

# A New Reactive Processor with Architectural Support for Control Dominated Embedded Systems

Partha S Roop      Z. Salcic      M. Biglari-Abhari      A. Bigdeli  
University of Auckland  
Department of EEE, Auckland, New Zealand  
{p.roop, z.salcic, m.abhari, a.bigdeli}@auckland.ac.nz

## Abstract

Control dominated embedded systems *have to be designed for fast reaction to asynchronous external events occurring in the environment. Such systems must be able to perform signal emission, signal polling, preemption and priority resolution efficiently. Current microprocessors and microcontrollers, however, have no direct support for such tasks and employ indirect mechanisms such as polling (via a port) and interrupts. In this paper, we propose a new processor core having architectural support for reactivity at the instruction level. The new processor core (called REFLIX) is an extension of an existing open source processor (FLIX) core with additional instructions to support reactivity. Initial benchmarking results (for some control dominated programs) show that REFLIX performs, on an average, 5.92 times faster compared to FLIX and has 77% code size reduction when compared to some conventional processors.*

## 1. Introduction

Embedded systems are *application specific* digital systems which normally reside within a larger electronic or mechanical environment. They are also reactive systems, which are in continuous interaction with their environment. The design paradigm for embedded systems, called code-sign [9], focuses on synthesis of interacting hardware and software components from a high-level description of system behavior. The hardware components are mapped to either ASICs or FPGAs while the software components are synthesized on a target microprocessor or microcontroller. One of the recent trends in codesign is to rely on processors for specific applications that better match the requirements of those applications than general purpose processors [4].

Embedded applications may be broadly classified as either *control dominated* (in which input events arrive completely asynchronously and the arrival time of these events is critical), or *data dominated* (input events arrive at regular intervals and the value of the event is more critical than the time of arrival). In control dominated systems fast reaction to input events is critical. Data dominated systems, on the other hand, require efficient processing while computing some mathematical function of the input streams [2]. In this paper, our focus is control dominated systems which require fast reaction to input events. We propose a novel processor,

called REFLIX, that has architectural support for execution of control dominated applications.

REFLIX architecture is inspired by a synchronous programming language called Esterel [3] that provides neat set of constructs for modelling, verification and synthesis of reactive systems. The environment of a Esterel program consists of a set of sensors and signals, which can be modeled abstractly using constructs available in the language. The activation clock of any Esterel program is a predefined event called the *tick*. During every tick the Esterel kernel samples its environment and performs a set of instantaneous reactions based on the values of sensors and signals in its environment during the present tick. The main constructs for interaction with the environment are *await* (which is a delay construct), *emit* (which performs signal emission to the environment), *sustain* (which emits a signal for ever), *abort* (which is a preemption construct), and *trap* (which is similar to software interrupts). These constructs together with *synchronous broadcast* communication between concurrent threads forms the core of Esterel for modelling control dominated aspects of the embedded system.

REFLIX provides architectural features to directly support reactivity demanded by control dominated systems. These include instruction level support for signal emission, signal polling, preemption and priority resolution. Conventionally, signal emission and signal polling are done in processors using either *memory mapped* scheme or *IO mapped* scheme. In the former, references will have to be made to specific region of external memory and in the latter a port needs to be accessed. In REFLIX we support instructions to do this directly. Preemption is normally handled in processors using either *polling*, which wastes considerable CPU cycles, or *interrupts* which are much more efficient but have context switching overhead. For control dominated tasks, where fast preemption is vital and there is no requirement to return to the exact context, interrupts have unnecessary overhead. We propose an alternative mechanism called *ABORT* at the instruction level which is ideal for control dominated applications. Finally, we propose a novel way of priority resolution through nesting of *ABORT* instructions, which is designed for control dominated applications.

REFLIX is implemented by introducing these architectural extensions to an open source processor called FLIX [10] using FPGA technology. Initial benchmarking results show considerable improvement in execution time and code size.

The main contributions of the paper are as follows:

1. Instruction set architecture (ISA) that directly supports

reactivity: the proposed architecture is a first attempt to support reactivity at the native instruction set level. One of the main extensions is the inclusion of a mechanism called ABORT for handling preemption and priority. This will enable easy and efficient implementation of control driven embedded software during code-sign.

2. Reusability: Though some reactive languages and tools support direct hardware synthesis from high-level specifications, new netlist will have to be generated for every new design. REFLIX, on the other hand, can be reused for the software component of any new embedded application.
3. Better performance and code density: The proposed extensions lead to about 5.95 times speedup compared to FLIX and about 77% reduction in code size compared to four conventional processors (FLIX, Intel 8051, Motorola 68HC11 and Altera NIOS).

### 1.1. Related Work

One of the trends in implementation of embedded systems, such as cell phones, medical appliances and home appliances is to rely on processors for specific applications that better match the requirements of those applications than general purpose processors [4]. There are several approaches suggested for customization of processors. Some of them rely on using existing architectures, such as those of ARM [5] or MIPS [12] or use parametrized processor cores that are customized at the time of their compilation such as the Altera NIOS processor [1]. None of these processors, however, support any generic mechanism of reactivity and preemption beyond usual interrupt structures. REFLIX, supports a novel preemption mechanism called *ABORT* which is better suited to control dominated applications.

REFLIX processor core is inspired by some of the features of Esterel language. Esterel language has been used in the past for the generation of hardware circuits [6]. Esterel language is designed to capture requirements of control dominated systems at a higher level. By capturing similar features at native instruction level, REFLIX is to provide an efficient and reusable platform for the design and implementation of control dominated embedded systems.

This paper is organized as follows: In Section 2 we present the new instructions for native support of reactivity and then compare this to conventional processors. In Section 3 we present a prototype design for supporting the new instructions. In Section 4 we present some results on comparison of REFLIX to other processors. The final section is devoted to some concluding remarks.

## 2. Reactivity and REFLIX

Five most important features required to support control dominated reactive tasks are: *Signal emission*, *Signal polling*, *Preemption*, *Priority resolution* and *Concurrency*. REFLIX has native instruction set and architectural support to handle a subset of these features, as presented in the next section.

### 2.1. Architectural Support for Reactivity in REFLIX

In this section, we introduce the main architectural features of REFLIX processor and the main ideas that lead to its design. For rapid prototyping, an open source processor core called FLIX [10] was selected.

The main extensions to FLIX core are as follows:

1. A new preemption mechanism suitable for reactivity called ABORT for handling asynchronous events with different priorities.
2. Variable number of input sensor and output sensor lines: in the initial prototype we support 16 sensor input lines, Sin[15..0], and 16 signal output lines Sout[15..0]. These lines are used for immediate reaction to external events.
3. Introduction of internal timers that generate user programmable *time out* signals. Time out signals can be used for interaction with the environment and also can be fed back to REFLIX core for synchronization purposes.
4. An instruction set that has native support for all the reactive features discussed in the previous section except concurrency.

The external view of REFLIX is shown in Figure 1.

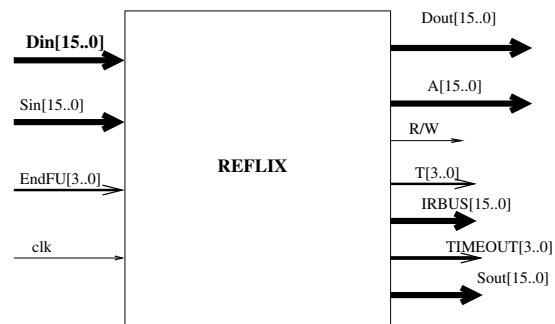


Figure 1. REFLIX: External View

### 2.2. Native Reactive Instructions

The instruction set architecture (ISA) of REFLIX is an extension of the FLIX ISA [10]. The original FLIX provides basic instructions for memory reference operations, which use direct or register addressing, register transfer operations, arithmetic/logic operations, conditional and unconditional control transfer operations. All instructions are 16 bits and are fetched as one word from memory.

The original FLIX instruction set is appended with new group of instructions that support reactive processing. There are eight basic instructions in the reactive category and they are presented in Table 1. Most of REFLIX instructions are only one word long, but some of the reactive instructions require two words for immediate operand or address information.

Esterel Feature	Instruction Syntax	Length	Description
<i>Preemption</i>	<b>ABORT</b> <i>signal, address</i>	2 Words	Preemption instruction. ABORT has a body up to the instruction whose address is indicated in the instruction (called continuation address since after preemption program continues from this address). <i>signal</i> can be either an external one or a <i>TimeOut</i> received from internal timer.
<i>Signal Emission</i>	<b>EMIT</b> <i>signal</i>	1 Word	The specified signal is set high for one instruction cycle.
<i>Signal Polling</i>	<b>SAWAIT</b> <i>signal</i>	1 Word	Wait until specified signal occurs in the environment.
<i>Delay</i>	<b>TAWAIT</b> <i>delay</i>	2 Words	Wait until delay (no. of instruction cycles) elapses.
<i>Conditional Signal Polling</i>	<b>CAWAIT</b> <i>signal1, signal2, address</i>	2 Words	Wait until either <i>signal1</i> or <i>signal2</i> occur. If <i>signal1</i> occurs then execute instructions from the address following this instruction, else from specified <i>address</i> .
<i>Signal Presence</i>	<b>PRESENT</b> <i>signal, address</i>	2 Words	If <i>signal</i> is present next instruction is fetched from the next consecutive address. Otherwise, it is fetched from the specified <i>address</i> .
<i>Delay</i>	<b>TAWAIT</b> <i>delay</i>	2 Words	Wait until delay (no. of instruction cycles) elapses.
<i>Signal Sustainance</i>	<b>SUSTAIN</b> <i>signal</i>	1 Word	Specified signal is set high for ever.

**Table 1. REFLIX instructions supporting reactive processing.**

EMIT and SUSTAIN are instructions to generate external signals (outputs) which can last one system tick (EMIT) or indefinitely (SUSTAIN). SAWAIT and TAWAIT represent *busy waiting* mechanism on two types of events: external signals and time outs (generated by internal timers). CAWAIT is a conditional polling mechanism to support branching after delay. PRESENT instruction is like a branch instruction where the branch condition is dependent on external signal. Finally, ABORT is the preemption instruction and also can handle priority. We illustrate the instruction set using the following example.

### 2.3. Example: Pump Controller

Consider, for example, the following specification of a simple *pump controller* [7]:

*A pump controller is used to control the operation of a pump inside a mine which may have high methane levels. The pump is used to pump out water (whenever the water level exceeds the desired level) provided the methane level is below the desired level (RIGHT-METHANE). Whenever, methane level goes above this desired level (NOT-RIGHT-METHANE), the controller must stop the pump and wait until right methane level is restored.*

An implementation of this specification in REFLIX assembly language is as follows:

```

start1:
  ABORT NOT-RIGHT-METHANE, L1
  #abort body
  loop:
    SAWAIT HIGH-WATER-LEVEL
    EMIT START-PUMP
    SAWAIT LOW-WATER-LEVEL
    EMIT STOP-PUMP
    JMP loop
  #end of abort body
L1:
  #handle exception

```

```

EMIT STOP-PUMP
SAWAIT RIGHT-METHANE
#return to resume normal operation
JMP start1

```

The pump controller is implemented in two parts: normal behavior, which is enclosed by an *ABORT* statement (preemption mechanism in REFLIX), and exception behavior, which is provided from the continuation address of the *ABORT* (when the body of the *ABORT* is preempted, program control resumes from this address). The behavior of the *ABORT* body has an *SAWAIT* statement that samples the water level. Whenever water level is high (*HIGH-WATER-LEVEL* triggers), signal *START-PUMP* is emitted to start the pump. Another *SAWAIT* is used to check the water level again when it reaches the low level. Then the pump is immediately stopped using another *EMIT* statement.

This normal behavior is continued until the enclosing *ABORT* triggers (activates). *ABORT* activates either when the signal *NOT-RIGHT-METHANE* occurs in the environment and the body has not finished execution, or when the body terminates before this signal occurs. In this case, since the body is an infinite loop, it never terminates and *ABORT* triggers only when the signal *NOT-RIGHT-METHANE* occurs.

When *ABORT* triggers, current instruction in the body is completed and the next instruction is automatically fetched from the continuation address provided with *ABORT* (L1 in our example). So whenever, *NOT-RIGHT-METHANE* is detected the pump is immediately stopped (by an *EMIT* statement) and the controller waits for methane level to restore (*RIGHT-METHANE* is sensed using another *SAWAIT*) before resuming normal operation.

## 2.4. Comparison to Conventional Processors

Consider how the same behavior can be achieved using a conventional microcontroller such as Intel 8051:

```

#setup interrupt vector
ORG addr
DD HighMethanLevel

#wait for high water level
start:
loop1: MOV A, HIGH-WATER-LEVEL
      CJNE A, Px, loop1
      #water level high detected; start pump
      MOV Px, START-PUMP
loop2: #wait for low water level
      CJNE A, Px, loop2
      #water level low detected; stop pump
      MOV Px, STOP-PUMP
      LJMP start

#Interrupt service routine (ISR)
HighMethaneLevel
#save registers to be used in ISR
PUSH A
#stop the pump
MOV Px, STOP-PUMP
#wait for right methane level
MOV A, RIGHT-METHANE
loop:  CJNE A, Px, loop
      #restore registers
      POP A
      #before return from interrupt
      #modify point of return to start
      POP direct
      PUSH start
      #return from interrupt
      RETI

```

In this implementation, an interrupt vector needs to be setup for handling high methane level within the mine. The main routine samples the water level and starts or stops the pump appropriately. The interrupt routine stops the pump when methane level is not right and waits until right level is detected before returning to main program. This example illustrates that even ignoring the initialization and context switching overhead in 8051, which might have significant impact for some applications, we still have 14 instructions in 8051 (or, 17 instructions with a polling mechanism for abort handling) compared to 9 instructions in REFLIX. REFLIX code has no context switching overhead since saving and restoring context is not a requirement of the application.

## 2.5. Priority

Let us now consider a slightly modified pump controller specification as given below [7]:

*A pump controller is used to control the operation of a pump inside a mine which may have high methane levels. The pump is used to pump out water (whenever the water level exceeds the desired level) provided the methane level is below the desired level (RIGHT-METHANE). Whenever, methane level goes above this desired level (NOT-RIGHT-METHANE), the controller must stop the pump and wait until right methane level is restored. If at any time, however, the methane level is too high (NOT-RIGHT-METHANE is only marginally high) then the pump must be stopped immediately and an ALARM must be generated. Pumping is stopped until right methane level is restored*

Note that in this specification there is a higher priority preemption condition triggered by HIGH-METHANE over NOT-RIGHT-METHANE. Priority is implemented in REFLIX using nesting of ABORT instructions with outer ABORT instructions having higher priority over inner ones. Let us consider the following implementation of this specification in REFLIX:

```

start:
ABORT HIGH-METHANE ADDR1
start1:
      ABORT NOT-RIGHT-METHANE ADDR
      #abort body
      loop:
            SAAWAIT HIGH-WATER-LEVEL
            EMIT START-PUMP
            SAAWAIT LOW-WATER-LEVEL
            EMIT STOP-PUMP
            JMP loop
      #end of abort body
ADDR:
#handle exception
EMIT STOP-PUMP
SAAWAIT RIGHT-METHANE
#return to resume normal operation
JMP start1

ADD1:
#handle high methane
EMIT STOP-PUMP
EMIT ALARM
SAAWAIT RIGHT-METHANE
JUMP start

```

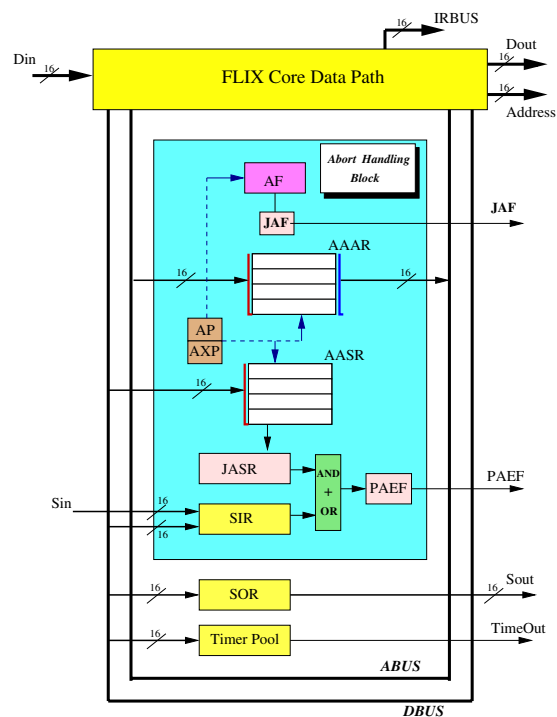


Figure 2. REFLIX Data Path

In this implementation, HIGH-METHANE has higher priority over NOT-RIGHT-METHANE. Hence, whenever HIGH-METHANE is detected, the pumping is stopped and alarm is generated. This example illustrates the simplicity of priority handling in REFLIX. In a conventional processor, implementation of this would require explicit polling to determine when the higher priority signal triggers (nested interrupts if employed will be equally expensive as some extra overhead will have to be incorporated to force non-return to the interrupted program). ABORT provides an elegant mechanism to incorporate priorities in control dominated tasks. Also, the overhead associated with context switching is completely eliminated since this is not a requirement for the task involved.

### 3. Prototype Design

The datapath of REFLIX prototype is shown in Figure 2. The datapath is organized around two internal buses, called ABUS and DBUS. ABUS is used for carrying address information between internal registers while DBUS is used for carrying data. They also enable two pairs of transfers between two internal registers at the same time (machine cycle). This datapath is similar to the FLIX datapath, an open core processor [10]. Conceptual operation of the control unit of REFLIX is shown in Figure 3.

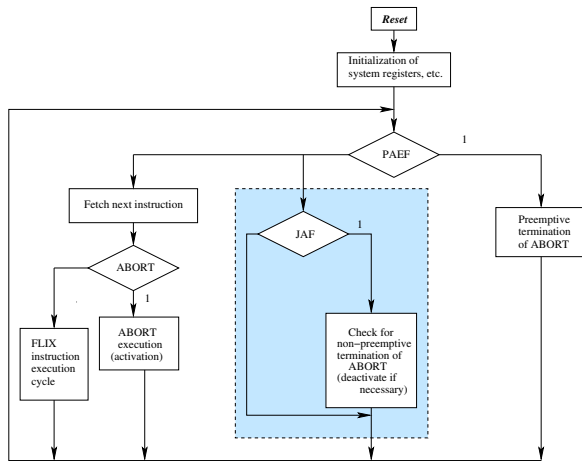


Figure 3. REFLIX Control Unit

Upon power-up or reset, REFLIX goes through initialization phase to set the initial condition of all registers. In the execution step, priority is given to handling of preemptions, which may trigger during current instruction cycle (because of pending ABORT instructions that might trigger now). For ABORT handling, all recorded events on monitored signals during one instruction cycle will be given attention in the following instruction cycle. When there is no pending event for preemption, the control unit performs one or both of the following actions:

1. Next instruction is fetched from memory and executed.
2. Non-preemptive termination of ABORT instruction, if any, is performed. This happens when ABORT body finishes execution before preemption happens.

PAEF (pending abort event flag) and JAF (joint abort flag) shown in Figure 3 are discussed in the next section.

#### 3.1. Abort Handling Block

ABORT instruction is introduced in REFLIX to perform preemption with priorities. In the current prototype, ABORT instruction can work with up to 16 different external input signals and up to 4 internal timer generated signals. Up to four levels of nesting of ABORTs is supported for handling priorities. A dedicated hardware unit has been

added to facilitate the execution of ABORT instructions, which is termed as the Abort Handling Block (AHB).

An ABORT instruction is active from the instant it is executed to the instant it is preempted or the ABORT body has been fully executed. Body is preempted either when the preemption condition happens in the environment before the body has finished execution (preemptive ABORT termination) or if the body finishes execution before this (non-preemptive ABORT termination).

ABORT instruction is executed through two stages, which are supported by the AHB:

1. **ABORT Activation:** When an ABORT instruction is fetched and decoded, it becomes active. The continuation address and the signal involved are stored in appropriate registers (called Active Abort Address Register (AAAR) and Active Abort Signal Register (AASR) respectively).
2. **ABORT Termination:** Once the designated signal occurs in the environment (and provided no higher priority ABORT signal is also active), ABORT is taken and an unconditional jump to the continuation address is executed, or, if the continuation address is reached and the designated signal has not occurred in the environment, ABORT is automatically terminated.

Abort Handling Block (AHB) supports nesting of ABORT statements. The AHB contains active abort signal register (AASR) block with four 16-bit registers. These four registers are provided to support four levels of ABORT. Similarly, active abort address register (AAAR) has four 16-bit registers to store the continuation address of four levels of ABORT. (AAAR(0) and AASR(0) correspond to the outer most ABORT, and AAAR(3) and AASR(3) to the inner most ABORT). Each bit in a 16-bit register of AASR indicates if the corresponding signal is active. A detailed design of AHB and REFLIX control can be found in [11].

### 4. Implementation and Discussion

REFLIX has been implemented using FPLD technology as a proof of concept prototype. The open-source VHDL code available for FLIX in [10] was modified to introduce the architectural extensions discussed in the previous section. We first simulated the design using Active HDL from ALDEC and then mapped the design to Altera FPGAs.

Since we developed a processor for control dominated applications, we needed a set of benchmark programs to validate the ideas. Since, no such benchmark is directly available, we created a set of benchmark applications for control dominated systems. These include a *transmission control protocol (TCP) transmitter and receiver*, a *pump controller* [7], an *automatic teller machine (ATM) controller*, a *traffic light controller* taken from the POLIS code-sign tool [2], a *lift controller* and a *startup benchmark* [8] which builds a segment of the cell modem startup procedure.

In all these applications, we abstracted the data handling code and only focused on the reactive code (as a result the benchmarks are small). As our first comparison, we compared the execution time for FLIX and REFLIX over the same application programs. They are compiled and executed for both REFLIX and FLIX. Table 2 indicates the total number of instruction cycles for each of these benchmarks.

Application	Nesting	REFLIX	FLIX	Speedup
TCP Transmitter	0	10	46	4.6
TCP Receiver	0	9	41	4.55
Pump Controller	2	12	83	5.92
Startup benchmark	1	23	102	4.43
ATM Machine	3	22	244	11.09
Traffic Light Controller	2	23	152	6.60
Lift Controller	0	27	116	4.14
Average				5.92

**Table 2. Comparison of execution time of REFLIX and FLIX.**

Since the instruction cycle duration of FLIX and REFLIX are identical, comparing the number of instruction cycles is equivalent to the execution time. On an average, REFLIX turns out to be 5.92 times faster than FLIX while executing these control dominated programs. *This shows that an existing processor can be modified to enable more efficient implementation of the same control dominated application programs.*

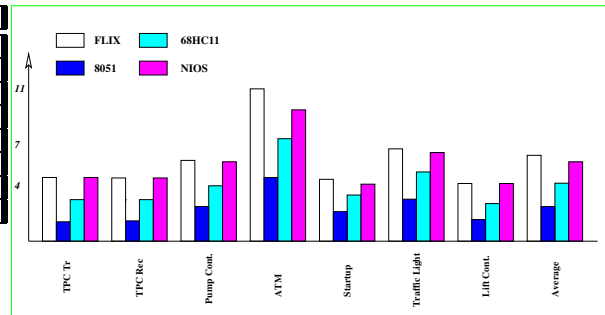
In this section, we present some preliminary comparisons (number of 16-bit words) of REFLIX with three popular processors widely used for embedded applications. We have used the same control dominated applications (Table 2), initially written in Esterel and subsequently mapped to REFLIX, Motorola 68HC11, Intel 8051, and NIOS (16-bit version) processors.

Figure 4 summarizes these results. The Y-Axis indicates the code size increase for each processor (for every benchmark which appears in the X-Axis) when compared to REFLIX (which is normalized to 1). *These results show that native reactive support produces very compact code for control dominated applications compared to conventional processors.*

Intel 8051 supports both memory mapped and direct IO and also has some instructions for signal polling, due to which it has better code density compared to 68HC11 and NIOS. REFLIX has, on an average, 77% reduction in code size as it directly supports most features of control dominated applications. The most interesting point to note is that, 8051 and REFLIX have marginally different code size when the levels of nested aborts are low. With increasing levels of nesting of aborts, conventional processors require much more instructions to handle priority and preemption. This is highlighted by the fact that code size increase in these processors were minimal in the TCP transmitter and receiver benchmarks, which has no aborts but maximum code size increase happened in the ATM benchmark, which has 3 levels of nesting of aborts.

## 5. Conclusion and Future Work

Control dominated embedded applications are intrinsically reactive and require fast reaction to external events. This paper presents a new approach to designing architectures that better support control dominated embedded applications. We built a prototype processor, REFLIX, by extending an open source processor FLIX with native support for reactivity and a new preemption mechanism called *ABORT*. We then compared execution time of FLIX and REFLIX on a set of control dominated applications and obtained an average speedup of 5.92 times using REFLIX. We



**Figure 4. Code size increase for each processor compared to REFLIX.**

also made memory footprint comparisons of REFLIX and a set of conventional processors. REFLIX produced considerably more compact code compared to all these processors.

Though native support for reactivity is very novel, the proof of concept prototype (REFLIX) has several limitations. Firstly, REFLIX is not pipelined and its design is also not parametric. Secondly, we have no support for concurrent task execution in REFLIX. Finally, a set of tools to support research on REFLIX are under development at the moment.

## References

- [1] Altera-Corporation. Excalibur embedded processor solutions. <http://www.altera.com>.
- [2] F. Balarin, M. Chiodo, P. Guisto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A. Sangiovanno-Vincentelli, E. Sentovich, K. Suzuki, and B. Tabbara. *Hardware Software Codesign of Embedded Systems - The POLIS Approach*. Kluwer, 1997.
- [3] G. Berry and G. Gonthier. The ESTEREL synchronous programming language. *Science of Computer Programming*, 19:87–152, 1992.
- [4] J. A. Fisher. Customized instruction sets for embedded processors. In *36th Design Automation Conference*, pages 253–257, 1999.
- [5] S. Furber. *ARM system-on-chip architecture*. Addison-Wesley, 2000.
- [6] A. Girault and G. Berry. Circuit generation and verification of Esterel. In *International Symposium on Signals, Circuits and Systems*, pages 85–90, Iasi, Romania, 1999.
- [7] H. Gomaa. *Software design methods for concurrent and real-time systems*. Addison-Wesley, 1993.
- [8] ITU. *International Telecommunication Union, ITU-T recommendation v.32 edition*, 1993.
- [9] M. Meerwein, C. Baumgartner, and W. Glauert. Linking codesign and reuse in embedded systems design. In *8th International Workshop on Hardware Software Codesign*, San Diego, CA, USA, 2000. ACM.
- [10] Z. Salcic. *VHDL and FPLDs in Digital System Design*. Kluwer Academic, 1998.
- [11] Z. Salcic, P. Roop, M. Biglari-Abhari, and A. Bigdeli. Reflex: A processor core for reactive embedded applications. In *12th International Conference on Field-Programmable Logic*, number 2438 in LNCS, pages 945–954. Springer, 2002.
- [12] Triscend. The configurable system on a chip. <http://www.triscend.com>.