Operand Registers and Explicit Operand Forwarding

James Balfour, R. Curtis Harting and William J. Dally *Fellow*
Computer Systems Laboratory, Stanford University, Stanford, CA, USA
{jbalfour,dally}@cva.stanford.edu

***Abstract*—Operand register files are small, inexpensive register files that are integrated with function units in the execute stage of the pipeline, effectively extending the pipeline operand registers into register files. Explicit operand forwarding lets software opportunistically orchestrate the routing of operands through the forwarding network to avoid writing ephemeral values to registers. Both mechanisms let software capture short-term reuse and locality close to the function units, improving energy efficiency by allowing a significant fraction of operands to be delivered from inexpensive registers that are integrated with the function units. An evaluation shows that capturing operand bandwidth close to the function units allows operand registers to reduce the energy consumed in the register files and forwarding network of an embedded processor by 61%, and allows explicit forwarding to reduce the energy consumed by 26%.**

***Index Terms*—energy efficient register organization, operand registers, explicit operand forwarding, embedded processor**

I. Introduction

ENERGY consumption in processors is dominated by communication, specifically data and instruction movement, not computation. Consequently, even low-power programmable processors consume significantly more energy than dedicated fixed-function hardware, which allows the communication of data between function units to be aggressively optimized. This is particularly problematic in embedded systems because performing common operations, such as fixed-point arithmetic and logic operations, is inexpensive compared to delivering data and instructions to the function units. The register file alone can account for 16% of the energy consumed in an embedded processor, and 42% of the datapath energy [3]. The situation does not improve with advances in semiconductor technology: communication benefits less than computation from improvements in semiconductor technology, and the interconnect-dominated register files and buses that deliver instructions and data to the function units will continue to consume an increasing fraction of the energy.

This paper describes a register organization that uses operand registers and explicit operand forwarding to reduce the energy consumed staging operands in registers. Operand registers extend a conventional register organization with small, distributed sets of inexpensive general-purpose registers, each of which is integrated with a single function unit in the execute stage. The shallow, inexpensive register files that implement the operand registers effectively extend the pipeline registers that precede the function units into small register files, preserving the low access energy of the pipeline registers while satisfying a greater fraction of operand references. This lets software capture short-term reuse and instruction-level producer-consumer locality near the function units. Explicit operand forwarding lets software control the routing of operands through the forwarding network so that ephemeral values need not be written to registers. Both mechanisms increase software control over the movement of operands between function units and register files. An evaluation demonstrates that operand registers reduce the energy consumed in the register files and forwarding network of an embedded processor by 61% and explicit forwarding reduces the energy consumed by 26%. Significantly, less energy is consumed staging the operands to an instruction in operand registers than is consumed performing common fixed-point arithmetic operations.

Manuscript submitted: 5-May-2009. Manuscript accepted: 23-Jun-2009. Final manuscript received: 28-July-2009.

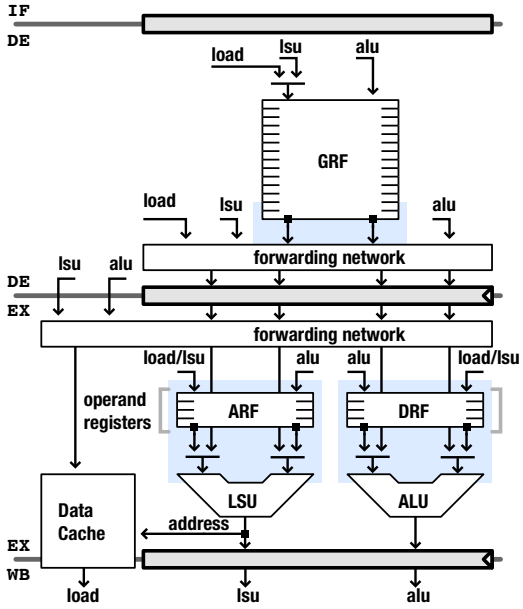

Fig. 1. The address register file (ARF) and data register file (DRF) are tiny (4-entry) register files that are integrated with the function units. The load-store unit LSU executes memory operations and simple arithmetic operations. The arithmetic and logic unit ALU executes simple and complex arithmetic instructions. Reference filtering reduces demand for operand bandwidth at the general-purpose register file, which allows the number of ports to be reduced.

Explicit operand forwarding and operand registers are described in the following section, which is followed by an evaluation of how these mechanisms reduce the energy consumed delivering operands. We consider related work and then conclude.

II. Register Organization and Compilation

The function unit and register organization of the embedded processor used in the evaluation are illustrated in Fig. 1. The LSU performs memory operations, such as load and store; the ALU performs complex arithmetic operations, such as shift and multiplication. Both function units perform simple arithmetic operations; consequently, many common instructions may execute in either function unit.

The processor has been designed in a 45 nm CMOS process. The process has been tailored for low-power applications, and provides transistors with thick gate oxides to reduce leakage. Table I lists the energy consumed performing basic arithmetic operations and transferring operands and results between function units and register files. The energy is derived from HSPICE simulations that include device and interconnect capacitances that were extracted after layout. Observe that less energy is consumed performing a 32-bit addition than reading an operand from the 32-entry general-purpose register file.

Posted to the IEEE & CSDL on 7/29/2009
DOI 10.1109/L-CA.2009.45 1556-6056/09/$25.00 © 2009 Published by the IEEE Computer Society

A. Explicit Operand Forwarding

Consider the following code and the corresponding instruction sequence generated during compilation.

```
int x = a + b - c;
-----------------
add %r5 = %r2 + %r3;
sub %r1 = %r5 - %r4;
```

The compiler-introduced temporary assigned to register `%r5` is delivered by the forwarding network, but register `%r5` is unnecessarily written when the code executes. Register `%r5` simply establishes a name that allows the forwarding logic to detect that the result of the `add` instruction must be forwarded to the `sub` instruction. Writing the intermediate result to the register file consumes more energy than performing the operation that produced it: the 32-bit addition consumes 0.52 pJ; writing the result back to the register file consumes 1.45 pJ. Furthermore, this use of register `%r5` unnecessarily increases register pressure.

Explicit operand forwarding lets software orchestrate the routing of operands through the forwarding network so that ephemeral values need not be written to registers. Explicit operand forwarding introduces the concept of a *forwarding register* to provide an explicit name for the result of the last instruction that a function unit executed. Forwarding registers establish architectural names for the physical pipeline registers residing at the end of the execute (EX) stage. Software uses forwarding registers as operands to explicitly forward the result of a previous instruction, and directs the result of an operation to a forwarding register to indicate that the result should not be written to the register file. Revisiting the example, the following code sequence explicitly forwards the result of the `add` operation through forwarding register `%t0`, which is associated with the ALU.

```
int x = a + b - c;
-----------------
add %t0 = %r2 + %r3;
sub %r1 = %t0 - %r4;
```

Forwarding registers retain the result of the last instruction that wrote the execute (EX) pipeline register. To preserve the registers, datapath control logic clock-gates the pipeline registers during interlocks and when NOPs execute. This allows operands to be explicitly forwarded between instructions that execute multiple cycles apart when intervening instructions do not update the physical registers used to communicate the operands [8]. The results of load instructions cannot be explicitly forwarded because load values do not traverse the pipeline registers at the end of the execute (EX) stage; instead, architectural registers retime load values arriving from memory.

Processors with deeper pipelines generally need more registers to cover longer operation latencies. Explicit forwarding can be extended for deeper pipelines by establishing additional explicit names for the results of recently executed instructions that are available within the forwarding network. However, the pipeline registers that stage the forwarded operands are not conventional architectural registers and may be difficult to preserve when interrupts and exceptions occur. When interrupts are not a concern, the compiler can be directed to disallow the explicit forwarding of operands between instructions that may cause exceptions.

B. Operand Registers

Operand registers are implemented as very small register files that are integrated with the function units in the execute stage of the pipeline. This allows operand registers to provide the energy efficiency of explicit forwarding while preserving the behavior of general-purpose registers. Operand registers let software explicitly capture short-term reuse and locality close to the function units. Like explicit operand forwarding, operand registers reduce the fraction of the operand bandwidth that reaches the general-purpose register file. This improves energy efficiency: the small operand register files are inexpensive to access, and the 0.2 pJ cost of traversing the forwarding network and pipeline register preceding the execute (EX) stage is avoided. For example, delivering two operands from a general-purpose register file and writing back the result consumes 3.1 pJ, which exceeds the 2.2 pJ consumed by a 16-bit multiplication; using operand registers to stage the operands and result consumes only 0.78 pJ, which is close to the 0.52 pJ consumed by a 32-bit addition. Accessing the operand register files in the execute (EX) stage may contribute to longer critical path delays. However, the operand register read occurs in parallel with the activation of the forwarding network, and much of the operand register file access time is covered by the forwarding of operands from the write-back (WB) stage. Regardless, register organizations using operand registers require fast, and consequently small, operand register files to avoid creating critical paths in the execute stage.

TABLE I
ENERGY CONSUMED BY COMMON OPERATIONS

Arithmetic Operations		Relative Energy
32-bit addition	520 fJ	1×
16-bit multiply	2,200 fJ	4.2×
General-Purpose Register File — 32 words (4R+2W)		
32-bit read	830 fJ	1.6×
32-bit write back	1,450fJ	2.8×
General-Purpose Register File — 32 words (2R+2W)		
32-bit read	800 fJ	1.5×
32-bit write back	1,380fJ	2.7×
Operand Register File — 4 words (2R+2W)		
32-bit read	120 fJ	0.2×
32-bit write back	540 fJ	1.0×
Forwarding Register — 1 word (1R+1W)		
32-bit forward	530 fJ	1.0×
Data Cache — 2K words (1R+1W) [8-way set-associative with CAM tags]		
32-bit load	10,100 fJ	19.4×
32-bit store	14,900 fJ	28.6×

Each operand register is assigned to a single function unit, and only its associated function unit can read it. The function units can write any register, which allows the function units to communicate through operand registers or the more expensive backing registers. This organization allows software to keep a significant fraction of the operand bandwidth local to the function units, to place operands close to the function unit that will consume them, and to keep data produced and consumed on the same function unit local to the function unit. Furthermore, reference filtering by the operand registers reduces demand for operand bandwidth from the shared general-purpose registers, which allows the number of read ports to the general-purpose register file to be reduced without adversely impacting performance. This makes the general-purpose register file smaller and shortens its bit-lines, reducing its access costs by decreasing the bit-line capacitance switched during read and write operations. It also reduces the cost of updating a register because fewer access devices contribute to the capacitance internal to each register.

C. Compilation

Explicit operand forwarding and operand registers require additional compiler passes and optimizations. An explicit operand forwarding compiler pass can be implemented as a peephole optimization performed after instruction scheduling and register allocation. However, the pass is better performed between instruction scheduling and register allocation to avoid allocating registers to variables that can be explicitly forwarded. Typically, variables that can be explicitly forwarded are also ideal candidates for operand registers and will displace other variables competing for operand registers; consequently, performing an explicit forwarding pass before register allocation

```
x.re = a.re * b.re - a.im * b.im;
x.im = a.re * b.im + a.im * b.re;
-------------------------------------------------------
t1 = load [a.re];      [defs=1 uses=2 life=8 int=0.38]
t2 = load [a.im];      [defs=1 uses=2 life=8 int=0.38]
t3 = load [b.re];      [defs=1 uses=2 life=7 int=0.43]
t4 = load [b.im];      [defs=1 uses=2 life=5 int=0.60]
t5 = mult t1 * t3;     [defs=1 uses=1 life=2 int=1.00]
t6 = mult t2 * t4;     [defs=1 uses=1 life=1 int=2.00]
t7 = sub t5 - t6;      [defs=1 uses=1 life=1 int=2.00]
[x.re] = store t7;
t8 = mult t1 * t4;     [defs=1 uses=1 life=2 int=1.00]
t9 = mult t2 * t3;     [defs=1 uses=1 life=1 int=2.00]
t10 = add t8 + t9;     [defs=1 uses=1 life=1 int=2.00]
[x.im] = store t10;
; data-vars    = [t10 t9 t7 t6 t8 t5 t4 t3 t2 t1]
; address-vars = [a b x ]
```

Fig. 2. Operand Intensity. The code computes the product of two complex values. The order in which variables are assigned registers is listed at the bottom. Data register candidates `t10`, `t9`, `t7`, and `t6`, which are prioritized because they have the greatest operand intensity, could be explicitly forwarded.

allows more variables to be assigned to operand registers and achieves better operand register utilization.

Registers are allocated using a conventional register allocator that accounts for the restricted connectivity between the register files and function units. Variables that are read by instructions that execute on different function units are assigned to general-purpose registers. The compiler preferentially schedules dependent instructions on the same function unit to expose opportunities for using the local operand registers to stage intermediates between dependent instructions.

Intuitively, the operand bandwidth captured in operand registers can be increased by allocating operand registers to variables that are short-lived and to variables that are long-lived and frequently accessed. The register allocator can identify variables that exhibit these properties by computing a measure of *operand intensity* for each variable. Operand intensity is computed by dividing the aggregate number of definitions and uses of a variable by its lifetime; when an access appears inside a loop, the depth of the loop can be used to weight the access, and the lifetime can be estimated as the number of instructions at which point the variable is live, as computed by a conventional live-variable data-flow analysis. An example of the computation appears in Fig. 2.

III. Evaluation and Analysis

The kernels used in the evaluation were written in C and compiled with the front-end of the LLVM [4] port of GCC. The instruction selection, instruction scheduling, register allocation, and back-end optimization passes were implemented for the target architecture, and use the scheduling and register allocation algorithms described above. The register allocator actively coalesces variables introduced by the single static assignment form used in the front-end. The compiled benchmarks were executed on a cycle-accurate simulator to collect access statistics from which average operand access energies are derived (Fig. 3). The 7 embedded kernels used in the evaluation are: **`aes`**, an optimized implementation of the advanced encryption standard block cipher with 10 rounds and a 128-bit key; **`conv2d`**, a 5×5 RGB image filter; **`crosscor`**, a cross-correlation filter of two fixed-point signals at 16 discrete time lags; **`fft`**, a 1024-point Fast Fourier Transform of a 16-bit fixed-point real-valued signal; **`fir`**, a 32-tap finite impulse response filter of a 16-bit fixed-point signal; **`jpeg`**, a JPEG image compression routine; **`rgb2yuv`**, an RGB to YUV conversion routine; and **`viterbi`**, an implementation of the Viterbi decoder used in the GSM standard for voice data.

We keep the aggregate number of registers constant by reducing the number of general-purpose registers available to the compiler when operand registers are added. Decreasing the capacity of the general purpose register file reduces the energy expended accessing

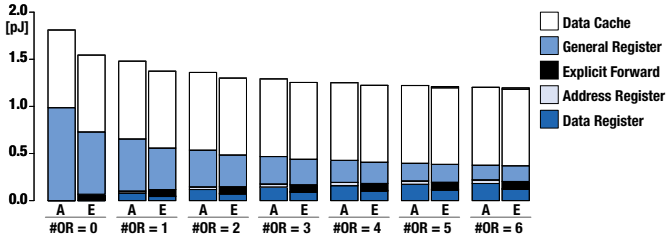

Fig. 3. Breakdown of data access energy. The number of operand registers per function unit (**OR**) is shown below each column. (**A**) automatic forwarding; (**E**) explicit forwarding.

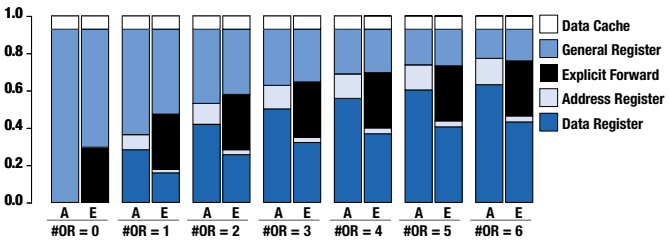

Fig. 4. Breakdown of data bandwidth. The number of operand registers per function unit (**OR**) is shown below each column. (**A**) automatic forwarding; (**E**) explicit forwarding.

it; however, we conservatively exclude reductions in the cost of accessing the general-purpose register file when its capacity decreases from our evaluations and analysis to isolate improvements due to the operand registers. Explicit forwarding and operand registers together reduce the energy consumed in the registers and forwarding network by 62% and reduce the average energy consumed by 34%. Most of the improvement is due to the operand registers. Explicit forwarding alone reduces the register and forwarding energy 26%, which results in a 15% reduction in data delivery energy. Operand registers reduce the energy consumed in the registers and forwarding network by 61%, which corresponds to a 34% reduction in the energy consumed staging operands.

The breakdown of data bandwidth appears in Fig. 4. Explicit forwarding alone keeps 29.6% of the bandwidth local to the function units; providing one operand register per function unit keeps 36.6% of the bandwidth local. The single operand register configuration captures a greater fraction of the bandwidth because unrelated instructions may intervene between the producing and consuming instructions. The fraction of the operand bandwidth captured in operand registers increases with the number of operand registers; with 4 operand registers per function unit, 68.9% of the operand bandwidth is captured by operand registers, which allows the number of general-purpose register file read ports to be reduced to two without affecting performance. Further increasing the number of operand registers increases bandwidth capture beyond 77.2%, but additional spills appear in kernels with significant register pressure, such as **`fft`**. These spills appear because communication between the operand registers and function units is limited.

The reduction in address register bandwidth observed when operands are explicitly forwarded (Fig. 4) reflects an abundance of ephemeral intermediate values within address calculations preceding load and store instructions. The structure of data dependencies and working sets affects the extent to which code benefits from explicit forwarding and operand registers, as the **`aes`** and **`rgb2yuv`** kernels illustrate. Whereas **`aes`** is dominated by long sequences of data-dependent instructions, **`rgb2yuv`** exhibits many short sequences of instructions with related data-dependencies, and the RGB samples form a critical working set that can be captured in a small number of

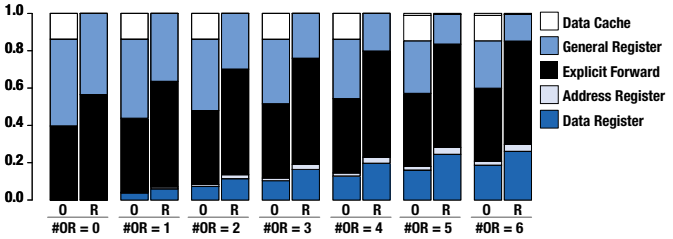

Fig. 5. Breakdown of **aes** operand (O) and result (R) bandwidth.

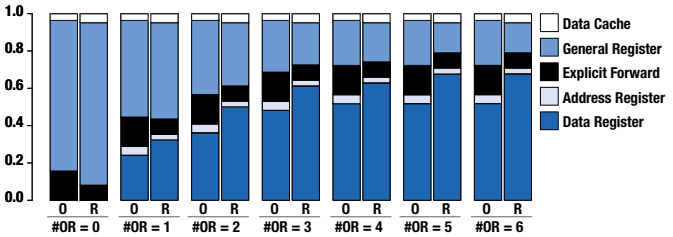

Fig. 6. Breakdown of **rgb2yuv** operand (O) and result (R) bandwidth.

registers. Consequently, the **aes** kernel benefits more from explicit forwarding, while the **rgb2yuv** kernel benefits more from operand registers (Fig. 7). The breakdown of the operand bandwidth illustrates the differences between the working sets of the two kernels. The **aes** kernel lacks small working sets with significant short-term locality and reuse for the compiler to promote to operand registers; consequently, operand register bandwidth in **aes** increases slowly as additional operand registers are introduced (Fig. 5). Conversely, the **rgb2yuv** kernel exhibits an abundance of small, critical working sets that the compiler is able to capture with a limited number of operand registers; consequently, operand register bandwidth increases rapidly, and four data and four address registers are sufficient to capture all the critical working sets of **rgb2yuv**. Further increasing the number of operand registers results in little additional improvement, as only low-intensity variables are available for promotion to operand registers.

Explicit forwarding and operand registers provide greater benefits when VLSI technology and design constraints limit the efficiency of multi-ported register files and memories. For example, memory compilers are rarely designed to generate efficient small memories, which increases the cost of staging operands in general-purpose registers and improves the benefits of operand registers and explicit forwarding. Similarly, standard cell libraries rarely allow wide multiplexers to be implemented efficiently, which may increase the cost of implementing operand registers and make explicit forwarding more attractive.

IV. Related Work

Distributed [6], clustered [2], banked [1], and hierarchical [9] register organizations improve access times and densities by partitioning registers across multiple register files and reducing the number of read and write ports to each register file. The number of ports needed to deliver a certain operand bandwidth can be reduced by using SIMD register organizations [5] to allow each port to deliver or receive multiple operands to multiple function units. The hierarchical CRAY-1 register organization [7] associates dedicated scalar and address registers with groups of scalar and address function units and provides dedicated backing register files to filter spills to memory. In contrast, operand register files are significantly smaller, optimized for access energy, and each is integrated with a single function unit in the execute stage of the pipeline to reduce the cost of transferring operands between an operand register file and its dedicated function unit. This allows operand registers to achieve greater reductions in energy consumption in processors with short

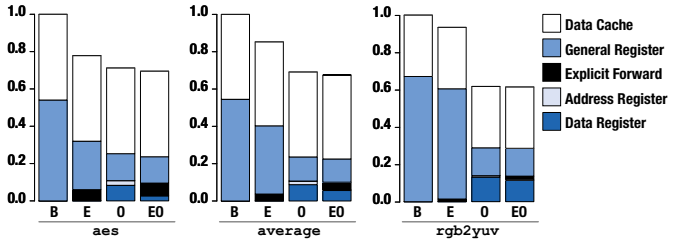

Fig. 7. Normalized data access energy. **(B)** baseline; **(E)** explicit forwarding; **(O)** 4 operand registers; **(EO)** explicit forwarding and 4 operand registers.

operation latencies and workloads with critical working sets and short-term producer-consumer locality that can be captured in a small number of registers. Operand registers further differ by allowing both levels of the register hierarchy to be directly accessed. This avoids the overhead of executing instructions to explicitly move data between the backing register files and those accessible by the function units [9], and avoids the overhead of replicating data in multiple register files [1].

Horizontally microcoded machines such as the FPS-164 [8] required that software explicitly route all results and operands between function units and register files. This work introduces opportunistic explicit software forwarding into a conventional pipeline, where automatic forwarding performed by the forwarding control logic avoids the increase in code size and dynamic instruction counts needed to explicitly route all data.

V. Conclusion

Operand register files are small, inexpensive register files that are integrated in the execute stage of the pipeline; explicit operand forwarding lets software opportunistically orchestrate the routing of operands through the forwarding network to avoid writing ephemeral values to registers. Both let software capture short-term reuse and locality in inexpensive registers that are integrated with the function units. This keeps a significant fraction of operand bandwidth local to the function units, and reduces the energy consumed communicating data through registers and the forwarding network by 62%.

References

[1] J.-L. Cruz, A. González, M. Valero, and N. P. Topham, "Multiple-banked register file architectures," in *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, 2000, pp. 316–325.

[2] K. I. Farkas, P. Chow, N. P. Jouppi, and Z. Vranesic, "The multicluster architecture: Reducing processor cycle time through partitioning," *Int. J. Parallel Program.*, vol. 27, no. 5, pp. 327–356, 1999.

[3] D. Gonzales, "Micro-RISC architecture for the wireless market," *Micro, IEEE*, vol. 19, no. 4, pp. 30–37, Jul-Aug 1999.

[4] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[5] M. Pericàs, E. Ayguadé, J. Zalamea, J. Llosa, and M. Valero, "Power-efficient VLIW design using clustering and widening," *International Journal of Embedded Systems*, vol. 3, no. 3, pp. 141–149, 2008.

[6] S. Rixner, W. J. Dally, B. Khailany, P. R. Mattson, U. J. Kapasi, and J. D. Owens, "Register organization for media processing," in *HPCA 6: Proc. of the Sixth International Symposium on High-Performance Computer Architecture*, 2000, pp. 375–386.

[7] R. M. Russell, "The CRAY-1 computer system," *Commun. ACM*, vol. 21, no. 1, pp. 63–72, 1978.

[8] R. F. Touzeau, "A fortran compiler for the FPS-164 scientific computer," in *SIGPLAN '84: Proceedings of the 1984 SIGPLAN symposium on Compiler construction*, 1984, pp. 48–57.

[9] J. Zalamea, J. Llosa, E. Ayguadé, and M. Valero, "Two-level hierarchical register file organization for VLIW processors," in *MICRO 33: Proc. of the 33rd annual ACM/IEEE International Symposium on Microarchitecture*. ACM Press, 2000, pp. 137–146.