Performance-Driven Syntax-Directed Synthesis of Asynchronous Processors

Luis A. Plana, Doug Edwards, Sam Taylor, Luis Tarazona, and Andrew Bardsley
Advanced Processor Technologies Group
School of Computer Science, The University of Manchester
Manchester, M13 9PL, United Kingdom
{lplana, dedwards, smtaylor, ltarazona, abardsley}@cs.manchester.ac.uk

ABSTRACT

The development of robust and efficient synthesis tools is important if asynchronous design is to gain more widespread acceptance. Syntax-directed translation is a powerful synthesis paradigm that compiles transparently a system specification written in a high-level language into a network of pre-designed handshaking modules. The transparency is provided by a one-to-one mapping from language constructs to the module networks that implement them. This gives the designer flexibility, at the language level, to optimise the resulting circuit in terms of performance, area or power.

This paper introduces new techniques that exploit this flexibility to improve the performance of synthesised asynchronous systems. The results of a series of transistor level simulations show that these techniques, combined with optimised handshake module implementations, can produce close to a ten-fold improvement in the performance of a 32-bit, ARM-compatible, asynchronous processor used in an experimental smartcard SoC, without introducing any changes to the original processor architecture.

Categories and Subject Descriptors

B.5 [**Register-Transfer-Level Implementations**]: Design Aids—*Automatic synthesis, Optimization*

General Terms

Design, Performance

1. INTRODUCTION

Most modern embedded systems are synthesised using CAD tools. Although a large proportion of these systems are synchronous, interest in asynchronous circuits and tools is continually growing for their low EMI, robust on-chip interconnect and their potential to deal effectively with process variation.

There are several asynchronous synthesis systems available. Some target the synthesis of asynchronous controllers, *e.g.*, Petrify [1] and Minimalist [2]. Others target both control and datapath but may require user intervention or guidance during the synthesis process, *e.g.*, TAST [3] and the CSP-like CHP system [4]. Tangram [5] and Balsa [6] are fully-automated systems that have successfully synthesised large-scale circuits using syntax-directed compilation. This paper focuses on this synthesis approach and examines the opportunities to optimise the performance of the generated circuits.

2. SYNTAX-DIRECTED SYNTHESIS

Syntax-Directed translation is a powerful synthesis technique. The synthesis process involves compiling descriptions written in a high-level language into a network of pre-designed modules. This approach gives a 'transparent' compilation, i.e., there is a one-to-one mapping from a language construct to the network of modules that implements it. This direct mapping gives the designer flexibility at the language level to impact the resulting circuit; incremental changes at the language level result in predictable changes in the implementation. The source code specification may have a large impact on the performance, power consumption and area of the resulting circuit.

In many cases, the synthesis system can evaluate the generated module network to provide the user with an early estimate of the performance and area of the resulting circuit. An experienced designer can optimise the resulting circuit in terms of performance, area or power by choosing the right specification. Syntax-directed translation has been used successfully in the synthesis of several embedded systems, including the G3Card smartcard System-on-Chip, an asynchronous MIPS microprocessor [7] and the ARM996HS, the first commercially-available synthesisable asynchonous ARM. The G3Card SoC is a good example of an asynchronous embedded system. A prototype was fabricated in a 0.18μm process and was fully functional on first silicon. It contains two different, full-featured implementations of an ARM compatible, asynchronous processor, a Memory Protection Unit, an asynchronous interface to standard synchronous RAM, a synchronisation coprocessor, and several peripherals, all synthesised using the Balsa synthesis system.

2.1 The Balsa Synthesis System

Balsa is a synthesis system that generates purely asynchronous macromodular circuits, called handshake circuits.

Proposed originally for use with the Tangram language (upon which Balsa is heavily based), handshake circuits offer an attractive paradigm for circuit synthesis. Complex descriptions written in the source language are translated into a circuit consisting of instances of handshake components composed in a macromodular style.

```
procedure buffer
(
  parameter DataType : type;
  input  in  : DataType;
  output out : DataType
) is
    variable buf : DataType
begin
    loop
        in -> buf;
        out <- buf
    end -- loop
end -- procedure buffer
```

Figure 1: Balsa code for 1-place buffer.

Figure 1 shows how a simple 1-place buffer is specified in Balsa. The specification is parameterised in the type of data that the register holds. Data is stored in a variable [*buf*] and the operation is described as an infinite repetition [*loop*] of two actions: input [–>] data from channel [*in*] into *buf* sequenced [*;*] with output of the [<–] data in *buf* to channel [*out*].

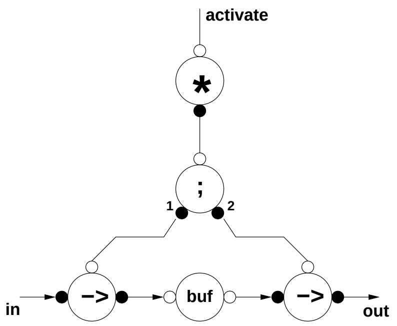

Figure 2: 1-Place buffer handshake circuit.

Figure 2 shows the handshake circuit for a Balsa-synthesised pipeline register. The data is stored in a latch [*buf*] and a Sequence component [*;*] is used to sequence the writing [→] to and reading [→] from [*buf*]. A Loop component [***] repeatedly activates the Sequencer. The one-to-one mapping of the language constructs into handshake components is very clear. Even though the handshake circuit is composed of a number of handshake components it is not a complex circuit. The handshake components in the figure are all very simple: the Transferrer component [→] is implemented with wires only, the Loop component requires a *NOR* gate and the Sequencer consists of a *C-element* and an *AND* gate.

handshake components are selected from a relatively small set and are straightforward to implement. They are interconnected through channels. Each channel connects an *active* port on one component to a *passive* port on another. The sense of the port (active or passive) indicates the direction of the handshake. An active port initiates a handshake (sends the request) and the passive port acknowledges. Channels can carry data and this can flow in either the same direction as the request (a *push* channel) or in the opposite direction (a *pull* channel).

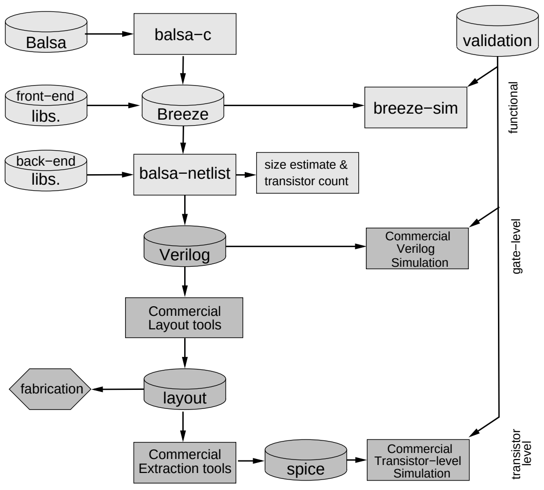

Figure 3: Balsa design flow.

The generation of the handshake Circuit is the first step in the Balsa synthesis flow, shown in Figure 3. The Balsa compiler generates an intermediate netlist, in *Breeze* format, which can be used for functional validation and early performance estimates. The Balsa netlister generates a structural verilog netlist based on the target celll library and the selected asynchronous style and data encoding. The Verilog netlist can be processed with commercial layout and extraction tools for further validation and fabrication.

The G3Card SoC synthesised with Balsa was based on the SPA processor, a fully synthesised, 32 bit, 100% ARM compatible processor core. SPA was implemented as a simple, ARM7 style, 3-stage pipeline . Both *dual-rail* (1-of-2 data encoding, quasi-delay insensitive -Q^nDI- timing assumptions) and *bundled data* (single-rail data encoding, data-bundling timing assumptions) were implemented from the same Balsa specification.

The main goal of the dual-rail processor implementation was to defeat power analysis, therefore, a power-balanced circuit was targeted. Performance was not a significant requirement for the smartcard, however, the synthesised SPA was significantly slower than expected. The following section explores performance-oriented techniques that result in improved performance.

3. PERFORMANCE-DRIVEN SYNTHESIS

The techniques introduced in this section target optimised performance of Balsa-synthesised circuits. Area is not considered a significant factor, although the Results Section shows that significant reductions in area are also achieved. These techniques presented rely on the use of new handshake components that eliminate unnecessary synchronisation between data and control and alow more concurrent operation. Details of this new handshake components can be found elsewhere [8].

3.1 Efficient Pipeline Control

Almost all modern embedded processors are pipelined, therefore, asynchronous synthesis tools must generate efficient pipeline control logic. Balsa has no special language constructs or handshake components to specify or implement pipelines. Pipeline stages are usually specified in Balsa procedures and pipeline registers are implemented using conventional variables. Balsa does not allow concurrent reads and writes to the same variable which means that, when a variable is used as a pipeline register, the stages at either side of the variable cannot normally process data concurrently.

As indicated earlier, the pipeline registers are variables inside each stage and both input and output registers are used. This structure essentially implements a half-occupancy pipeline: a pipeline with twice as many stages, with alternating *processing* and *empty* stages. Without the empty stages, adjacent processing stages would not be able to operate concurrently, severely limiting the throughput of the pipeline.

The use of the new handshake components introduced earlier results in a more efficient pipeline implementation. Pipeline registers are not general-purpose variables: they are always written by a stage and then read by the following one. This write/read access pattern allows the use of a single variable as a true pipeline register. In this case, the registers are specified outside the processing stages and the stages contain only the processing logic. The pipeline register implemented using the new handshake components turns out to be very simple and performs quite well compared to highly optimised controllers.

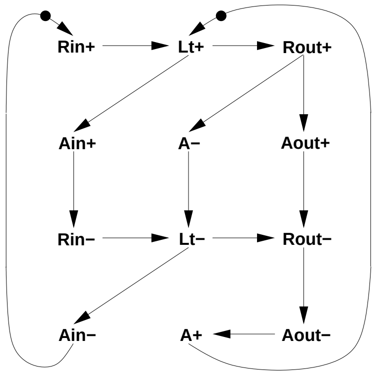

Figure 4: Pipeline control.

Figure 4 shows the behaviour of the pipeline controller. R and A represent the request and acknowledge signals used to implement the *in* and *out* channels, and *Lt* represents the latch enable signal.

The behaviour depicted in the figure shows that the Balsa-synthesised pipeline controller implements an efficient, fully-decoupled, request-activated protocol. Figure 5 shows the implementation of the pipeline register controller as generated by the Balsa back-end. The *activate* signal is used to initialise the register.

3.2 True Asynchronous Operation

The pipeline structures described above operate in a *pseudo-synchronous* fashion, i.e., the transfers of all data items from one stage to the next occur simultaneously as in synchronous systems. Although there is no global clock, data

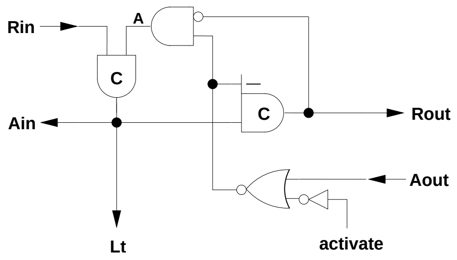

Figure 5: Pipeline control handshake circuit.

advances through the pipeline in *lockstep*, using local handshake channels. This *regular* operation is easy to understand and evaluate but can reduce the overall performance of the synthesised processor.

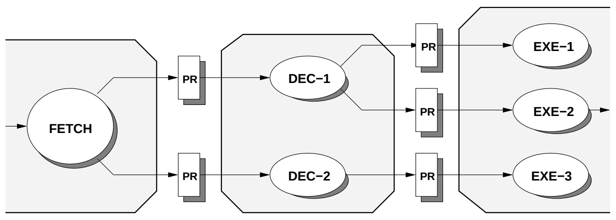

Figure 6: True asynchronous pipeline.

True asynchronous pipelines, make no attempt to have a lockstep operation, as shown in Figure 6. Each unit within a pipeline stage is allowed to progress at its own pace, handshaking individually with units in the previous and following stages. This means, for example, that different data items sent by units in the Decoder can arrive in the Execute stage at completely unrelated times. Consequently, different units in the Execute stage can operate on data items that correspond to different instructions, giving some of them a head start. Given the elastic nature of asynchronous pipelines, true asynchronous operation will result in improved performance in most cases.

3.3 Data-Driven Operation

A drawback of syntax-directed synthesis is the overhead imposed on the circuits by the control-driven approach to the translation. Data and control are frequently synchronised and, often, control is slower than data reducing the performance of the circuit as data is stalled while control catches up.

Figure 7(a) shows a segment of a simplified Balsa description of the *EXECUTE* stage of the SPA processor. This is an example of a control-driven description. In this code the operations are explicitly sequenced, as is commonly done in high-level language descriptions. The syntax-directed translation will result in a handshake Circuit consisting of a tree of control components that direct the movement of data through the datapath.

Figure 8 shows the simplified handshake Circuit. *Transferrers* [→] are used to control the flow of data through the datapath. Given that the control tree guarantees mutually exclusive operation, an uncontrolled *Merge* component [|] can be used to the merge the results of the different units

```
doRegisterRead;
case instruction of
add then
    doShift;
    doAlu
| mul then
    doMul
| ldr, str then
    doMemAccess
end;
doRegisterWriteBack
```

(a)

```
doRegisterRead ||
steerRegData ||
doShift ||
doAlu ||
doMul ||
doMemAccess ||
multiplexResults ||
doRegisterWriteBack
```

(b)

Figure 7: Execute stage Balsa code.

into the register write-back. In every *step*, the control will *start* and operation and wait until the *result* is ready before starting the next one. The control circuit is generated as part of the syntax-directed translation and will resemble the description, with 3 *Sequencers*, a *Case* and several *completion detection* elements. The latency through the control tree is likely to be very large, affecting the performance of the circuit.

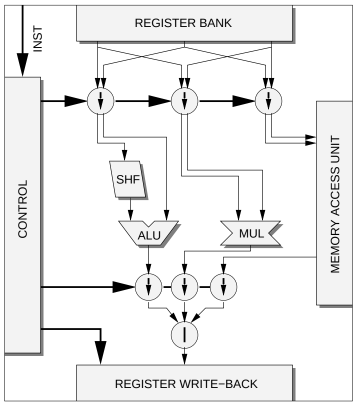

Figure 8: Control-Driven Execute stage.

The control-data synchronisation may seem like a necessary price to pay to guarantee correct operation but this is not the case. In asynchronous circuits valid data identifies itself. There is no need for explicit sequencing of the operations: the units can wait until data arrives, process it and send result data out. Figure 7(b) shows an alternative, data-driven description of the same stage.

In this description, all units are activated in parallel. A *steer* and a *multiplex* units are added to guide the data. Figure 9 shows the simplified handshake Circuit. All the units are ready to receive data and will start operating as soon as the data arrives. In a *multiply* instruction, the data is sent to the *MUL* unit and the rest will remain ready. Although the *steer* and *multiplex* modules require control signals, these can be setup directly by the decoder, without involving any sequencing and without any need to synchronise with the data. The *steering* control signals are very likely to be ready before the data and will not delay the operations, clearly improving the performance of the stage.

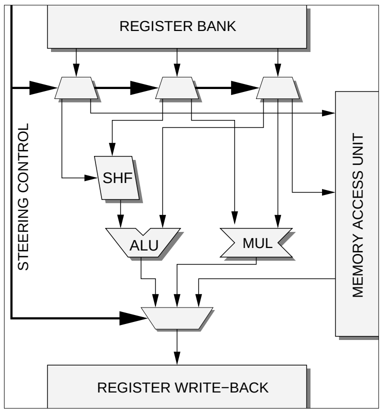

Figure 9: Data-Driven Execute stage.

3.4 Speculative Operation

Speculative operation is an important tool in the design of modern embedded processors. Significant performance improvements can be obtained if the results of speculative operations are useful most of the time. The ARM ISA establishes that all instructions are conditional, i.e., they can be executed or skipped depending on the condition codes. Program execution statistics show that most instructions are executed, opening the possibility of speculatively starting the instruction and throwing away the results if the condition code test fails.

Speculative operation is not always straightforward to implement in synthesised asynchronous systems. In these systems, handshake channels, and not individual signals, are used to communicate data. A system is likely to deadlock if a channel is prevented from completing a handshake cycle. For this reason, SPA has no speculative operation: it evaluates the condition code of an incoming instruction and starts execution only if the condition passes.

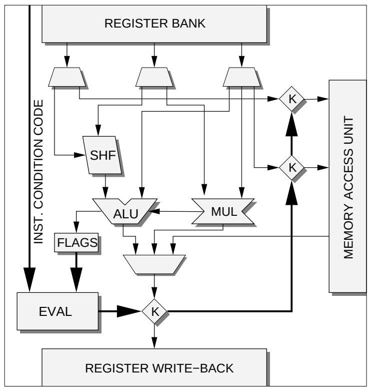

Figure 10: Speculative operation control.

A performance-oriented implementation can incorporate speculative operation in the execute unit: the evaluation of the condition code can be carried out concurrently with the execution of the instruction. If the condition fails the instruction is discarded at a checkpoint without any result being written back. The key issues are the location of the checkpoints and the need to allow all handshake channels to complete their cycles. Figure 10 shows how kill modules [K] are used for this purpose. Data-processing operations are started speculatively and, if the condition test fails, are discarded before the write-back of the results. On the other hand, data memory operations are not started speculatively as the performance and power penalties for the discarded instruction would be extremely high. This strategy will result in a performance improvement only if the percentage of executed instructions is high, but this is usually the case.

4. RESULTS

This section shows the results of pre-layout, transistor-level simulations of several implementations of the SPA processor using a $0.18\mu m$ standard cell library. The simulations show that the techniques described earlier result in significant performance improvements.

Processor	DMIPS	Relative Perf.	Trans.	Relative Size
Bundled Data				
SPA	10.17	1.00	283,663	1.00
nanoSpa	22.46	2.21	181,749	0.64
nanoSpa with new HCs	54.44	5.35	242,724	0.86
Dual Rail				
SPA	6.53	1.00	717,549	1.00
nanoSpa	18.57	2.84	611,578	0.85
nanoSpa with new HCs	58.37	8.94	570,920	0.80

Table 1: Simulation Results.

The performance-driven techniques were applied in the synthesis of nanoSpa, a new Balsa specification of the original SPA architecture. NanoSpa is organised as a 3-stage, Harvard-style pipeline but has a few differences with respect to SPA: (*i*) the decoder stage lacks the Thumb module and the coprocessor interface, (*ii*) the execute stage incorporates all the functional units present in SPA, (*iii*) nanoSpa implements user and supervisor operating modes only, lacking the other ARM modes, and (*iv*) nanoSpa does not support neither interrupts nor memory aborts. Although not easy to evaluate, these differences should not have a large impact on the relative performance of the two implementations.

Table 1 shows the results of the execution of the Dhrystone benchmark program for the original SPA and two different implementations of nanoSpa. The table includes results for bundled data and dual-rail versions. The performance of the original SPA is set as the reference.

The table shows that efficient pipeline control, true asynchronous operation, speculation and optimal combinational logic provide outstanding results. The basic nanoSpa cores, with the original handshake components, are a remarkable 2.21 (bundled data) and 2.84 (dual-rail) times faster that the original SPA implementations.

Table 1 also shows that the combination of the performance-driven specification with the use of the new handshake components results in very significant performance improvements, reaching 5.35 (bundled data) and 8.94 (dual-rail) times the performance of the original SPA.

The new techniques and handshake components provide a larger performance improvement in dual-rail implementations than in single-rail ones. This is in part due to the fact that the original dual-rail implementations were less efficient and, therefore, had more room for improvement.

Finally, Table 1 also shows the transistor counts for the different processor implementations. It is clear from the table that the new cores are significantly smaller than the original SPA implementations. The table also shows that the new handshake components, while significantly improving the performance of nanoSpa, have a relatively small impact on the size of the circuit. In fact, the dual-rail version with the new components is smaller. The large differences in transistor counts with respect to the original SPA indicate that there is enough room to incorporate the additional functionality without a large impact on the performance.

5. CONCLUSIONS

The work presented in this paper confirms that syntax-directed compilation is a powerful synthesis approach and, combined with an efficient set of handshake components, can automatically generate efficient asynchronous systems for complex, real world applications.

Extensive simulation results show that the introduction of new handshake components, used to implement parallel, sequential and input control, can double the performance of existing designs without the need to modify the source descriptions. Additionally, the introduction of new performance-oriented techniques used to implement efficient pipeline control, true asynchronous behaviour and speculative operation can triple the performance of existing designs. The combination of new components and techniques has been shown to generate a new implementation of an existing 32-bit, ARM-compatible processor with close to ten times the performance of the original one.

6. REFERENCES

[1] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. *IEICE Transactions on Information and Systems*, E80-D(3):315–325, March 1997.

[2] R. M. Fuhrer, S. M. Nowick, M. Theobald, N. K. Jha, and L. A. Plana. MINIMALIST: An environment for the synthesis and verification of burst-mode asynchronous machines. In *Proc. International Workshop on Logic Synthesis*, June 1998.

[3] TIMA Laboratory, Concurrent Integrated Systems Group. TAST: Tool for asynchronous circuit synthesis. *http://tima.imag.fr/cis/Tast/tast.html*, 2002.

[4] A. J. Martin. Programming in VLSI: From communicating processes to delay-insensitive circuits. In C.A.R. Hoare, editor, *Developments in Concurrency and Communication*, UT Year of Programming Series, pages 1–64, Addison-Wesley, Reading MA, 1990.

[5] A. Peeters and K. van Berkel. Single-rail handshake circuits. In *Proc. Working Conf. on Asynchronous Design Methodologies*, pages 53–62, May 1995.

[6] A. Bardsley. *Implementing Balsa Handshake Circuits*. PhD thesis, Department of Computer Science, University of Manchester, 2000.

[7] Q.Y. Zhang and G. Theodoropoulos. Towards an asynchronous MIPS processor. In *Cryptographic Hardware and Embedded Systems (CHES 2003)*, volume 2779 of *Lecture Notes in Computer Science*, pages 137–150. Springer-Verlag, 2003.

[8] Luis A. Plana, Sam Taylor, and Doug Edwards. Attacking control overhead to improve synthesised asynchronous circuit performance. In *Proc. International Conf. Computer Design (ICCD)*, pages 703–710. IEEE Computer Society Press, October 2005.