Improving Register File Banking with a Power-Aware Unroller

José L. Ayala, Marisa López-Vallejo
Departamento de Ingeniería Electrónica
Universidad Politécnica de Madrid (Spain)
Email: {jayala,marisa}@die.upm.es

***Abstract*— The complexity of the register file determines the cycle time of high performance wide-issue microprocessors due to the access time and size of this structure. Both parameters are directly related to the number of read and write ports of the register file. Therefore, it is a priority goal to reduce this complexity in order to allow the efficient implementation of complex superscalar machines. The proposed banked register file architecture, also supported with a design-specific compiler, efficiently reduces both the complexity and the energy consumption of this device. In particular, the loop unrolling mechanisms devised modify this compiler optimization to take advantage of the underlying architecture without compromising the system performance.**

I. Introduction

In high performance wide-issue processors the register file plays a critical role in determining the cycle time, directly thorough its size. Furthermore, it accounts for a significant fraction of the processor core's power consumption. These register files need to be large to support multiple in-flight instructions and multi-ported to avoid stalling the instruction issue. In the Alpha 21464, the register file design was several times larger than the 64 KB caches [1] and was split to reduce cycle time impact. Both large size and high number of ports result in slow access and high energy dissipation. This constitutes a very serious problem for the microprocessor industry, unable to integrate such complex devices in the final integrated circuit.

Current trend on superscalar machines is on designing multi-pipelined architectures. Recent improvements on technology integration and computer architecture enable this growing complexity. However, the bottleneck is clearly determined by the access time to the multi-ported register file and the unfeasible complexity of this device. Therefore, important efforts have to be done on reducing the number of register file ports and energy consumption.

The approach presented in this paper proposes a banked register file architecture and a compiler support where the register assignment has been modified to allow the reduction of the number of ports in the register file. Unlike previous approaches, this technique does not rely on additional memories or complex logic that increase the total energy dissipation on the system. Moreover, the proposed approach avoids any penalty on the system performance by relying the complexity of the improvement in the compilation phase. The technique presented for reducing the complexity and number of ports of the register file has also been combined with a low-power policy which shows optimum results on energy savings by means of a simple and reduced extra hardware.

Finally, the effect of the loop unrolling mechanism is extensively analyzed, and different approaches which modify this compiler optimization are proposed in order to take advantage of the underlying architecture without compromising the system performance.

The next section of the paper provides the motivation for this work by presenting some previous works on the same topic. Section III describes our proposed technique while the unrolling mechanism is presented in section IV. The experimental setup as well as the simulation results are shown in section V. Finally, section VI will draw some conclusions.

II. Related Work

The energy complexity of register files has been carefully studied during last years. In [2], Zyuban et al. compare various register file circuitry techniques considering their energy efficiencies as a function of architectural parameters such as the number of registers and the number of ports. The dependence of register file access energy upon technology scaling was also studied by the authors. More recent approaches provide register file models for estimating delay, energy consumption and area on complex implementations of the register file. The work in [3] shows that partitioning the register file following three axes (data-parallel axis, instruction-level parallel axis and memory hierarchy axis) reduces the cost of register storage and communication but with some performance punishment. This work also develops a taxonomy of register architectures by partitioning and by optimizing the hierarchical register organization to operate on streams of data.

Some previous works have proposed hierarchical and banked architectures for the register file but in these cases, unlike our, the only goal is to reduce the size and number of registers in the register file. Some examples are the works found in [4], [5] and [6]. Some compiler approaches have also been presented, like the work in [7] focused on VLIW processors.

These previous approaches oriented to reduce area and complexity of the register file usually require substantial changes to the pipeline and have the potential to create significant

This work was supported by the Spanish Ministry of Science and Technology under contract TIC2003-07036

complications as noted in [8]. In this work Park et al. propose to reduce the number of register ports through two proposals, one for reads and the other for writes. The first approach needs of an extra pipeline stage, while the second one performs bypass prediction. Both techniques can cause pipeline stalls due to mis-predictions, punishing global system performance.

The interest on banked register files arised years ago, when some partitioned architectures were proposed to reduce the complexity of the monolithic register file [9]. Those approaches did not consider the energy implications of the technique and the energy overhead of the extra logic. Late works have improved former approaches using compile-time techniques for clustered processors to exploit the temporal locality of references to remote registers in a cluster [10]. But neither this technique considers the use of low power techniques and the design of power-aware logic to reduce the energy consumption of the register file as well as the number of ports.

An interesting work by Tseng et al. [11] achieves quite good energy savings in a banked register file without compromising the IPC of the machine. However, the number of register file ports is not reduced and even increased since the main goal of these authors is not to reduce the register file complexity.

Further, in [12] Kim et al. introduce techniques for reducing the number of ports in the register file by means of the addition of some auxiliary memory structures. This approach can effectively reduce the number of ports without noticeable impacting the system performance or IPC. However, the energy overhead of the extra hardware is not negligible.

Finally, the influence of compiler optimizations in power consumption has been previously studied at different detail levels [13], [14], supporting these works the main assumptions made in our research.

To the best of our knowledge, no single approach has considered both the banked architecture of the register file as a low-power low-complexity mechanism, and the impact of loop unrolling in the efficiency of this technique.

III. Approach

As was previously said, there are two main goals in our approach. First, to reduce the number of ports of the register file to decrease the complexity and the access time of this module. Second, to reduce the energy consumption of the same device.

These goals will be achieved by means of a modified register file architecture split into smaller banks with reduced number of ports, and a compiler support that modifies the register assignment in order to efficiently use those banks. Morover, a low-power policy turns unused banks into a low-power state to save as much energy as possible. The following sections will describe both architectural and compiler modifications proposed in our approach.

A. Banked Register File

To achieve our goals, the register file is split into several sub-banks with reduced complexity, number of ports and integrated registers. These banks must provide the same amount of registers and the sufficient read and write ports for enabling the normal functioning of the pipeline without performance penalty. For instance, a 22R/11W register file with 256 architectural registers, can be split into four 6R/3 banks with 64 architectural registers per bank. Current architectures integrate similar number or even more registers in the register file.

The reduction of ports and size in the register file simplifies the complexity of the design, reduces the energy consumption per access and permits to follow the current trend on designing multi-superscalar machines with a rapidly increasing number of parallel pipelines.

B. Register Assignment

To implement the improved register assignment policy we have chosen the GNU compiler gcc 3.2 which is publicly available, generates high quality code and supports multiple embedded and high-performance processors.

The compiler performs a first register assignment by translating the logical registers coded in the instruction word into the physical registers available in the hardware. This software assignment is later modified by the register renaming hardware to avoid hazards. The approach presented in the paper assumes a banked register renaming hardware like the one proposed in [15] for saving energy. In this way, the compiler decisions are not destroyed by the renaming mechanism because it restricts renaming to a single register file bank.

As was previously commented, this approach presents a static technique based on a modified compiler together with a dynamic technique based on modification of the register renaming hardware. Both schemes will be explained in the following paragraphs.

Gcc, when assigning an architectural register to the instruction operands, retrieves the first available register from a list (a FIFO) of free registers. The order of the registers inside the list is not representative and depends on the specific hardware architecture. Since gcc does not consider any restriction on assigning the registers, these are selected consequently and, therefore, all the operands in our approach would come from the same register file bank if no modification to the register assignment algorithm is accomplished.

Our register assignment policy modifies the traditional assignment by promoting every operand in the instruction to a different register file bank. With this modification, the number of ports in every register file bank is reduced since less accesses per bank are performed in parallel.

The algorithm followed by the compiler to assign the architectural registers is shown in Figure 1. Initially, the first available register in the list of free registers is selected. This register is double-checked to be free and not system-reserved and, after that, compared to the registers assigned to the other operands of the instruction. If the register file bank for the operand under assignment matches any of the other operands of the instruction, this register is promoted to the next bank and the procedure is repeated. When the register is selected, the liveness of the register is calculated and the annotation is generated. Once the register assignment is completed, every operand of the instruction has been disposed in a different register file bank.

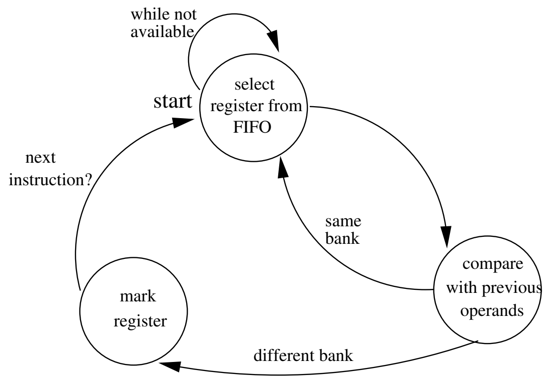

Fig. 1. Register assignment algorithm

The approach presented in this work is developed for the separate implementation of the rename buffers [16]. In our discussion about the implementation of register renaming alternatives we will consider only one aspect: the allocation of a free rename buffer to a destination register. Whenever an instruction referencing a destination register is issued, a new rename register needs to be allocated to the destination register concerned. This allocation is performed by looking for and reserving a free entry, and properly initializing its fields, in a FIFO of free registers.

In our case we split the 256-position FIFO of free registers into four 64-position FIFOs. The size of these FIFOs has to be large enough to provide rename buffers to the destination registers but avoiding to become empty[1]. In this way, the problem of indetermination concerning the register renaming process has been mostly solved. Instead of assigning free rename buffers from the same FIFO to the whole set of logical registers, the first set of 16-logical registers is assigned to the first FIFO, the second set to the second FIFO, the third to the third one and the last set of 16-logical registers is assigned to the last 64-position FIFO.

Summarizing, the banked register file architecture is supported for a banked FIFO of free registers which assures that the renamed registers belong to a specific register file bank, and for a modified compiler which spreads the intruction operands among the different banks of the FIFO. Therefore, the assignment of physical registers is partially controlled from the compilation stage.

C. Low-Power Policy

The implementation of the register file in several banks can also be used to reduce the energy consumption in this device. The low-power policy turns into a low-power state those banks not required by the current instruction.

This precise low-power policy applies a voltage scaling technique to reduce the power consumption of these unused banks to a minimum. A dual voltage source is needed as well as the control logic to detect which banks should be turned off and when to perform this reconfiguration. The memory cells are turned into a low-power state which requires to reinstate the cell contents one clock cycle before the actual access [17].

[1]The correctness of the 64-position choice has been validated by simulation.

The control logic only needs to attend to the operand labels coded in the instruction to detect the unused banks (a detailed explanation of this logic and such approach can be read in [15]). The detection of the banks required by the instruction has to be done with enough time in advance for turning them on before the access happens. Given that a maximum of three register file banks are accessed per instruction, the unused banks are turned into the low-power state reducing significatively the energy consumption.

During the fetch phase of the out-of-order pipeline, the instructions are read from the instruction memory and placed into one of the fetch buffers. If no fetch buffer is available, then fetch stalls. During the dispatch phase, the instructions are taken from one of the fetch buffers and placed at the tail of the instruction queue. When an instruction is enqueued, it may or may not have all of its operands, and it will wait in the instruction queue for the availability of the operands. Once all the operands of an instruction are valid, its operands, opcode, and ID are sent to the input registers of the appropriate functional unit. Therefore, there is time enough for turning the registers on before the access happens without any performance penalty.

IV. Design of the Loop-Unroller

Previous research and simulations conducted on this topic have shown how the presented approach can fail when the number of required registers cannot be satisfied by the register file bank. In those cases, the pipeline has to stall, reducing the system performance. This effect is particularly notorious inside loops when the compiler unrolls them.

Loop unrolling intends to increase instruction level parallelism of loop bodies by unrolling the loop body multiple times in order to schedule several loop iterations together. The transformation also reduces the number of times loop control statements are executed. As loop unrolling reduces execution time through effective exploitation of ILP from different iterations, it has been presented in the past as an effective compiler mechanism to reduce the energy consumption.

However, loop unrollers perform better for in-order architectures. Current widely-available compilers are not able to exploit the dynamic scheduling facilities found in out-of-order processors, and the ILP improvements are not so spectacular. On the other hand, the unroll of outer loops (or the unroll of inner loops by large factors) exploits the register requirements and increases the energy consumption on the register file. Recent research in modern architectures has shown how loop unrolling proved to have little effect in terms of program execution time [14]. Moreover, these works did not consider the increment on energy consumption due to the increased register usage when the unrolling takes place.

This section presents a power-aware unroller mechanism to efficiently reduce the energy consumption in the register file of out-of-order processors. The modified unroller considers the following alternatives:

- Selection of an unroll factor which fits the register requirements into the available register file bank.
- Use of an unroll bank of registers to perform unrolling in a safe, energy-controlled space.

- Deactivation of the loop unrolling optimization.

All these different possibilities of implementing the unroller will be explained next.

A. Selection of the Unroll Factor

Once register assignment is performed by the compiler and before any loop unroll has taken place, the number of required registers inside the ness is perfectly known. The loop unroller exploits the register requirements by placing several copies of the same code and increasing in this way the register demands. Subsequent compiler optimizations (for example, software pipelining) will reduce the number of demanded registers by promoting unused registers or reusing operands.

Therefore, though the exact number of required registers cannot be known in advance, this number can be estimated. Once the estimated number of required registers is known, the unroller can select the unroll factor which fits the register requirements into the available register file bank while the others remain off.

B. Use of the Unroll Bank of Registers

The previous phase can result in the selection of an unroll factor too small if the loop requires a great amount of registers. This reduced unroll factor could determine a penalty in the system performance because other optimizations such as common-subexpresion elimination, induction-variable optimizations, instruction scheduling or software pipelining loose effectiveness. For that reason, an *unroll bank* of registers will be considered.

This unroll bank of registers consists of a bank with reserved registers, whose size is bigger than any of the register file banks, and that remains off during normal execution. When needed, the unroll bank has to be explicitly turned on by the compiler, switching off the rest of register file banks (i.e. the working register file bank is reconfigured to a bigger one to be used during the loop).

In order to perform this operation, the registers currently used as inputs inside the loop have to be moved to the unroll bank of registers (the contents have to be copied). Also, when the loop exits, the output registers have to be moved to the register file bank they belong to. This requires some extra clock cycles to perform the operation, which negatively impact the system performance. Therefore, this compiler optimization will only be allowed in long and frequently executed loops with strong register requirements (i.e. those loops which represent higher energy savings and whose energy-execution trade-off is justified).

C. Deactivation of the Loop Unrolling Optimization

Previous compilation phases can return an error result if the estimated unroll factor remains below a threshold, or the required registers inside a target loop cannot be fed by the unroll bank of registers. In those cases, provided that the main goal is the power reduction, the loop unrolling optimization will be deactivated for the considering loop. When the estimated unrolling factor is below the previously set threshold, the optimization will be deactivated. Therefore, the banked approach previously presented can perform without modification and the energy savings will correspond to those unused register file banks. The complete picture of the unrolling process is shown in Figure 2.

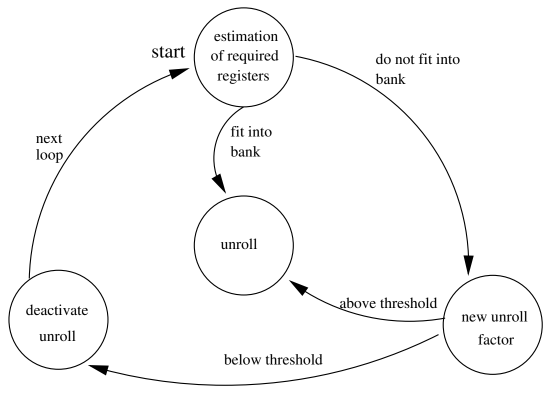

Fig. 2. Loop unrolling mechanism

V. Implementation and Experimental Results

The architectural modifications proposed in the previous sections have been simulated with a simulator derived from the SimpleScalar 4.0 tool suite [18]. SimpleScalar is an execution-driven simulator that implements a derivative of the MIPS-IV instruction set. The architectural details of this approach are described in [15] and [19]. The proposed loop unrolling modifications have been performed modifying the implementation of the GNU compiler gcc 3.2, which is publicly available, generates high quality code and supports multiple embedded and high-performance processors.

The simulated baseline configuration uses a multi-ported register file implementation with 256 registers and 6R/3W ports, e.g. it is configured for a 3-issue machine. The central implementation of the register file will be splited into 8 banks with 32 registers per bank.

Next sections will describe the experimental results collected for every proposed compiler modification.

A. Selection of the Unrolling Loop

As was previously explained, the banked architecture designed to reduce the complexity and energy consumption of the register file, fails on providing the required registers when the loop unrolling mechanism exploits the register demands. These failed cases are related to the specific register demands of every benchmark as well as to the loop's possibilities to be unrolled.

Figure 3 shows the percentage of failed loops (those that exploit the register demands and cannot be fed by the register file banks) for some benchmarks selected from the MiBench suite. As can be seen, the behavior of every bechmark regarding the percentage of failed loops is quite different, ranging from the 2.2% of the *susan* benchmark, to the 24.6% of *bitcount*.

The selection of the loop unrolling factor mechanism has been implemented in two different ways in this first stage of research. Firstly, as the loop unrolling factor can be hardly

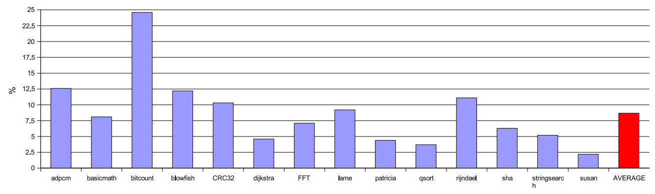

Fig. 3. Percentage of failed loops

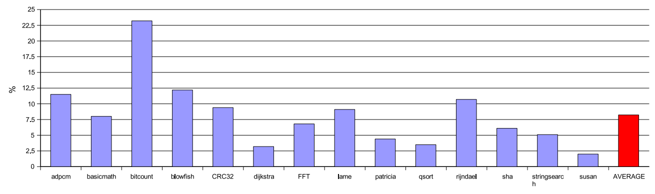

Fig. 4. Percentage of failed loops after factor selection (worst case estimation)

estimated before the full compilation process has finished, the worst case estimation has been selected. The maximun unrolling factor used by gcc can actually be known from the compiler internals before unrolling the loop. Using this estimation, the unroller factor is selected in order to keep the register demands fed by the register file bank and, at the same time, select always an unroller factor above 2 (to allow further compilation stages to increase the performance of the executed code).

Figure 4 shows the percentage of failed loops after the unroller factor has been selected based on the worst case estimation. As can be seen, the percentage of failed loops has been reduced for every benchmark, moving the average from 8.69% to 8.23%. The remaining failed loops respond to the wrong estimation made by the worst case estimation case, and the impossibility to find a loop unroller factor above the threshold.

Secondly, the loop unroller factor is estimated after a post-compilation stage. In this approach, the compiler is allowed to finish the compilation process and, after recovering the selected unroll factor, recompiled with the new selection. Figure 5 shows the percentage of failed loops after the untoller factor has been selected on the post-compilation estimation. As can be seen, those failed loops due to the wrong estimation of the unroller loop are corrected, and the average count of failed loops is decreased to 5.96%. However, the unroller factor cannot always be effectively tuned and some failed loops still remain.

B. Use of the Unroll Bank of Registers

For this set of experiments, the architecture is extended with an extra bank of 64 registers. This *unroll bank* of registers will be used when the unroller factor cannot be conveniently selected. The post-compilation mechanism will be used to estimate the factor selected by the compiler.

The use of an extra bank of registers requires to copy the register contents from the register file to this bank. In order to minimize the performance penalty incurred by this mechanism, just the most time-consumming loop inside the code will

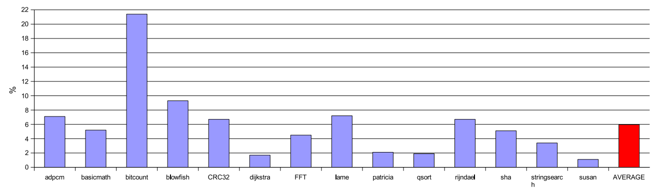

Fig. 5. Percentage of failed loops after factor selection (post-compilation estimation)

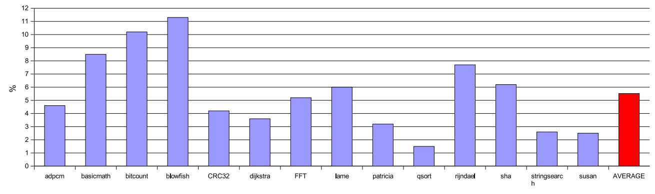

Fig. 6. Percentage of failed loops after the use of the unroll bank of registers

be selected. Previous profiling of the code has revealed this information.

Figure 6 shows the percentage of failed loops after compilation using the unroll bank. As can be seen, some of the benchmarks (those with a loop with represent a significant portion of the execution time) present in this approach a better behavior, while others (those where a significant loop cannot be found) have increased the percentage of failed loops. In average, the percentage of failed loops have been decreased to a 5.52%.

For this approach, the performance penalty incurred by the extra clock cycles needed to use the unroll bank of registers has also been measured (Figure 7). In average, this does not represent more than a 3%.

C. Deactivation of the Loop Unrolling Optimization

Some of the benchmarks' loops cannot be fit in the register file banks by tuning the loop unrolling factor and meeting at the same time the defined threshold. For those loops, the loop unrolling optimization is deactivated. The size of the register file banks (32 register) has been selected in the way that they can feed the register demands for every benchmarks when no un rolling is applied. Therefore, the percentage of failed loops after the deactivation of the unrolling is very reduced (Figure 8).

Although the deactivation of this compiler optimization could seem that strongly compromises the system performance, recent research for modern architectures has shown how loop unrolling proved to have little effect in terms of program execution time [14].

VI. Conclusions

Partitioned register organizations of the register file dramatically reduce delay, area, and power dissipation in comparison to the traditional central implementations. Moreover, banked implementations of the register file scale better when designing high-complexity high-performance superscalar architectures.

Common compiler implementations usually increase the quality of the code as well as the execution performance.

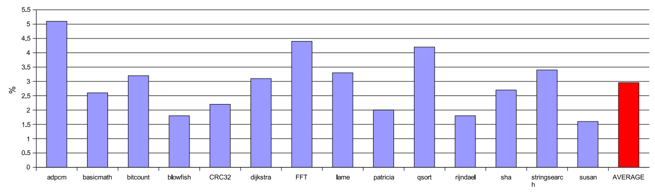

Fig. 7. Performance penalty after the use of the unroll bank of registers

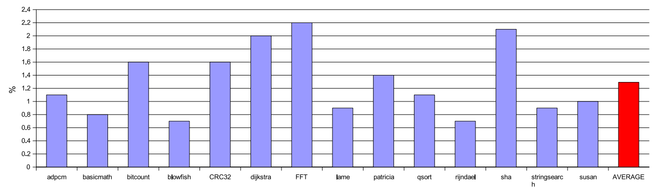

Fig. 8. Percentage of failed loops after deactivation of the loop unrolling

However, they can turn inefficient when the underlying architecture has been modified attending to area, delay or energy considerations. In this way, when the register file is partitioned, the traditional unrolling mechanism can exploit the register demands of internal loops, stalling the pipeline execution if the register file bank is not able to feed them.

In this paper, the proposed banked architecture for the register file is supported by a design-specific compiler where both the register assignment task and the loop unrolling mechanism have been modified to take advantage of the complexity-aware power-aware register file with a low performance impact.

REFERENCES

[1] R. Preston, "Design of an 8-wide superscalar RISC microprocessor with simultaneous multithreading,," in *ISSCC Digest of Technical Papers*, 2002.

[2] V. Zyuban and P. Kogge, "The energy complexity of register files," University of Notre Dame, Tech. Rep., 1997.

[3] S. Rixner, W. J. Dally, B. Khailany, P. Mattson, U. J. Kapasi, and J. D. Owens, "Register organization for media processing," in *International Symposium on High-Performance Computer Architecture*, 2000.

[4] R. Balasubramonian, S. Dwarkadas, and D. H. Albonesi, "Reducing the complexity of the register file in dynamic superscalar processors," in *International Symposium on Microarchitecture*, 2001.

[5] E. Borch, S. Manne, J. Emer, and E. Tune, "Loose loops sink chips," in *International Symposium on High-Performance Computer Architecture*, 2002.

[6] J. L. Cruz, A. González, M. Valero, and N. P. Topham, "Multiple-banked register file architecture," in *International Symposium on Computer Architecture*, 2000.

[7] J. Zalamea, J. Llosa, E. Ayguadé, and M. Valero, "Two-level hierarchical register file organization for VLIW processors," in *International Symposium on Microarchitecture*, 2000.

[8] I. Park, M. D. Powell, and T. N. Vijaykumar, "Reducing register ports for higher speed and lower energy," in *International symposium on Microarchitecture*, 2002.

[9] J. Janssen and H. Corporaal, "Partitioned register file for TTAs," in *International Symposium on Microarchitecture*, 1995.

[10] K. Kailas, M. Franklin, and K. Ebcioglu, "A partitioned register file architecture and compilation scheme for clustered ILP processors," in *International Euro-Par Conference*, 2002.

[11] J. H. Tseng and K. Asanović, "Banked multiported register files for high-frequency superscalar microprocessors," in *International Symposium on Computer Architecture*, 2003.

[12] N. S. Kim and T. Mudge, "Reducing register ports using delayed write-back queues and operand pre-fetch," in *International Conference on Supercomputing*, 2003.

[13] H. Yang, G. R. Gao, A. Marquez, G. Cai, and Z. Hu, "Power and energy impact by loop transformations," in *Innovative Architectures for Future-Generation High-Performance Processors and Systems*, 2003.

[14] J. S. Seng and D. M. Tullsen, "The effect of compiler optimizations on Pentium 4 power consumption," in *Workshop on Interaction between Compilers and Computer Architectures*, 2003.

[15] J. L. Ayala, M. López-Vallejo, and A. Veidenbaum., "Energy-efficient register renaming in high-performance processors," in *Workshop on Application Specific Processors*, 2003.

[16] D. Sima, T. Fountain, and P. Kacsuk, *Advanced computer architectures: a design space approach*. Addison-Wesley Longman, 1997.

[17] K. Flautner, N. S. Kim, S. Martin, D. Blaauw, and T. Mudge, "Drowsy Caches: Simple Techniques for Reducing Leakage Power," in *International Symposium on Computer Architecture*, 2002.

[18] T. Austin, E. Larson, and D. Ernst, "SimpleScalar: An Infrastructure for Computer System Modeling," *Computer*, vol. 35, no. 2, pp. 59–67, February 2002.

[19] J. L. Ayala, M. López-Vallejo, and A. Veidenbaum., "A compiler-assisted banked register file architecture," in *Workshop on Application Specific Processors*, 2004.