets the access to the queue, it successfully fetches `T1` and starts executing from

line 34. Meanwhile, the first core completes the fork process and fetches `T2` from the queue. Assuming `S` is not emitting by `T2` executing on the first core, the semaphore count of `S` is decremented by `T2` on line 35 and two potential emitters remain. On line 18, the second core attempts to read `S` but fails. It then adds `T1` back to the queue on line 18. At this point, the current state of the queue corresponds to Fig. 20(d). The second core then fetches `T3` and starts executing from line 46. Shortly after the second core started executing `T3`, `T2` executing the first core terminates normally and decrements the thread count on line 39. As three more threads are still active, the first core fetches `T4` from the queue and starts executing from line 48. The current status of the queue is now represented by Fig. 20(e).

Assuming neither `T3` nor `T4` emit `S`, no more potential emitters remain and `S` is unlocked. The second core completes `T4` normally and fetches `T1` from the queue and starts executing from line 17. In the meanwhile, the first core also finishes executing `T3` and it sees one thread is still active indicated by `_thread_0_count`. The first core attempts to fetch another thread but gets blocked by the empty queue. The states of the cores and the queue now correspond to Fig. 20(f).

As `S` is now unlocked, `T1` successfully completes and decrements the thread count. Since no more thread is active, the second core jumps from line 32 to line 49 and resumes the root thread. However, the root thread has to execute on the first core, the second core passes the control to the first core by adding the resumption address to the queue on line 54. Finally, the first core successfully takes over control and continues from line 52.

We summarize the elegance and the flexibility of the dynamic thread distribution approach by highlighting the following salient features:

- Unresolved signals are protected from premature access by signal locks implemented as hardware counting semaphores.
- Scheduling of threads are collectively managed by a single cyclic executive shared by all processors. There is little overhead from moving a thread from one processor to another as other processors are able to fetch threads from the globally shared hardware queue directly.
- Forking threads can be performed by any processor whenever the processor's assigned thread forks. The newly spawned child threads are dynamically added to the global thread queue and the parent thread is suspended by leaving it out from the thread queue. Joining threads is done by whichever processor that happens to execute the last terminating child thread. The resume address of the parent thread is known at compile-time and the parent thread is waken up when scheduled by adding to the parent thread to the queue.
- All processors execute from the same compiled binary and have different starting addresses within the binary.

6 Experimental results

To evaluate the proposed approaches, a set of mixed control and data dominated programs have been selected for benchmarking. These programs are presented in Table 1.

The first column in Table 1 lists the names of the programs. The number of lines of Esterel code is shown in the second column and the third column shows the number of lines of C code. The C code implements host procedures that are called from Esterel. The fourth column shows the maximum number of threads that can potentially be executing in parallel.

The first program *ABIC* is the smallest example of all with only 21 lines of code. This program was created to test how well our approach works with small Esterel programs that contain no data computation. The second program, *MCA200*, is taken from [9]. It is a program that implements

Name	LOC (Esterel)	LOC (C)	Threads
ABIC	21	0	3
MCA200	4956	0	55
WW (Estbench)	1087	676	22
WW09	317	0	7
LiftController	272	230	12
LZSS	42	462	5
Life	74	105	4
Mandelbrot	168	160	9

Table 1: A list of benchmarking programs

a shock absorber controller. It is also the largest control dominated Esterel example listed here, with no data computation. *WW (Estbench)* is a wristwatch example also taken from [9]. It is the largest program of all with considerable amount of data computation through C function calls. *WW09* is an alternative implementation of wristwatch developed from scratch using purely control statements. This example does not contain any data computation. *LiftController* is a design based on Vahid and Givargis's example [16]. The design has both control and data parts. *LZSS* is a text archiver using the Lempel-Ziv Storer Szymanski *(LZSS)* algorithm. This program contains the largest amount of data computation in C code of all the examples shown here. The *Life* example is a program that simulates the evolution of the program determined by its initial state. It is adapted from the MPI version [1] into Esterel. This example performs large amount of arithmetic computation over a two-dimensional array. *Mandelbrot*, adapted from [2], is a program that computes a mandelbrot set. The data computation involves floating point calculation and this type of computation is used to simulate a typical DSP application.

In the following subsections, two types of implementation of load distribution will be benchmarked. The first set of benchmarks evaluate the static load distribution approach. We will also perform the experiments with and without an OS to study the effects of an OS on the performance. The second set of benchmarks evaluates the dynamic distribution approach. We will proceed by describing the experimental environment in the next subsection.

6.1 Embedded multi-core architecture for benchmarking

To evaluate the performance of executing Esterel on a multi-core without an OS, an embedded multi-core architecture has been developed for benchmarking. The architecture consists of four Xilinx Microblazes. Each core has its own local on-chip memory for fetching instructions and storing local data. Typically, the instruction memory port and the data memory port on a Microblaze processor are both connected to the same dual-port, on-chip memory. For a dual Microblaze system, the dual-port memory can be used as a shared memory between the two cores for communication purposes. When the on-chip memory is connected to the Local Memory Bus (LMB), accessing the memory takes only a single clock cycle. This still holds when both of the memory ports are being accessed at the same time. The fast memory access is essential for a multi-core architecture. Both cores can access the local memories and the shared memory within one processor clock cycle. For more than two cores, the dual-port memory needs to be multiplexed. As an example, a quad-core system using the multiplexed design is illustrated in Fig. 23.

The quad-core system consists of separate Local Memory Buses for each core. The local memory and the shared memory are separated in the address space as illustrated by Fig. 23(b). On processor start up, both the local memory and the shared memory in the system are pro-

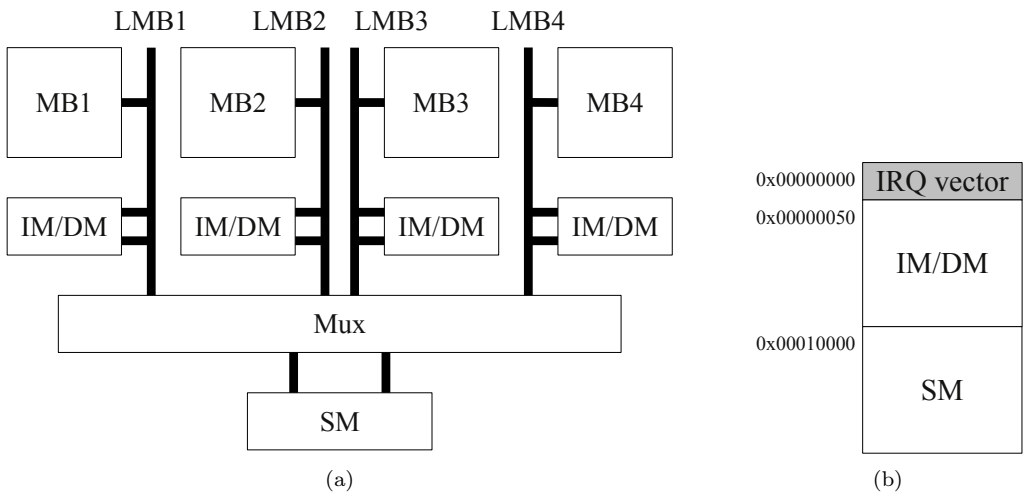

Figure 23: Memory architecture of a quad Microblaze system: (a) the memory architecutre; (b) the memory address space

grammed with the same program executable. Each core fetches instruction and can access local data from its own local memory, depicted as IM/DM in Fig. 23(a). The cores are multiplexed in round robin fashion when simultaneous access to the shared memory is requested. The maximum number of clock cycles to access the shared memory directly corresponds to the number of cores requesting for access.

In order to benchmark the scalability of the architecture, a Microblaze multi-core simulator was designed by adapting [17]. The simulator is able to simulate an arbitrary number of cores. The simulator is used to benchmark a hypothetical quad Microblaze system, where synthesizing this architecture would put too much constraint on the amount of on-chip memory available for each core.

6.2 Test setup

For evaluation purposes, the experiments were performed on two platforms. To test the POSIX thread based implementation running on a commodity multi-core, a desktop PC powered by a 2.4GHz Intel Core 2 Quad processor was used. The programs were compiled with GCC 4.5.0 and executed in Linux. To find out the performance on an embedded system without an OS, the Microblaze simulator is used to simulate a dual-core architecture. The reason for simulating only a dual-core will become apparent once the experimental results are presented.

The performance of the programs running on the desktop were measured using a pair of `clock_gettime()` C library calls. The time measured corresponds to how long each program took to complete the given input trace.

6.3 Performance of static load distribution

The results for static load distribution reveal that certain class of Esterel programs (Class C) are not amenable to speedup on multi-core platforms. The execution time of these programs are shown in Table 2. From the table, one can notice that control-dominated programs, do not

Table 2: Performance comparison on PC

Examples	1 core	2 cores
ABIC	3.5ms	4ms
WW (Estbench)	357ms	410ms
WW09	349us	347us
LiftController	2.57ms	3.06ms

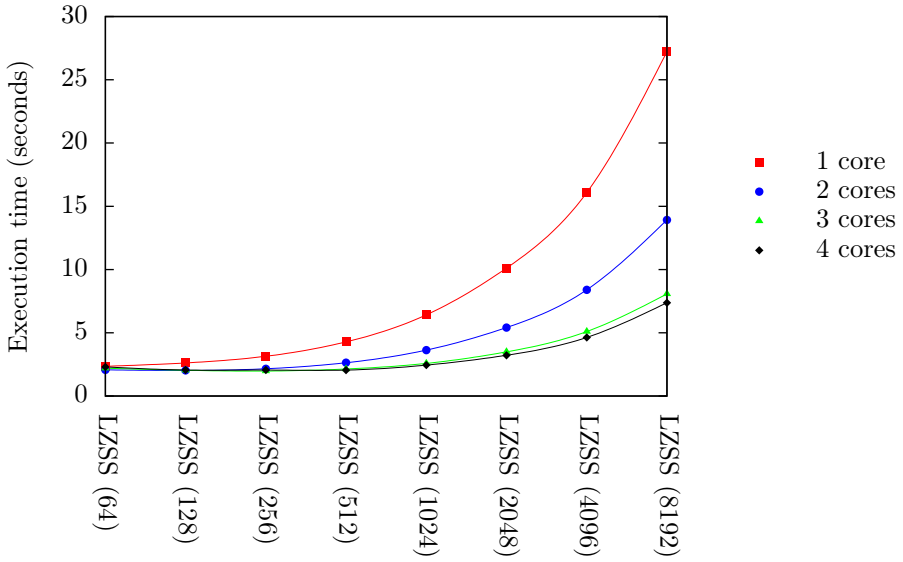

Figure 24: Performance comparison on PC (the lower the better)

benefit from the POSIX thread implementation running with an OS. The lack of performance gain can be attributed to the following:

- The Esterel scheduling overhead outweighs the actual computation performed within each Esterel thread. This happens when the Esterel threads found in a program are too small.
- Due to the unevently balanced load, leading to the lack of performance improvement.

The *LZSS* example, a heavily data dominated application has been selected as the main benchmark to run on the PC. *LZSS* has a parameter that sets the dictionary size of a text archiver, which uses the Lempel-Ziv Storer Szymanski algorithm [15]. The size of the dictionary affects the length of the data computation within a *tick*. The dictionary size can be practically set between 64 bytes to 8192 bytes. Beyond 8 kilobytes, the program would take too long to complete within a reasonable time-frame. The number of cores *LZSS* uses can be controlled by setting the number of partitions used in the compiler. Using the quad-core system described in Section 6.1, *LZSS* can be distributed up to a maximum of four partitions. The plot in Fig. 24 shows that *LZSS* will benefit from multi-core execution as the dictionary size increases. When running with all four cores, the maximum speedup achieved in the experiment is 3.7 times compared to a single core.

To find out how well the static load distribution works for an embedded system without an OS, the same examples were executed on a Xilinx dual-core MB system without any OS. The performance was measured using hardware clock counters. The counters are controllable from

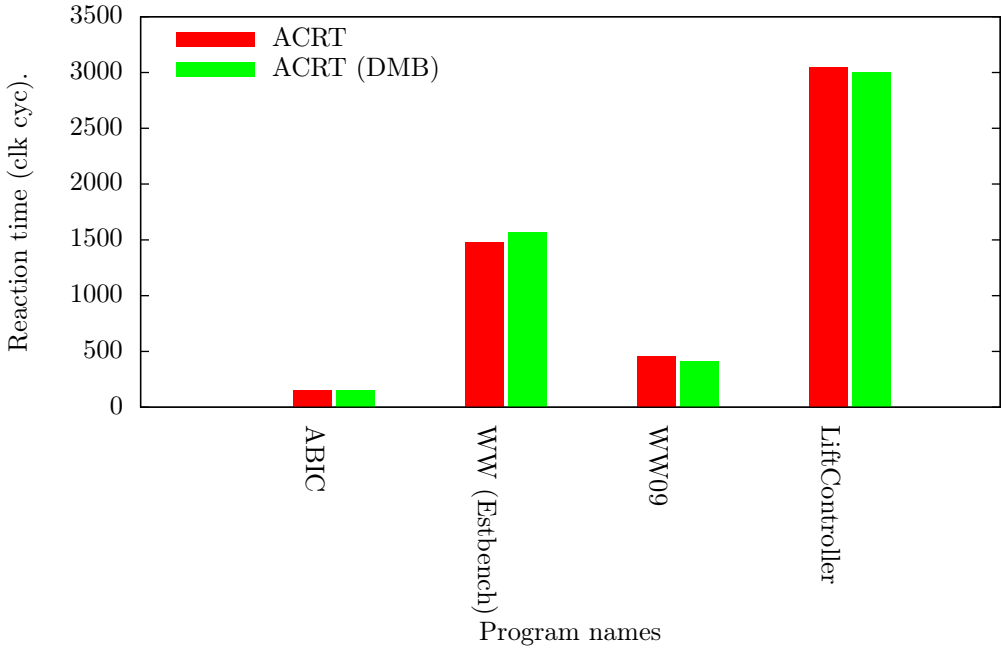

Figure 25: Performance comparison on dual-core Microblaze (lower the better)

their associated processors. Both counters are synchronized with the processor clock. Performance measurement requires the use of only the counter on the master processor. The counter on the slave processor is only used when measuring the load distribution.

The performance is compared in terms of reaction time of the tick length. With the precision offered by the counters, the reaction time of each program can be measured with an accuracy of the exact number of processor clock cycles it took to execute. The counter on the master processor is started just before it enters the *reactive function*, and stopped right after the *reactive function* returns. The difference between the two clock counts gives the reaction time of the current *tick*. The reaction times over the number of *ticks* taken to complete the input trace is then summed up. The average case reaction time (ACRT) can then be obtained by dividing the sum by the number of *ticks*.

Fig. 25 compares the ACRT of the programs when running on a single core with that on a dual-core (labeled with DMB). Having seen the results of the POSIX thread based implementation in Table 2, it is not surprising to see *ABIC* and *Wristwatch (Estbench)* performing worse. With bigger threads, examples such as *Wristwatch09* and *LiftController* were able to perform slightly better with two cores.

Again, the *LZSS* example is benchmarked with a range of dictionary sizes. The trend of the ACRT in Fig. 26 closely resembles the plot in Fig. 24. Overall, the speedup obtained from multi-core execution without an OS is slightly better than execution with an OS. The *LiftController* example showed speedup in Fig. 25, whereas the same example ran slower in Table. 2. The overhead from the threading library and the OS has caused the `LiftController` example to run slower with an OS.

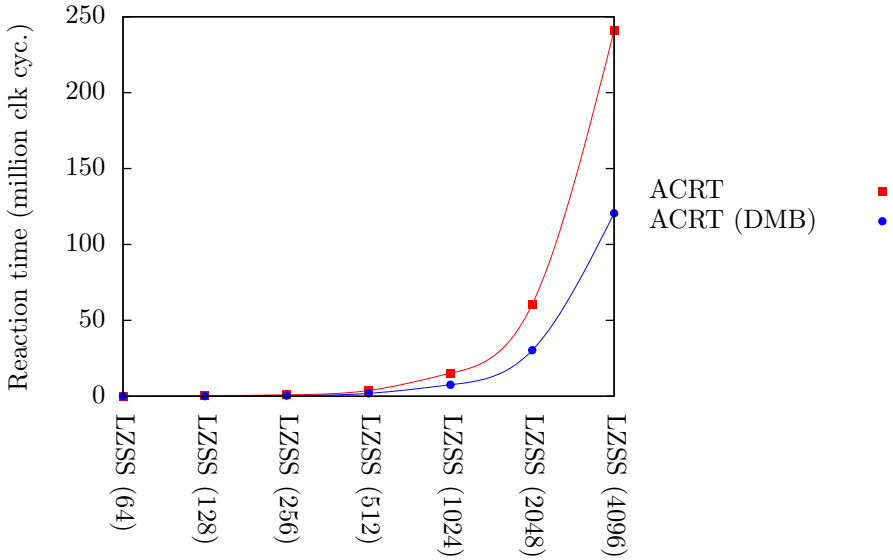

Figure 26: ACRT comparison of *LZSS* only on dual-core Microblaze (lower better)

Table 3: Comparison of processor idle time in %

Examples	Max (%)	Min (%)	Average (%)
ABIC	52	48	50
WW (Estbench)	72	1	40
WW09	49	9	34
LiftController	96	94	94
LZSS	0	0	0

6.4 The load balance of static load distribution

To evaluate the load balance, the maximum, minimum, and average processor idle time between each *fork-join* pair on the dual-core Microblaze were measured. The percentage shows the relative idle time of a core when a *fork* occurs. The results are presented in Table 3. The most balanced program is *LZSS* due to its computation-intensive nature. The data can be segmented perfectly for balanced execution in parallel. The least balanced program is *LiftController*. The data computation in this example is concentrated in one thread, while the remaining threads only perform control. *WW (Estbench)* and *WW09*, having more threads, were able to achieve a single digit percentage in processor idle time in the best case, but achieved 72% and 49% in the worst case respectively. On average, *WW (Estbench)* has a processor idle time of 40%, while *WW09* is slightly better with 34%. *ABIC*, the smallest example, has about 50% in processor idle time in all cases.

These results illustrate that achieving speedup with multi-core execution of Esterel is dependent on several factors. One of the key factors is the computation to communication ratio which is determined by the amount of dependency between threads on the different cores. The heuristic guided distribution tries to minimize this signal dependency. However, another important factor is the effect of such distribution on the processor idle time. As can be observed, the distribution is not designed to minimize the amount of time a processor remains idle since it is heuristic guided. Improving this will be a key to achieving more consistent and better speedup of the

distribution algorithm. By intuition, if a core becomes idle, it should ideally *steal* more work to do. This would require load distribution to be done at run-time. Hence, the dynamic load distribution approach has been developed. The evaluation of this approach is described next.

6.5 Performance benchmark of dynamic load distribution

The experimental results obtained from the dynamic load distribution approach shows big contrast compared to the static heuristic based load distribution. All benchmarked programs showed speedup across the board, as shown in Fig. 27

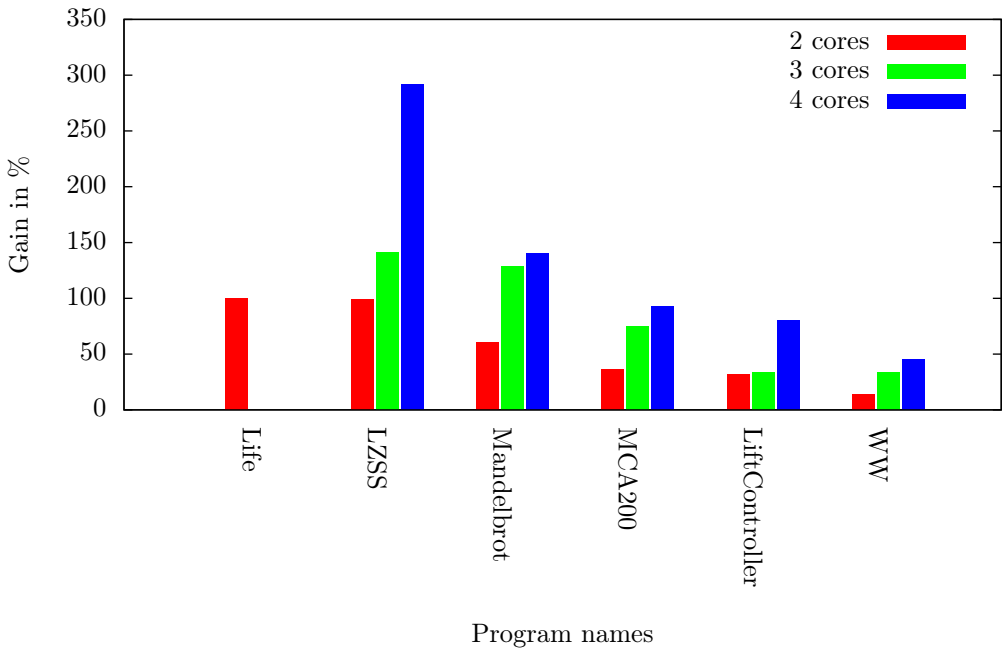

Figure 27: Speedup of the average case reaction time

The first example, *Life*, spawns only two threads throughout the program. Therefore, there are no results for tri-core and quad core experiments. It did, however, benefit from a dual-core with a performance gain of 99%. The *LZSS* and *LiftController* example showed interesting results for three cores compared to four cores. This ambiguity maybe explained as follows. There are four threads of identical load spawned by the program. When executed with three cores, three of these threads will be completed almost at the same time. The remaining thread will be picked up by whichever core completed its work first, leaving the other two cores idling. However, executing on four cores, all threads are evenly distributed and completed at the same time, resulting in minimal processor idle time. All other examples show varying degrees of performance gains from two to three cores. On average, the ACRT performance increased by 57% with two cores, 82% with three cores, and 130% with four cores.

The worst case reaction time (WCRT) measured from the longest *tick* length during the execution of each program is shown in Fig. 28. The figure illustrates a similar trend compared to the ACRT. However, upon careful inspection of the results for the WCRT, one can still observe that the examples benefit more from four cores than three cores. The WCRT performance on

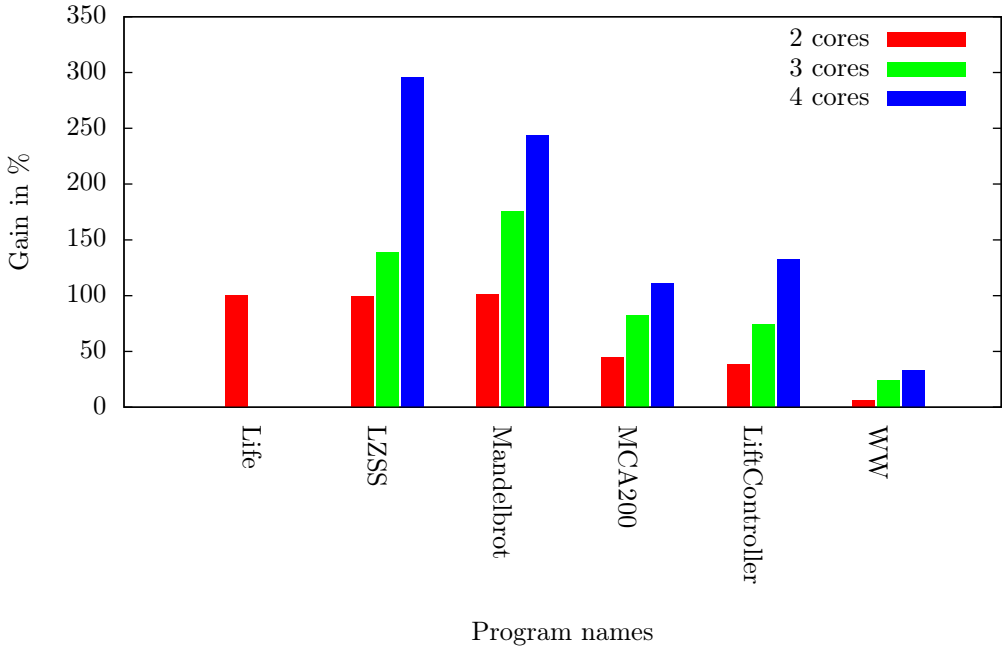

Figure 28: Speedup of the worst case reaction time

average increased by 65% with two cores, 99% with three cores, and 163% with four cores.

Overall, all examples showed performance gains from multi-core execution. Obviously, the data dominated examples benefited the most, while examples with less data computation did not gain as much. Interestingly, for a pure control program, if the program is large enough with many threads, the program may still have a good potential to perform better with a multi-core processor. The *MCA200* is a pure control program that nicely demonstrated significant benefits using the dynamic load distribution approach.

7 Conclusions

A key to effective parallelization of Esterel programs is a scheme for resolving the status of signals at run-time. Another essential element is a method for load balancing that tries to minimize communication between the cores. This paper introduced two approaches that address both these questions with a compiler that can either (1) automatically partition threads to cores, so as to minimize signal based communication between cores; and (2) dynamically distribute threads at run-time to reduce the processor idle times. These two highly dynamic techniques highlight the novelty of the proposed approach.

In contrast to the static approach, our compiler introduces the concept of signal locks to resolve signals at run-time. A signal remains locked until it is resolved to be either present or absent in every *tick*. The locking mechanism introduced in this paper can compute signal absence, and preserves reactivity and determinism of any causal Esterel program while doing so. The benchmarking results reveals that static load distribution is effective for parallelizing Esterel programs involving sufficient data computation.

While the static load distribution algorithm worked well for data dominated programs, its

load distribution heuristics does not work so well for control dominated programs. The second approach presented in this paper overcomes the problems by dynamically distributing the load at run-time. The experimental results have shown that the technique has improved the performance substantially. The benefits of the dynamic distribution are as follows:

1. The technique is capable of scaling across any number of processor cores, and is limited only by the hardware resources available.
2. The technique does not require complex timing analysis of the work load.
3. The processor idle time is less sensitive to environment non-determinism compared to static distribution.

We have demonstrated the effectiveness of our techniques over a benchmark of control-dominated and data-dominated programs for parallel execution. The thread distribution techniques described in this paper are able to distribute both the control code and data of Esterel. In contrast, the distribution technique introduced by Caspi *et al.* [7] replicates the control code across a network of processors, distributing only data.

Compared to [3], the effectiveness of their approach to parallelizing an Esterel program is questionable as parallelism is relatively coarse-grain compared to our approach. The distribution technique introduced in that work depends on the number of concurrent execution paths without data dependencies. As the threads within an Esterel program are tightly coupled, the effectiveness of their technique would be severely limited. Moreover, distribution in that work is achieved through OpenMP, which relies on an OS. Our approach will work with and without an OS.

An interesting work introduced by Ju *et al.* [11] described an approach to estimate worst-case reaction time (WCRT) of Esterel programs running on multiprocessors. They have also adopted the run-time signal resolution technique described in [18]. The contribution of that work is unclear as the estimated WCRT is only reported for multi-core execution and did not compare to the single-core execution. In contrast, we shown the effectiveness of our approach through a set of benchmarking programs executed on both single-core and multi-core processors.

The dynamic scheduling technique described in this paper is the key to effective distributed execution of Esterel. Despite the dynamic nature of the scheduling algorithm, section 2 has proven that the approach is sound and correct. Without the flexibility of the scheduling algorithm, load distribution would be tightly coupled the scheduling problem. Solving scheduling and load distribution at the same time would be far more complex, achieving speedups with parallel execution on multi-cores would be extremely difficult.

References

[1] A parallel implementation of *Life* using MPI. http://www.shodor.org/refdesk/Resources /Tutorials/PIExamples/Life.php. last accessed - 10.7.11.

[2] Mandelbrot Example. http://qt-project.org/doc/qt-4.8/threads-mandelbrot.html.

[3] D. Baudisch, J. Brandt, and K. Schneider. Multithreaded Code from Synchronous Programs: Extracting Independent Threads for OpenMP. In *Design, Automation and Test in Europe (DATE)*, pages 949–952, Dresden, Germany, 2010. EDA Consortium.

[4] G. Berry. Preemption in concurrent systems. In R. Shyamasundar, editor, *Foundations of Software Technology and Theoretical Computer Science*, volume 761 of *Lecture Notes in*

Computer Science, pages 72–93. Springer Berlin / Heidelberg, 1993. 10.1007/3-540-57529-4_44.

[5] G. Berry. *The Constructive Semantics of Pure Esterel (Draft Book)*. Available on-line, `http://www-sop.inria.fr/esterel.org` (Last Accessed: 28/4/2007), 1999.

[6] G. Berry and G. Gonthier. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[7] P. Caspi, A. Girault, and D. Pilaud. Automatic Distribution of Reactive Systems for Asynchronous Networks of Processors. *IEEE Trans. Software Engin.*, 25(3):416–427, May 1999.

[8] E. Closse, M. Poize, J. Pulou, P. Venier, and D. Weil. Saxo-rt: Interpreting Esterel Semantic on a Sequential Execution Structure. *Electronic Notes in Theoretical Computer Science*, 65(5):80–94, Jul 2002.

[9] S. A. Edwards. *EstBench Esterel Benchmark Suit.* `http://www1.cs.columbia.edu/~sedwards/software.html` (Last Accessed: 8/6/2007).

[10] S. A. Edwards and J. Zeng. Code Generation in the Columbia Esterel Compiler. *EURASIP Journal on Embedded Systems*, 2007:Article ID 52651, 31 pages, 2007. doi:10.1155/2007/52651.

[11] L. Ju, B. K. Huynh, A. Roychoudhury, and S. Chakraborty. Timing analysis of Esterel programs on general-purpose multiprocessors. In *Proceedings of the 47th Design Automation Conference (DAC)*, pages 48–51, New York, 2010. ACM.

[12] E. Lee. The problem with threads. *Computer*, 39(5):33–42, may 2006.

[13] G. E. Moore. Cramming more components onto integrated circuits. http://download.intel.com/museum/Moores_Law/Articles-Press_Releases/Gordon_Moore_1965_Article.pdf (last accesesd 17/06/2012), 1965.

[14] D. Potop-Butucaru and R. de Simone. Optimizations for faster execution of Esterel programs. In *Formal Methods and Models for Co-Design, 2003. MEMOCODE '03. Proceedings. First ACM and IEEE International Conference on*, pages 227–236, 2003.

[15] J. A. Storer and T. G. Szymanski. Data compression via textual substitution. *J. ACM*, 29(4):928–951, 1982.

[16] F. Vahid and T. Givargis. *Embedded System Design – A Unified Hardware/software Introduction.* John wiley and Sons, 2002.

[17] J. Whitham and N. Audsley. The Scratchpad Memory Management Unit for The Scratchpad Memory Management Unit for Microblaze: Implementation, Testing, and Case Study. Research Report YCS-2009-439, University of York, Real-Time Systems Group Department of Computer Science University of York, York, YO10 5DD, UK, 2009.

[18] L. H. Yoong, P. Roop, Z. Salcic, and F. Gruian. Compiling Esterel for Distributed Execution. In *International Workshop on Synchronous Languages, Applications, and Programming (SLAP'06)*, Vienna, Austria, March 2006.

[19] S. Yuan, S. Andalam, L. H. Yoong, P. S. Roop, and Z. Salcic. STARPro — A new multithreaded direct execution platform for Esterel. *Electron. Notes Theor. Comput. Sci.*, 238(1):37–55, June 2009.

[20] S. Yuan, L. Yoong, S. Andalam, P. Roop, and Z. Salcic. A New Multithreaded Architecture Supporting Direct Execution of Esterel. *EURASIP Journal on Embedded Systems*, 2009(1):610891, 2009.

Contents

1 An Esterel Example **4**

2 The run-time signal resolution approach **6**

3 Implementation of run-time signal resolution **11**
3.1 The GRC intermediate format . . . 11
3.1.1 Extensions to GRC . . . 13
3.2 The insertion algorithm for signal resolution nodes . . . 15
3.3 Generating code with run-time signal resolution . . . 19
3.4 Preservation of the scheduling algorithm . . . 21

4 Static load distribution **23**
4.1 The static load distribution algorithm . . . 23
4.2 Generating code from GRC . . . 25

5 Dynamic load distribution **29**
5.1 High level representation of dynamic thread distribution . . . 32
5.2 Implementation of dynamic thread distribution . . . 34

6 Experimental results **37**
6.1 Embedded multi-core architecture for benchmarking . . . 38
6.2 Test setup . . . 39
6.3 Performance of static load distribution . . . 39
6.4 The load balance of static load distribution . . . 42
6.5 Performance benchmark of dynamic load distribution . . . 43

7 Conclusions **44**

List of Figures

1 The `ParallelData` example written in Esterel . . . 5
2 A reaction timeline of the `ParallelData` example . . . 5
3 The high-level representation of the cyclic executive for run-time signal resolution 7
4 The `ParallelData` example represented in the Graph Code Format . . . 12
5 The `ParallelData` example represented in the Graph Code Format . . . 14
6 Illustration of places to insert *resolution* nodes . . . 15
7 Algorithm for inserting signal *resolution* nodes . . . 16
8 Auxiliary functions for the insertion algorithm . . . 17
9 An example of (a) a valid and (b) false dependency . . . 18
10 Algorithm for removing false dependencies . . . 19
11 An sketch of an example Esterel program . . . 20
12 The GRC representation of the `sketch` example . . . 21
13 The generated code of the `sketch` example with run-time signal resolution . . . 22
14 The partitioned GRC of the `ParallelData` example . . . 24
15 The distribution algorithm . . . 26
16 The partitioned GRC of the `sketch` example . . . 27
17 The generated code of the partitioned `sketch` example . . . 28
18 The reentrant version of the high level cyclic executive with run-time signal resolution 31

19 The GRC representation of the `sketch` example 32
20 The time-line of the state of the thread queue and processors 33
21 The `sketch` example illustrating thread queue access in the generated code . . . 35
22 Content of the (a) thread queue; and (b) mutex 36
23 Memory architecture of a quad Microblaze system: (a) the memory architecutre; (b) the memory address space 39
24 Performance comparison on PC (the lower the better) 40
25 Performance comparison on dual-core Microblaze (lower the better) 41
26 ACRT comparison of *LZSS* only on dual-core Microblaze (lower better) 42
27 Speedup of the average case reaction time 43
28 Speedup of the worst case reaction time 44

List of Tables

1 A list of benchmarking programs 38
2 Performance comparison on PC 40
3 Comparison of processor idle time in % 42

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399