Retiming Finite State Machines to Control Hardened Data-Paths[*]

Ivan Augé, François Donnet and Frédéric Pétrot
ASIM Department of the LIP6
Université Pierre et Marie Curie
Paris, France

Abstract

We introduce Fine Grain Scheduling (FGS) as a post-processing step to circuits classically designed as a data-path controlled by a finite state machine (FSM). Such circuits may have timing errors, particularly if they are generated by High Level Synthesis (HLS) tools that make use of crude temporal estimates during scheduling. FGS reschedules the FSM to ensure correct execution at a requested frequency on the data-path.

The proposed algorithm takes into account all the electrical constraints of the data-path, namely propagation times, set-up and hold times of memorization elements, and even delays due to the interconnects if the data-path is placed and routed. Like HLS algorithms, FGS supports multi-operators cells, multi-cycle operators and chaining. However, it also makes use of mutli-cycles chaining to allow the chaining of several operators over several cycles without intermediate memorizations.

Experimentation of FGS on an MPEG2 Variable Length Decoder and a full MJPEG decoder has demonstrated that the approach is particularly well suited for the design of asynchronous coprocessors. Synchronous processors cannot be scheduled by FGS because the inputs and outputs dates are modified.

1 Introduction

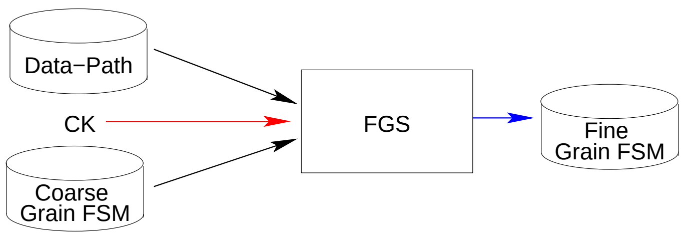
High Level Synthesis is expected to play a major rôle for embedded coprocessor design. Therefore, several HLS systems have been proposed in the last decade ([18, 7, 3] and many others). In these systems, the design is obtained automatically from high level description based on mainly two requirements: speed of execution of the target circuit and/or layout size. Unfortunately, most synthesis tasks are intractable in nature, hence the solutions found by the tools may not respect the design constraints. Due to the complexity of the problem, and in order to use designer insights, interactive synthesis environments have been devised[10, 9].

However, these tools are unable to cope with the delays induced by the physical wirering within the data-path, because it is a result of the synthesis. With the current technologies, the propagation time can only be roughly estimated by high level tools, as even in now well established 0.5 μm technologies, more than 40%[19] of the delays lies in the interconnect. Furthermore, the setup, hold and propagation times of the sequential resources are generally not taken into account. Ignoring these delays in a design mixing long and short timing paths induces race conditions for sampling the data, leading to possibly non-functioning circuits, independently of the clock period.

Likewise, the frequency of use of the target design is often a result of the synthesis [18, 4, 9, 16, 17] even though the running frequency of an ASIC is related to its environment and the other components with the which it communicates.

To ensure correct execution of the synthesized circuit, [11, 12] suggest to retime the data-path. It can unnecessarily retime false pathes and such approaches, especially the one based on procrastination, are costly in area.

In similar way, we introduce Fine Grain Scheduling (FGS) that reschedules the FSM.



[*]This work has been funded in part by the COSY 25443 Esprit Project.

Figure 1. FGS inputs and outputs.

As shown Figure 1, the input of FGS are:

1. The data-path with all timing informations, namely propagation times, set-up and hold times of memorization elements, and even delays due to the interconnects if the data-path is placed and routed,

2. The Coarse Grain FSM (CG-FSM) in the which only the sequence of operations is respected, not the precise timings. The states of this FSM correspond to the control-step in HLS,

3. The desired frequency for the rescheduled FSM.

Its output is a Fine Grain FSM (FG-FSM) that behaves like the CG-FSM but taking into account the timing information to run at the given frequency.

2 Related work

In [16], Parameswaran et al. share the concern of resynthesizing the controller of a circuit for a physically available data-path. Their work however mainly focuses on how to find an optimal clock for a given data-path, which we believe should be an input of the tool and not an output. They do not take the setup and hold times of the registers into account, leaving open the possibility of race conditions, and unfortunately do not detail their rescheduling algorithm.

Monahan and Brewer[14] have also noticed the importance of the interconnect delay problem in HLS. They start from an existing data-path at the RT level for the which they derive delay estimations, and allow the rebinding of connections. They propose a two phase approach in the which the automata is reevaluated to take estimated delay into account. However, the data-path is not hardened like in [16] or the current work, leading to possibly dreadful inaccuracies.

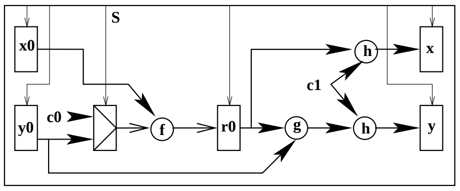
Juan et al. [9] present a environment that assist the HLS task. They still have the notion of critical path, and cannot lower the period below it. FGS would simply take more than a single cycle to execute the set of operation that lie on the critical path, without modification of the data-path. Also their work is not resource constrained, and they use a predictive floor-planning during refinement, that implies only a rough estimation of the delays.

Van Meerbergen et al. [21] use a regular target architecture template, that resemble much a VLIW processor, for the which delays can be more easily estimated. This is well suited for data intensive applications but is less useful for control dominated circuits because they are not dedicated to a given application domain.

The FGS algorithm is based on the notion of elementary transfers between registers. This notion is well known to processor designers [8], the only difference in HLS being that several operators can be chained between memorizations. However, unlike in [20] that also uses this notion, chaining can occur over several cycles *without* intermediate memorizations.

3 Fine Grain Scheduling

We show the algorithm on a small example. The Figure 2 presents the inputs of Fine Grain Scheduler: 1) a data-path with known electrical characteristics –Figure 2.a– , 2) the Figure 2.b gives the RTL instructions directly extracted from the CG-FSM control-steps. Those are called transfers, and their order matters. 3) a running frequency. The idea behind FGS is to reorganize the basic-blocks of the CG-FSM, moving instructions from one control-step to either a close control-step or to an added control-stepto begin with, and finally to suppress the useless control-steps.



a) Data-path

$t0$:	$r0{=}f(x0, y0)$	$t2$:	$r0{=}f(x0, c0)$
$t1$:	$y\ {=}h(c1, g(y0, r0))$	$t3$:	$x\ {=}h(c1, r0)$

b) Ordered list of transfers

Figure 2. inputs of the FGS algorithm

FGS deals with the scheduling of basic-blocks. Nevertheless it is different from the HLS scheduling algorithms[6] because the data-path of the operative part is completely defined. This fact induces that:

- Latency times are well known, and expressed in time units (e.g: tenths of nanosecond),
- The variables (respectively operators) have already been allocated to registers (respectively functional resources),
- And finally the knowlegde of the instructions reduces the set of execution pathes to the ones actually used.

Definitions

Transfer: A transfer is the motion of data from the outputs of a set of registers to the input of a target register. A transfer t is represented as a DAG, $D^t(V^t, A^t)$, whose vertices are operations and arcs are data dependencies as realized on the

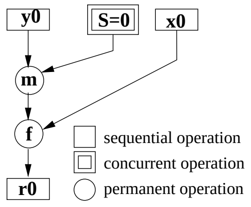


Figure 3. DAG representation of the $t0$ transfer

data-path. The Figure 3 shows the DAG of the $t0$ transfer of Figure 2. In this DAG, the rectangles represent the output of the control unit (memorized in **MIR**, the micro-instruction register), and the circles represent functional operations. A vertex is of three kinds:

COP: Concurrent OPerations do not modify the state of the data-path. For instance, changing the selection command of a multiplexer in a control-step only assigns **MIR**. The following control-step may restore the previous value and so restore the circuit in the previous state. They correspond to a value on bit fields of **MIR**. Two **COP**s are equivalent if they match the same bit field,

POP: Permanent OPerations always perform the same task and are associated to a single functional resource,

SOP: Sequential OPerations modify the state of the data-path. They perform memorization operation: once done, the overwritten value is lost. They usually correspond to a data-path register, and a bit field of **MIR**. Two **SOP**s are equivalent if they match the same bit field.

A transfer $D^t(V^t, A^t)$ has the following structural properties:

- V^t_{source} the set of vertices that have no predecessors. V^t_{source} only contains **COP** and **SOP**.

- V^t_{sink} the set of vertices that have no successors. $|V^t_{sink}| = 1$ and its element is a **SOP**.

- $V^t_{operator} = V^t - (V^t_{source} \cup V^t_{sink})$. All elements of $V^t_{operator}$ are **POP**s.

Transfer Graph: A transfer graph is a directed acyclic graph, $D(V, A)$, that represents the set of transfers that occur in the data-path for a given top level FSM transistion. The transfer graph is the concatenation of all transfers of the input list in the list order (Figure 2.b). The transfer D^t is added to the graph, and the vertices $v \in V^j_{source}$ are merged to most recently added equivalent vertices. Figure 4 shows the transfer graph resulting of the example of Figure 2.

Characterized Transfer: A characterized vertex is a vertex annotated with delays – see Figure 5.

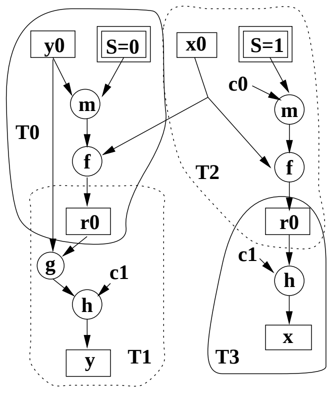


Figure 4. The transfer graph of Figure 2

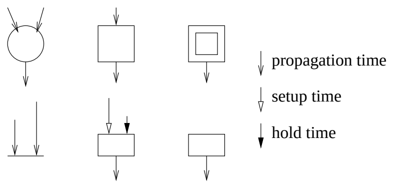


Figure 5. Characterized vertex

A **POP** vertex has a value associated to each couple of incoming and outcoming arc of the vertex. These values represent the set of propagation times of the corresponding physical cell.

A **COP** vertex has only one value associated to the outcoming arc, it corresponds to the propagation time from the clock to the **MIR** output bits associated to the **COP**.

A **SOP** vertex has 2 values associated to each incoming arc and 1 for each outcoming arc. They represent the set-up and hold times from the input relative to the clock and the propagation time from the clock to the output from the corresponding physical cell.

These values are delays extracted from the physical placed and routed data-path, so wire delays are taken into account.

The characterized transfer is obtained by replacing the original transfer vertices by characterized vertices. Figure 6 shows the characterized transfer of the transfer presented Figure 3. The values of the characterized vertices are graphically represented by the plain arrows.

Characterized Transfer Graph: It is obtained from the transfer graph by replacing transfers with characterized transfers. Nevertheless other arcs must be added to correctly model the behavior of the initial transfer sequence. These arcs implement the WAR (Write After Read) and WAW (Write After Write) precedence relations [15].

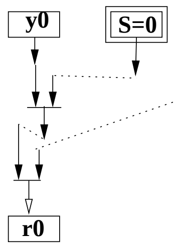


Figure 6. Characterized DAG of the $t0$ transfer

- The RAW relation denotes the data dependencies.
- The WAR relation express the fact that 2 equivalent **COP**s are used with different values. In our example this occurs for S=0 in the $t0$ transfer S=1 in the $t2$ transfer. S can be set to 1 only when this will not disturb the $t0$ transfer. This gives the arc from the r0 hold time to S=1 propagation time.
- The WAW relation indicates that two equivalent **SOP**s are used within two different transfers. In the example, r0 is used simultaneously in $t0$ and $t2$. The **SOP** of $t2$ must be performed after the **SOP** of $t0$, is because the same register cannot be loaded twice in the same cycle.

The resulting graph is plotted in Figure 7, with the previous relations outlined.

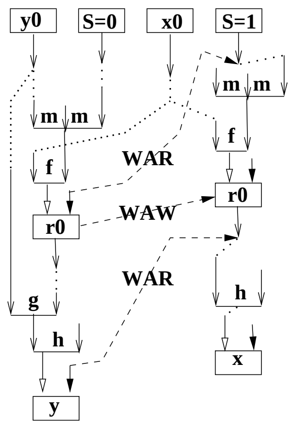


Figure 7. Characterized transfer graph

Scheduling the Characterized Transfer Graph

The scheduling rules are:

R1 Load a given register only once in a given cycle,

R2 Loading a register must respect its set-up time,

R3 Loading a register must not violate the hold time.

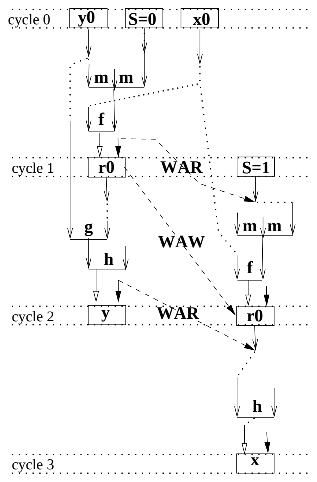


Figure 8. Scheduled transfer graph

The clock period defines a grid on the which the **SOP**s and the **COP**s must be *snapped*. A simple ASAP [13] algorithm with the constraint that all arcs points downwards (Figure 8) produces a scheduling that verifies all the scheduling rules. This *pointing downwards* relation is either combinational when concurent operations are involved or sequential when a permanent operation is involved.

Rule **R1** is enforced by the arcs implementing the WAW relations. Rule **R2** is enforced by RAW relations (data dependencies: the plain arrows). Rule **R3** is enforced by the the arcs implementing the WAR relations.

This scheduling allows all kinds of chaining and especially multi-cycles chaining without intermediate memorizations.

The only delays not taken into account are: a) the propagation time from the FSM state register to the data-path, and b) the propagation time from the data-path to the FSM state register. Point a) is solved because the control unit is a Moore FSM with a **MIR** that synchronizes the control signals and that we assume that the delays due to routing capacitances between the **MIR** and the operator command inputs are similar. This can be ensured by increasing the fan-out of the **MIR** buffers. Point b) is only due to the conditional branches of the FSM. We solve it by setting an upper bound on the FSM transition function. This upper bound becomes the timing constraint of the FSM synthesis tool.

4 Experimental results

We have experimented FGS in the User Guided High Level Synthesis (UGH) tools [1]. Figure 9 shows the UGH synthesis scheme. The specification is given by a behavior (either C or VHDL subset) and a draft data-path that mainly defines the allowed functional (apart from the multiplexers) and memorization ressources. UGH-CGS produces a structural synthesizable data-path and a CG-FSM. The structural data-path is built from the draft data-path by adding multiplexers and defining the size of each resources. The CG-FSM is build by paralellizing the behavior on the resources defined in the draft data-path. This CG-FSM corresponds to the initial behavior assuming that all resources have propagation times smaller than the cycle period.

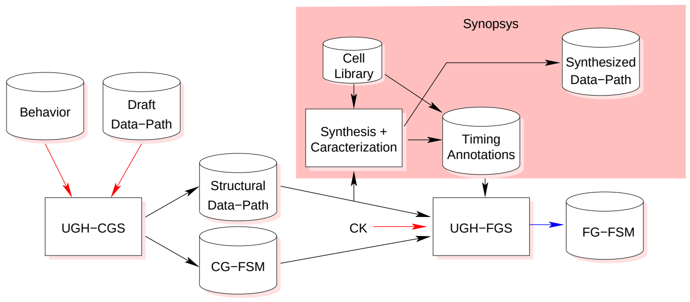


Figure 9. UGH synthesis scheme

The structural data-path is then synthesized using Synopsys. All delays (propagation, set-up and hold times) are extracted. These delays are used by FGS to retime the CG-FSM. The complete circuit is the synthesized data-path and the result of the synthesis of the FG-FSM.

We have experimented FGS with a *Motion JPEG* application. It contains five blocks that covers application kind from control-dominated to data-dominated. The results presented here are issued from a MPEG2 VLD decoder that we have developed in the COSY project[5] It performs Huffmann decoding and is very control dominated.

The **C** behavioral description is 300 lines long (the equivalent VHDL is 400 lines long), the draft data-path is 200 lines. We run GCS with this specification. We get the CG-FSM with 136 states and the structural data-path with approximatively 50 components. These components are 8 arithmetic operators (from 16 to 6 bits wide), one 32 bit shifter, 33 muxes (from 12 to 2 inputs), and 4 ROMs.

The structural data-path is synthesized on a 0.25 μm standard cell library using Synopsys in 4440 gates with a critical path of 15.8ns.

We run FGS for a frequency range from 20 MHz to 200 MHz. The Figure 10.a plots the number of states as a function of the frequency. When the frequency increases, the number of states increases too to fit the data-path delays. Up to 40 MHz, the number of states is constant because the data dependencies constraints are stronger than the timing constraints.

We simulate the circuit made of the FG-FSM and the synthesized data-path using a MPEG2 stream containing a few images. The Figure 10.b represents the execution time as a function of the frequency. We can see that we don't have an hyperbolic curve but a succession of hyperbolæ. This is due to the more often executed algorithmic sequences. Those sequences contain pathes that require 1, 2 or more cycles to be executed depending on the given frequency.

These results show that there are two solutions to support the 20-200MHz range.

1. A single circuit: the one that runs at 200MHz. This will be lesser and lesser efficient for the lower frequencies, as shown on Figure 10.b. For instance, the 200 MHz scheduled VLD running a 20 MHz is three times slower than the 20 MHz scheduled one,

2. Six different circuits per frequency range: 0-50, 50-60, 60-80, 80-100, 100-120, and 120-200 MHz.

Furthermore, the curve shows that the fabrication process induces unsecure frequencies, for example around 80 MHz. So it is advised to fine schedule at 100 MHz to get a 80 MHz circuit.

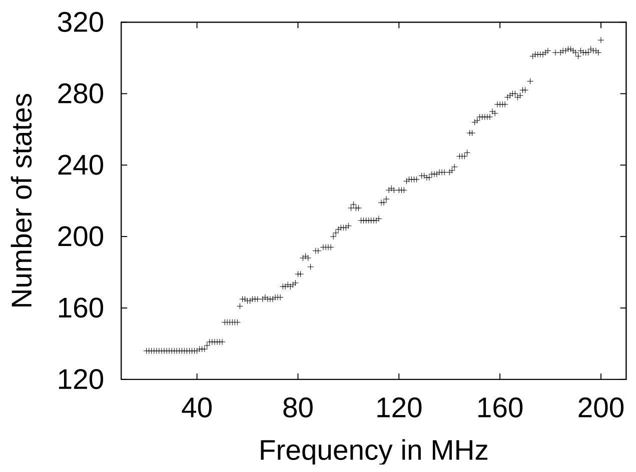
The same kind of curves can be observed also when scheduling data-dominated circuits, such as the inverse discrete cosine transform also used in MPEG decoding.

5 Conclusion

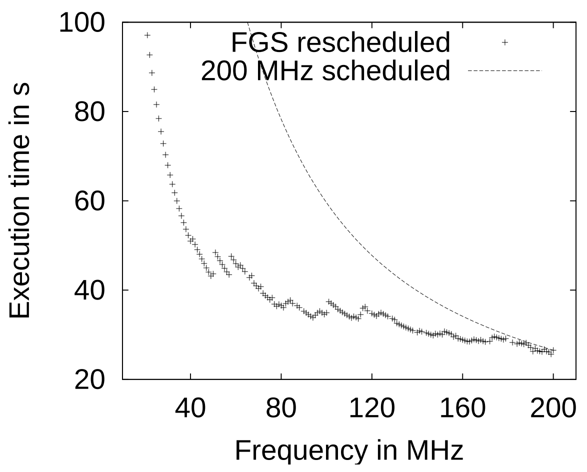
We have defined and developped the retiming of finite state machines to control hardened data-path as a post processing of the UGH tool suite.

However, it can also be useful for other HLS tools, to tune the synthesized circuit to the electrical constraints. Also, it can used after RTL synthesis, like Synopsys design_compiler, especially when it did not achieve the frequency constraint. FGS also permits to adapt an existing design to a higher or lower frequency. Moving to higher frequencies, it allows to have a correct circuit. Moving to lower frequencies, it allows to optimize execution times.

The main advantage of this approach is to preserve the area of the circuit when the frequency gap is reasonnable, because at worst FGS add a few states and transitions to the FSM.



a) Number of states



b) Execution times

Figure 10. FGS experimentations on the VLD example

The main restriction is that FGS can only be used to retime processors with asynchronous interfaces to their environments, since the communication dates are determined by FGS as a function of the data-path delays and the clock period.

Finally, our FGS implementation is distributed under GPL licence within the Disydent framework[2].

References

[1] I. Augé, R. K. Bawa, P. Guerrier, A. Greiner, L. Jacomme, and F. Pétrot. User guided high level synthesis. In R. Reis and L. Claensen, editors, *VLSI: Integrated Systems on Silicon*, pages 464–475, Gramado, Brazil, Aug. 1997. IFIP, Chapman & Hall.

[2] I. Augé and F. Pétrot. Digital system design environment, Nov. 2002. `http://www-asim.lip6.fr/disydent`.

[3] R. A. Bergamaschi and A. Kuehlman. A system for production use of high-level synthesis. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 1(3):233–243, 1993.

[4] J. Biesenack, M. Koster, et al. The siemens high-level synthesis system callas. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 1(3):244–253, 1993.

[5] E. de Kock, G. Essink, W. Smits, P. van der Wolf, J.-Y. Brunel, W. Kruijtzer, P. Lieverse, and K. Vissers. YAPI: Application modeling for signal processing systems. In *37th Design Automation Conference*, pages 402–405, 2000.

[6] D. D. Gajski, N. Dutt, A. Wu, and S. Lin. *High Level Synthesis, Introduction to Chip and System Design*, chapter 1. Kuwer Academic Publisher, 1992.

[7] J. Hallberg and Z. Peng. Synthesis under local timing constraints in the camad high-level synthesis system. In *the 21st Euromicro Conference*, Como, italy, aug 1995. IEEE.

[8] J. L. Hennessy and D. A. Patterson. *Computer architecture, a quantitative approach*. Morgan Kaufmann Publisher, Inc, 1990.

[9] H.-P. Juan, D. D. Gajski, and V. Chaiyakul. Clock-driven performance optimization in interactive behavioral synthesis. In *the ICCAD*, pages 154–157, San José, CA, Nov. 1996. IEEE.

[10] D. W. Knapp. Manual rescheduling and incremental repair of register level data paths. In *the ICCAD*, pages 58–61. IEEE, Nov. 1989.

[11] C. E. Leiserson and J. B. Saxe. Optimizing synchronous systems. *Journal of VLSI and Computer Systems*, 1(1):41–67, spring 1983.

[12] G. D. Micheli. Synchronous logic synthesis: Algorithms for cycle-time minimization. *IEEE Transactions on CAD/ICAS*, 10(1):63–73, Jan. 1991.

[13] G. D. Micheli. *Synthesis and Optimization of Digital Circuits*, chapter 9, page 441. McMcGraw-Hill, Inc., 1994.

[14] C. Monahan and F. Brewer. Concurrent analysis techniques for data path timing optimization. In *the DAC*, Las Vegas, NV, June 1996. IEEE.

[15] B. M. Pangrle and D. D. Gajski. Design tools for intelligent silicon compilation. *IEEE Transactions on Computer Aided Design*, 6(6):1098–1112, 1987.

[16] S. Parameswaran, P. Jha, and N. Dutt. Resynthesizing controllers for minimum execution time. In *the 2nd ASP-CHDL Conference*, pages 111–117, 1994.

[17] S. Park and K. Choi. Performance-driven scheduling with bit-level chaining. In *the DAC*, pages 286–291, New Orleans, LA, June 1999. IEEE.

[18] M. Rahmouni, K. O'Brien, and A. A. Jerraya. A loop-based scheduling algorithm for hardware description languages. *Parallel Processing Letters*, 4(3):351–364, 1994.

[19] J. Y. Sayah, R. Gupta, D. D. Sherlekar, P. S. Honsinger, J. M. Apte, S. W. Bollinger, H. H. Chen, S. DasGupta, E. P. Hsieh, A. D. Huber, E. J. Hughes, Z. M. Kurzum, V. B. Rao, T. Tabtieng, V. Valijan, and D. Y. Yang. Design planning for high-performance asics. *IBM Journal of Research and Development*, 40(4):431–452, July 1996.

[20] S. Tarafdar and M. Leeser. The DT-model: High level synthesis using data transfers. In *the DAC*, pages 114–119, San Francisco, CA, June 1998. IEEE.

[21] J. L. van Meerbergen, P. E. R. Lippens, W. F. J. Verhaegh, and A. van der Werf. Phideo: High-level synthesis for high throughput applications. *Journal of VLSI Signal Processing*, 9(1-2):89–104, May 1995.