Universität Dortmund
Lehrstuhl Informatik XII
44221 Dortmund

Moving Program Objects to Scratch-Pad Memory for Energy Reduction

Stefan Steinke, Christoph Zobiegala,
Lars Wehmeyer, Peter Marwedel

Technical Report # 756

10.4.2001

Moving Program Objects to Scratch-Pad Memory for Energy Reduction*

Stefan Steinke, Christoph Zobiegala, Lars Wehmeyer, Peter Marwedel

{steinke | zobiegal | wehmeyer | marwedel}@ls12.cs.uni-dortmund.de

University of Dortmund, Dept. of Computer Science

Otto-Hahn-Strasse 16

44221 Dortmund, Germany

Abstract

This paper presents a new approach for improving energy consumption of compiler generated software by using on-chip Scratch-Pad RAM more efficiently. This memory allocation technique moves program parts (functions or basic blocks and global data objects) into the limited Scratch-Pad RAM.

Experimental results show that this technique saves up to 80% of the total energy consumption depending on the application, the system architecture and the size of the Scratch-Pad RAM.

Keywords

Compiler, On-Chip RAM, Energy Consumption.

1 Introduction

The number of applications using embedded processors is steadily increasing. Their flexibility and size allows arbitrary applications on very small space with an increasing computing power. This enables the development of new applications like car navigation systems or redesigned systems, e.g. electronic control units for engine management.

Mobile devices like PDAs, mobile phones, digital cameras or eBooks have limited running time due to the capacity of their batteries. Other products like notebooks are furthermore limited concerning the heat dissipation of their processors.

New technical developments are being implemented to overcome these limitations. The fabrication technology of VLSI circuits is steadily improving and the chip structures are being scaled down. But the number of transistors on a chip is increasing at a higher ratio and demands further improvements. Besides voltage scaling and clock reduction techniques, possibilities of software modifications will have to be considered to further decrease the energy consumption of electronic systems. Improvements in software technology have the advantage of nearly zero production costs and the possibility of late changes in the development process or even later in the field without any redesign or change of the hardware components.

*This work has been supported by Agilent Technologies, USA

During the last five years, research efforts have been spent to develop techniques for software modifications that reduce energy consumption for the whole system including processor, memory and I/O, while maintaining the system's behavior.
Besides the processor itself, off-chip memory accesses come to a high percentage of total energy consumption. The reduction of these memory accesses therefore promises a high potential of energy savings. Several optimization techniques are known which take advantage of this fact. In this paper we present a novel technique which improves the usage of Scratch-Pad memory by assigning variables, program functions or even parts of functions either to Scratch-Pad or main memory. This replaces the off-chip memory accesses by Scratch-Pad memory accesses with lower energy consumption. Unlike Caches, the organization of objects to Scratch-Pad memory is left to the compiler.
In the next chapter, an overview about research work using memory hierarchy for energy models, energy related software generation and compilers using memory hierarchy is given.
In chapter 3, the algorithm for our new technique is presented in detail. The experimental setup described in chapter 4 forms the basis for the results obtained by evaluating the energy reduction using different benchmark applications. Following the conclusions, possible future work is outlined in the last chapter.

2 Related Work

Tiwari et al. [8-11] presented measurements of processor energy consumption based on the executed processor instructions. Their model consists of base costs for a single instruction and the inter-instruction costs which represent the additional energy consumption due to changes in circuit state for consecutive instructions. Tiwari's model does not incorporate the energy consumption of memory accesses.
A different model was proposed by Simunic et al. [7]. For hardware components like processor or memory, the two states 'active' and 'idle' are being distinguished. This allows the simulation and calculation of energy consumption based on data sheets without requiring specific measurements.
Based on Tiwari's model, we developed a power model which includes the memory costs, taking into account data width and direction.
Possible reductions of energy consumption with well known compiler optimizations (linear loop transformations, tiling, unrolling, fusion, fission and scalar expansion) were presented by Kandemir et al. [4]. They consider both the processor core and the memory system together. Their results indicate that, using unoptimized code, more energy is consumed within the memory system than in the core itself.
The aspect of assigning data objects of an application to different memories was described by Sjödin et al. [3]. They presented a memory allocator which assigns global data objects to either on-chip Scratch-Pad or off-chip memory. Simple, static and dynamic profiling techniques are compared with the result that a static profile is generally sufficient for small applications.
Panda et al. [5,6] presented a technique for exploiting on-chip Scratch-Pad memory by partitioning the application's scalar and array variables into off-chip DRAM and on-chip Scratch-Pad SRAM and D-Cache. They proposed a combination of SRAM and D-Cache together with a partitioning algorithm. This results in a 30% performance improvement compared to the sole utilization of SRAM or D-Cache.
In this paper we consider processors without caches like the AT91M40400 processor which includes an ARM7TDMI core and a 4K Scratch-Pad Memory. Taking advantage of von Neumann's architecture model where program and data

objects are stored in the same memory, our memory allocator assigns data as well as program objects to the Scratch-Pad or the main memory. Program objects in the Scratch-Pad memory lead to reduced energy consumption during instruction fetch whereas moved data objects reduce the memory access cost of load/store instructions.

3 Memory Allocator

The memory allocator is called in the last phase of a compiler run. The main steps of its algorithm can be described as follows:

1. Analysis of variable accesses

   All load/store instructions are analyzed and the corresponding variables identified. The number of executions and the variable size are evaluated to calculate the potential energy savings and the required memory size.

2. Analysis of functions and basic blocks

   All functions and basic blocks which can be moved to the Scratch-Pad memory are identified. The number of instructions in each basic block and the number of times each function is called are computed to determine the potential energy savings and the required memory size. Necessary additional jumps between basic blocks have to be taken into consideration at this point.

3. Solution of the knapsack problem

   Using a branch and bound algorithm, the set of objects with the highest sum of total energy savings that will fit in the Scratch-Pad memory is computed.

4. Move of selected memory objects

   The identified variables, functions and basic blocks are moved into the Scratch-Pad memory and for instructions, the necessary jumps are inserted.

The following sections present these steps in a more detailed manner.

3.1 Analysis of Variable Accesses

3.1.1 Identifying variables in memory

The algorithm described here has been integrated into a power aware C compiler developed for research purposes. Whenever possible, during the compiler run, a link is added from each load/store instruction to the corresponding variable which causes the memory access. This can be a scalar or a non-scalar variable.
Load and store instructions in the generated assembly code are identified and the associated variable is stored. If the corresponding variable cannot be identified, those load and store instructions are not taken into account.
Variables which do not cause a memory access because they are only held in registers are not relevant.

3.1.2 Number of executions of load/store instructions

A function call graph is generated and the number of execution times for each function exe_f and each basic block exe_{bb} is calculated. Basic blocks are those parts of a function which are executed sequentially without any interruption of the control flow.
Branching points like `if` statements split the control flow and the execution count is thus shared among the different paths.
For a single load/store instruction the number of executions $exe_{loadstore}$ is identical to that of the basic block exe_{bb} the statement is part of. The number of executions within each basic block is now summed up to the total access count of the corresponding variable.

3.1.3 Determination of size of variables

The memory size of a variable has to be known to calculate energy savings: The cost function for scalar and non-scalar variables is the ratio between the access count and the size of the variable.
Furthermore, the size of the variables $size(var)$ determines the limit for the maximum number of memory objects which can fit into the Scratch-Pad memory.
Since information concerning size is easily available within the compiler but very difficult to extract from assembly code alone, our approach of integrating the memory allocator into the compiler is a sensible approach.

3.1.4 Energy savings by moving variables

Based on the number of memory accesses which can be redirected to the Scratch-Pad memory, the difference in energy consumption can be computed.
Concerning variables, only load/store instructions are relevant. The difference between the energy consumption of the main memory and the Scratch-Pad memory along with potentially saved cycles due to faster memory access has to be multiplied by the number of executions.
The result of this step is a list of memory objects along with their size and an amount of energy which could be saved by moving the corresponding object to Scratch-Pad memory.

3.2 Analysis of functions and basic blocks

3.2.1 Identification of functions and basic blocks

In processors using one single memory for instruction and data, parts of the program could also be moved into the Scratch-Pad memory, just like the variables.
There are two possibilities of moving program objects:
First, a whole function could be moved completely. This is easy to evaluate since the call to the function label is only redirected to a different memory location. Thus no change in the generated assembly code is necessary, meaning the required memory size is easy to determine. It is mainly the linker's task to assign functions to different memories. A function may potentially be moved if its size allows it to fit into the Scratch-Pad memory.
Second, the compiler evaluates basic blocks: A closer analysis shows that very often only parts of a function are executed many times, whereas the remaining basic blocks are only executed once whenever the function is called. This is especially true for the innermost loop of a function which is usually executed an order of magnitude more

frequently than e.g. the beginning of a function. If the whole function cannot be moved to the Scratch-Pad memory, it still seems efficient to move at least the innermost loop.
However, basic blocks in leaf functions including an optimized return instruction cannot be moved. The jump from the main memory to the Scratch-Pad memory is implemented by a long branch instruction which corrupts the return address. Such basic blocks would require more complex code modifications.

3.2.2 Number of executed instructions within functions and basic blocks

The number of executed instructions of functions n_f and basic blocks n_{bb} can be determined based on the function call graph, just like the determination of the number of load/store accesses.
For basic blocks, the number of executions exe_{bb} has to be multiplied with the number of instructions $instr_{bb}$ within this block to get the total number of executed instructions n_{bb}.

$$n_{bb} = exe_{bb} * instr_{bb}$$

For a function, the sum of all executed instructions n_f of its basic blocks has to be computed.

$$n_f = \sum_i n_{bb,i}$$

3.2.3 Determination of the size of functions and basic blocks

The size of functions $size(f)$ can be determined on the basis of the assembly instructions generated by the compiler. External functions cannot be evaluated since their instructions are unknown during the run of the memory allocator within the compiler.
The size of basic blocks $size(bb)$ is more difficult to compute. If e.g. one basic block within a function is moved to the Scratch-Pad memory, the predecessor basic block needs an additional jump into the Scratch-Pad memory. The moved basic block needs a jump back to main memory.
A special case occurs for the first basic block of a function. This basic block is only reached by a call, so no insertion of an additional jump into the Scratch-Pad memory is necessary.
In the worst case a conditional branch can occur as the last statement of a basic block. In this case, if the jump condition is not true, the control flow would usually continue with the first instruction of the consecutive basic block 3 (c.f. figure 1).
Consequently, two jump instructions have to be inserted:
If the jump condition is true, the branch is executed and leads to an additional instruction which implements a jump to the basic block 4 in main memory.
If the jump condition is not true, an additional long branch is inserted to jump to the first statement of the following basic block 3 in main memory.
The exact implementation of this methodology depends on the processor and the jump range. In the THUMB mode of the ARM7TDMI processor, the range of conditional jumps is limited to 128 statements. In most cases this is not sufficient for jumping to a different memory block and an insertion of a long branch is necessary.

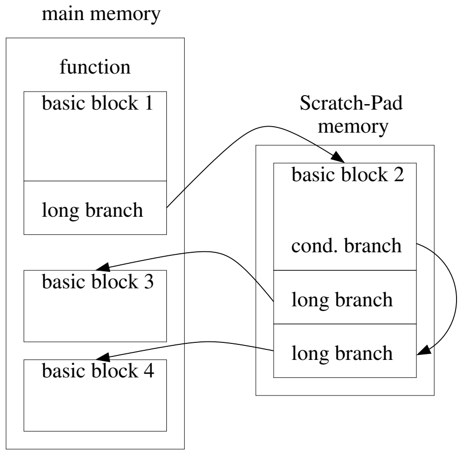

Figure 1: move of a basic block with conditional branch

3.2.4 Energy savings by moving functions or basic blocks

If functions or basic blocks are moved from main to Scratch-Pad memory, instruction fetching will change for the instructions within the moved objects. The difference in energy and number of cycles between main memory and Scratch-Pad memory for the instruction fetch has to be multiplied with the number of executed instructions within the moved objects.
The overhead necessary due to the insertion of jumps can cause a negative value with respect to energy savings for small basic blocks. This can occur if the energy consumption of the additional jumps is higher than the amount that can be saved by moving these instruction into Scratch-Pad memory. In this case, the basic block is not considered during the following steps.
The result of this step is a list of functions and basic blocks with a memory size which is needed in the Scratch-Pad memory and the amount of energy which could be saved by moving each object to Scratch-Pad memory.

3.3 Search best set of memory objects

In this phase, the combination of memory objects which fit into the Scratch-Pad memory and deliver the highest overall energy saving has to be computed.
To solve this problem, we propose the use of the well-known knapsack approach [12]. The Scratch-Pad memory forms the knapsack with a fixed size k. A set of memory objects s_{opt} has to be found which results in a maximum value for a specific cost function E. In the case presented here, energy savings are used as cost function.
The algorithm [12] works as follows:

1. sort all memory objects s by their valence, which is defined as:

$$val(s_i) = \frac{E(s_i)}{size(s_i)}$$

2. put all memory objects s into the set of considered objects $s_{current}$

3. determine critical index c such that all elements in $s_{current}$ up to but excluding s_c fit into the knapsack:

$$k^* = \sum_{i=0}^{c-1} size(s_i) <= k < \sum_{i=0}^{c} size(s_i)$$

4. compute the upper bound U:

$$U = \sum_{i=0}^{c-1} val(s_i) + \frac{(k - k^*)}{size(s_c)} * val(s_c)$$

5. compute the lower bound L in the following steps (n = total number of memory objects):

   $k_t mp$ = 0; L = 0
   for i = 1 to n
      if $k_t mp$ + size(s_i) $< k$ then
         $k_t mp$ = $k_t mp$ + size(s_i)
         L = L + val(s_i)

6. if lower bound L equals upper bound U, the set of objects is valid and added to a set T_{valid} of valid solutions

7. else branch to two subproblems:

   (a) critical element c is inserted into the set $s_{localopt}$ and removed from $s_{current}$. The knapsacksize k is reduced by size(s_c).

   (b) critical element c is removed from the set of considered objects $s_{current}$

   continue with step 3.

8. determine the element s_{opt} of T_{valid} with highest energy savings

The computing time depends on the number of objects n and the number of tree nodes t which is limited by $2^{n+1} - 1$:

$$O(n \log n + t * n)$$

The experimental results presented in the next section with functions and global variables require computing times below 10 ms.
The described problem is a knapsack problem with one dimension. It could be extended to a multi dimensional knapsack for two aspects:

1. If functions and basic blocks are considered at the same time, there is a dependency between a function and its basic blocks. If the function is moved, the basic blocks inside the function are moved automatically and an additional move of such a basic block does not help in finding a valid solution.

2. The size of the basic blocks depends on the necessary additional jumps. If two consecutive basic blocks are moved, the jumps between these basic blocks can be omitted. This means that the memory size of one basic block depends on the algorithm's result for other basic blocks.

Our future work will include implementing the extensions described above which will further improve the obtained results.

3.4 Move variables, functions and basic blocks

In the last phase of the memory allocator the functions, basic blocks and variables are actually moved to the Scratch-Pad memory. This is done by the linker.

4 Experimental Environment

The technique presented in the previous section was implemented and integrated into the encc compiler for the ARM7TDMI processor. This energy aware C compiler is being developed for research purposes. Besides the common optimization strategies for size and time, it also allows an additional optimization for energy.
The values used as input for the power model were obtained by a series of measurements with the evaluation board ATMEL AT91EB01 [2] which incorporates the AT91M40400 processor with a 4 KB Scratch-Pad RAM, but no data or instruction cache (figure 2).
For each processor instruction the compiler has access to a data base entry with the number of instruction cycles, memory size and energy consumption. The different kinds of memory are also modeled with a data base for the energy consumption depending on data width and direction.

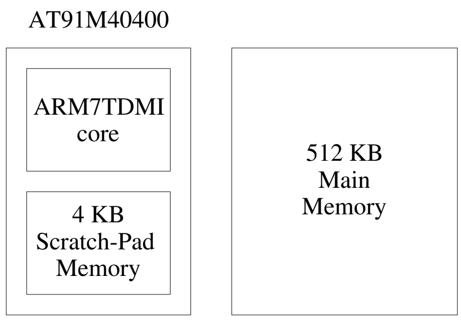

Figure 2: Evaluation Board AT91EB01

These two data bases are used by the code selector to choose optimal instructions and, later in the compiler run, by the memory allocator to find the optimal set of memory objects. The generated assembly code is simulated using the instruction set simulator included in the ARM software development toolkit ARM SDT 2.50 [1]. In our experiments, the trace output is evaluated using a simulation trace analyzer which is part of the environment of the energy aware compiler encc.

5 Results

The effectiveness of the presented technique was evaluated for three different types of benchmarks.
First, the technique was applied to a suite of different sorting algorithms (bubble sort, heap sort, quick sort, selection sort, insertion sort).
Furthermore, two benchmarks often used for digital signal processing were used for evaluation (biquad_N_sections, lattice filter).
Finally, a matrix multiplication (matrix_mult) and a multimedia application (me_ivlin) are part of the benchmark set.

In the following figures the energy consumption for the benchmarks described above is shown for varying Scratch-Pad sizes from 0 Bytes to 2048 Bytes.
To allow a better comparison between the different benchmarks, the values were normalized to 100 percent for the configuration without using Scratch-Pad memory (size = 0) for each individual benchmark.
Figures 3 and 4 demonstrate the results for the assignment of functions and global variables to Scratch-Pad memory in relation to its size. In figure 3, Scratch-Pad memory begins to show a positive effect on energy consumption at a size between 64 and 1024 Bytes. Even with such a very small memory the energy consumption can be reduced by 40 to 65%. This value steadily increases up to the maximum value, where all functions and global variables fit into the Scratch-Pad memory. The energy saving in this case varies between 55% and 80%, depending on the benchmark.

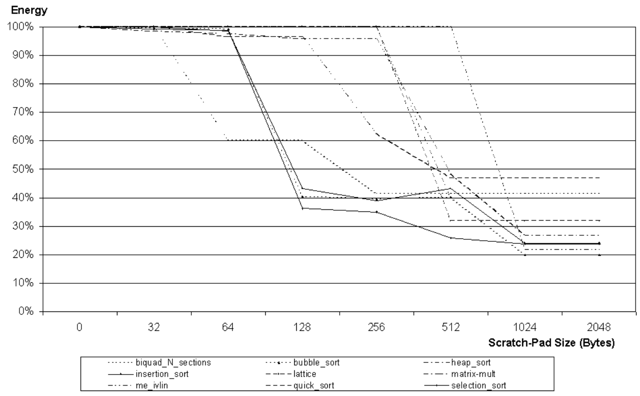

Figure 3: Moving functions and global variables

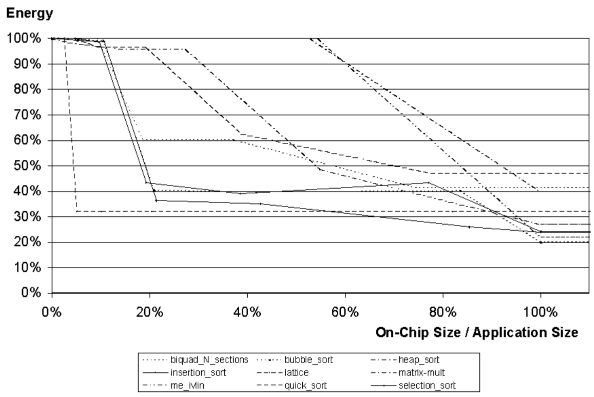

Figure 4: Moving functions and global variables (relative)

Since the size of the Scratch-Pad memory is limited, a part of the program usually remains in the main memory. The ratio between total memory size of the benchmark and Scratch-Pad memory size is shown on the x axis in figure 4.
The lattice filter benchmark can save nearly 70% with less than 10% Scratch-Pad size, whereas other benchmarks like matrix-mult require more than 50% to show first improvements. These results show that the optimal size or optimal ratio between Scratch-Pad size and total application size is application dependent.
The same experiment was performed moving basic blocks instead of functions. These results are shown in figures 5

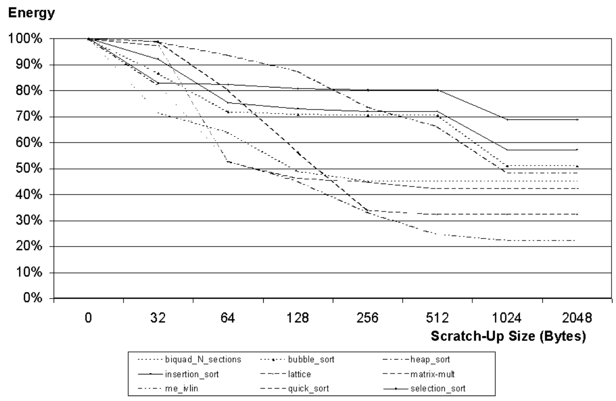

Figure 5: Moving basic blocks and global variables

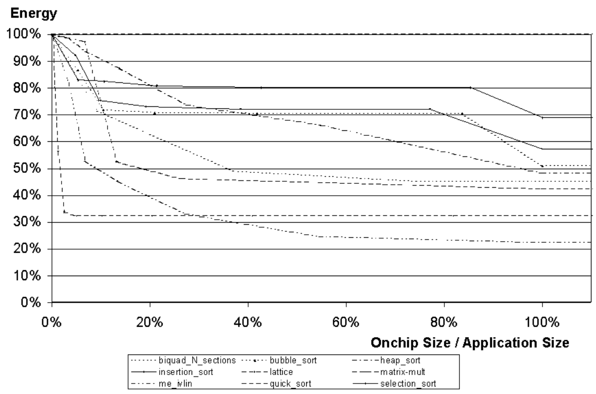

Figure 6: Moving basic blocks and global variables (relative)

and 6. It can be stated that improvements start with smaller Scratch-Pad sizes compared to moving functions. This is due to the smaller size of basic blocks in comparison with functions. If the available Scratch-Pad size increases up to the total application size, the results are worse than those achievable by moving functions. The reason is the overhead caused by the additional jumps which reduce the overall benefit. Also, there are some basic blocks e.g. in the selection_sort benchmark which can not be moved at all.

6 Conclusion

In this paper a technique is presented which utilizes the Scratch-Pad memory for global variables and program functions or basic blocks. Our experiments show energy reductions of up to 78% of system energy consumption.

For small Scratch-Pad memory sizes better results are obtained by moving basic blocks together with global variables. For a high ratio between Scratch-Pad memory to total application memory size it is better to move whole functions and global variables.

The described technique is easily integrated into existing compilers without fundamental modifications of the generated assembly code.

7 Future Work

The extension of the branch and bound algorithm for solving multidimensional knapsack problems will allow a combined treatment of basic blocks and functions together with global variables.
Another object which could be moved to the Scratch-Pad memory is the program stack. In order to do this, an evaluation of the maximum stack size will be necessary.
Furthermore experiments are planned with benchmarks which need more memory size.

8 References

[1] www.arm.com, Advanced RISC Machines Ltd.

[2] AT91M40400 processor, www.atmel.com, ATMEL Corporation

[3] J. Sjödin, B. Fröderberg, T. Lindgren, "Allocation of Global Data Objects in On-Chip RAM", Proc. of the ACM CASES 98 Workshop on Compiler and Architectural Support for Embedded Computer Systems, December, 1998.

[4] M. Kandemir, and N. Vijakrishnan, M.J. Irwin and W. Ye, "Influence of Compiler Optimizations on System Power", Proc. Of the 37th Design Automation Conference, Los Angeles, CA, 2000

[5] P. R. Panda, N. D. Dutt, A. Nicolau, "Efficient Utilization of Scratch-Pad Memory in Embedded Processor Applications", In European Design and Test Conference, Paris, March 1997

[6] P. R. Panda, N. Dutt, A. Nicolau, "Memory Issues in Embedded Systems-On-Chip", Kluwer Academic Publishers, 1999

[7] T. Simunic, L. Benini, G. De Micheli, "Cycle-Accurate Simulation of Energy Consumption in Embedded Systems", Proc. Of the 36th Design Automation Conference, New Orleans, 1999

[8] V. Tiwari, and M. Lee, "Power Analysis of a 32-bit Embedded Microcontroller", VLSI Design Journal, 1998(7)

[9] V. Tiwari, and S. Malik, and A. Wolfe, "Compilation Techniques for Low Energy: An Overview", Proc. of the 1994 IEEE Symposium on Low Power Electronics, San Diego, CA, 1994

[10] V. Tiwari, and S. Malik, and A. Wolfe, "Power Analysis of Embedded Software: A First Step towards Software Power Minimization", IEEE, Trans. On VLSI Systems, December, 1994

[11] V. Tiwari, and S. Malik, and A. Wolfe, "Instruction Level Power Analysis and Optimization of Software", Journal of VLSI Signal Processing Systems, August, 1996

[12] M. Syslo, and N. Deo, and J. Kowalik, "Discrete Optimization Algorithms", Prentice-Hall, New Jersey, 1983