Optimal Loop-Unrolling Mechanisms and Architectural Extensions for an Energy-Efficient Design of Shared Register Files in MPSoCs

José L. Ayala†, David Atienza‡, Marisa López-Vallejo†, J. M. Mendías‡, R. Hermida‡, C. A. López-Barrio†
†Departamento de Ingeniería Electrónica
Universidad Politécnica de Madrid (Spain)
Email: {jayala,marisa,barrio}@die.upm.es
‡Departamento de Arquitectura de Computadores y Automática
Universidad Complutense de Madrid (Spain)
Email: {datienza,mendias,rhermida}@dacya.ucm.es

Abstract

In this paper we introduce a new hardware/software approach to reduce the energy of the shared register file in upcoming embedded architectures with several VLIW processors. This work includes a set of architectural extensions and special loop unrolling techniques for the compilers of MPSoC platforms. This complete hardware/software support enables reducing the energy consumed in the register file of MPSoC architectures up to a 60% without introducing performance penalties.

1 Context and Motivation

Business analysts forecast a 200 billion dollar market for system-on-chip (SoC) media-rich, mobile wireless terminals in the near future [19]. On the other hand, the demanding multimedia applications ask for the development of multi-processor systems with high performance capabilities. These forthcoming MultiProcessor System-on-Chip (MPSoC) platforms will include several heterogeneous processors as one of the most effective way to tackle at the same time all the different multimedia services present in such systems, even some initial platforms start to be available today (e.g. ST Nomadik [17], Philips Nexperia [14], TI OMAP [18]).

Unfortunately, the semiconductor industry is still facing several technological challenges to build these systems. They require an enormous computational performance (2 - 30*G*OPS) with low energy consumption demands (0.3-2*W*) [20]. Although current desktop processors offer these performance requirements, they consume too much power (10-100*W*) [9]. Therefore, while keeping the performance figures, the power consumption needs to be at least two or three orders of magnitude lower. Within these context, methods to reduce the power consumption of the new MPSoC platforms are in great need.

In new proposed embedded MPSoC platforms with several processing elements [7, 18], the shared register file heavily affects the cycle time and consumes a very significant portion of the total energy consumed in the whole system. Moreover, it has become one of the critical processor hotspots. The main reasons are that it is large and multiported to support concurrent access of the multiple present processors. These characteristics lead to a large increase in the power dissipation (and indirectly temperature as well) of the whole system [1]. Hence, it is crucial to reduce the energy spent on it.

A large body of research has been devoted to decrease the energy of multiported register files in high performance processors. From the hardware point of view, several authors have studied the complexity of shared register files and proposed distributed schemes [21] and techniques to split the global microarchitecture into distributed clusters with subsets of the register file and functional units [16,22]. Similarly, in the intent to reduce the complexity of the register files, the benefits of multilevel register file organizations [8] have been studied. Conversely, other techniques retain the idea of a centralized architecture, but the register file is split into interleaved banks, which reduces the total number of ports in each bank [13]. In a more general context, additional work has been performed to propose efficient Voltage Scaling techniques according to the application's behavior to reduce power consumption of the system [12].

From the software point of view, several software pipelining strategies to distribute the use of the register file targeted at reducing memory pressure in VLIW systems have been outlined [2, 3]. Also, compiler techniques, including complex register renaming, have been proposed re-

Proceedings of the Innovative Architecture for Future Generation High-Performance Processors and Systems (IWIA'05)
1527-1366/05 $20.00 © 2005 IEEE

cently for in-order processors to reduce the energy spent in the register file [5,6].

In this paper we introduce a new hardware/software approach to reduce the energy of the shared register file in upcoming embedded architectures with several VLIW processors. This work extends our previous work for high-performance monoprocessor systems [4] by including a set of architectural extensions (Section 2) and special loop unrolling techniques (Section 3) for the compilers of MPSoC platforms. This complete hardware/software support enables reducing the energy consumed in the register file of MPSoC architectures without introducing performance penalties. The experimental validation of the proposed techniques has been performed extending the architecture proposed in CRISP [7], and taking advantage of its compilation and simulation capabilities.

2 Proposed Architectural Extensions

The baseline architecture described by CRISP [7] consists of a selectable number of processing elements (VLIW processors), which communicate with a shared register file through a full crossbar network. This architecture has been extended and modified in the following way to support the unrolling mechanisms proposed in this paper (Figure 1):

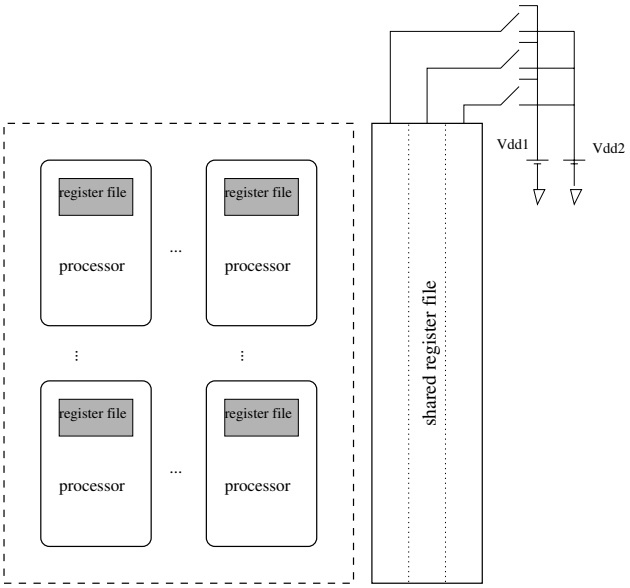

Figure 1. Proposed architecture

- The register file shared among all processing elements has been split into several banks, which can be independently accessed by the processors. A Dynamic Voltage Scaling (DVS) technique is used to turn the unused banks into a low power state and thus save as much energy as possible in the system. The DVS technique scales down the voltage power supply of the unused banks, reducing the power consumption to a minimum while keeping intact their contents thanks to the use of high threshold-voltage transistors [10].

- Every processing element includes now an additional local register file with a reduced size compared to the shared register. All these local register files are switched off during normal functioning and become active on demand.

- Hardware support is provided to the compiler to power up banks of registers in the shared register file or the local register files in several processing elements when they are needed. In normal execution of the system, most of the banks of the shared register file are kept in a low-power state thanks to our modified register assignment implemented in the compiler. When needed, the register file banks or local register files will be powered up to feed the register demands of the code. The selection between both configurations (i.e. extra banks in the shared register file, or the use of local register files) is based on energy considerations analyzed by the compiler using our proposed unrolling mechanisms for MPSoC systems (Section 3).

3 Implementation of Unrolling Mechanisms

3.1 Overview of the Unrolling Mechanisms

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. In addition, as loop unrolling reduces execution time through effective exploitation of Instruction-Level Parallelism (ILP) from different iterations, it has been proposed to use it as an effective compiler mechanism to reduce energy consumption in the instruction memory hierarchy.

However, compilers perform better unrolling for in-order architectures and were originally designed for monoprocessor systems. Current widely-available compilers are not able to exploit the dynamic scheduling facilities provided by out-of-order processors because of the complex control flow present in such systems. This makes very difficult to decide which instructions will be executed next; thus, ILP improvements are not so significant. On the other hand, the unroll of outer loops (or the unroll of many inner loops iterations) exploits the register requirements by using more ports and, subsequently, increasing the energy consumption of the register file. Also, recent research in modern architectures has shown how loop unrolling proved to have little

effect in terms of program execution time [15]. Moreover, these pieces of work do not consider the increment on energy consumption due to the increased register usage when the unrolling takes place, which is mandatory to be taken into account for MPSoC systems.

The practical implementation of the traditional unrolling mechanism in current compilers works as follows. In a first phase, an initial estimation of the required registers is performed for those loops in which the unrolling mechanism is going to be applied, namely where ILP can be clearly analyzed and theoretically exploited by the compiler. The compiler also estimates the maximum number of times that the loop can be unrolled while still meeting the system performance (this number is known in the literature as the *unroll factor*). Then, in the second phase or unrolling phase, the compiler takes into account the data dependencies among registers and, as a results, the effective unroll factor is usually adjusted to a smaller value than the initially estimated one. Finally, the demands of the register file increase during the unrolling phase, and the capacity of the register file is traditionally selected to be big enough to feed the worst-case number of registers for the ILP selected during the compilation process. This leads to oversized devices with high energy consumption and complexity in terms of number of ports.

As a matter of fact, the previous effect of increasing energy consumption is even more dramatic in MPSoCs with a shared register file, where the device has to provide the operands to every processor in the system. Therefore, mechanisms to keep the register demands inside unrolled loops under control are needed, as well as to reduce the energy consumption and size of the register file.

In this section, we propose a mechanism to unroll loops reducing the energy consumption in the register file of out-of-order processors, which is where a larger gap exists between ILP degree in present applications and effective register file management from an energy point of view. Our proposed unrolling mechanism considers the three following alternatives:

- Selection of an unroll factor which fits the register requirements of every processor into the same register file bank.
- Use of additional and private unroll bank of registers per processor (i.e. local register file) to perform unrolling in a safe, and energy-controlled space.
- Deactivation of the loop unrolling optimization in case not effective trade-offs can be exploited between ILP degree and energy consumption in the register file.

In the following subsections we describe in detail each of the previous techniques of our unrolling mechanism.

3.2 Selection of the Unroll Factor

As we have previously mentioned, the number of times that a loop is unrolled by the compiler is known as *unroll factor*. This value depends on the analysis performed by the compiler, which estimates the maximum factor that can be used before performing the unroll, but it changes depending on the data dependencies and execution constraints.

Once register assignment is performed by the compiler and before any loop unroll has taken place, the number of required registers inside the loop is perfectly known. Then, the loop unrolling mechanism exploits the register requirements by placing several copies of the same code and increasing the register demands in this way. 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 with the information provided by the compiler.

After this estimation of the number of required registers has been performed, the unrolling mechanism can select the unroll factor that fits the register requirements into the available register file bank while the others remain off. To this end, assumptions like the percentage of register usage by every processor, or the effective unroll factor selected by the compiler must be made to implement such functionality. For the former, a homogeneous distribution of registers is assumed, while for the latter a worst case scenario and a post-compilation approach is analyzed in Section 4.

3.3 Use of the Unroll Bank of Registers

The previous unrolling phase can result in the selection of an unroll factor that is too small if the loop requires a large number 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 [11] lose effectiveness. For that reason, an *unroll bank* of registers is also considered in our approach.

This unroll bank of registers consists of a local bank of reserved registers for each processor. These banks are bigger than any of the register file banks, but remain off during normal execution. Hence, they do not consume any power. When it is needed, the unroll bank of a certain processor is explicitly turned on by the compiler.

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, namely the contents have to be copied. Also, when the loop exits, the output registers have to be moved back to the register file bank they belong to. This

process requires some extra clock cycles to perform the operation, which negatively impact the system performance. Therefore, this compiler optimization is only allowed in long and frequently executed loops with strong register requirements, such as those loops which represent higher energy savings and whose energy-execution trade-off is justified.

As a result, this technique is only employed when the data independence among the code run by the processors has been detected. In case of data dependencies or uncertainty of such behavior, the shared register file is configured as a smaller device.

3.4 Deactivation of the Loop Unrolling Optimization

Previous compilation phases can create more penalty in execution time and/or energy consumption than benefits if the estimated unroll factor remains below a certain threshold, or if the required registers inside a target loop cannot be fed by the unroll bank of registers. In these cases, provided that the main goal is power reduction, the loop unrolling optimization is deactivated for the loop under consideration. Therefore, the banked approach previously presented can perform without modification and the energy savings correspond to those achieved by switching off the unused register file banks. As a result, the complete flow of the unrolling process including the three previously explained mechanisms is shown in Figure 2. Note that they have been easily integrated in a finite state machine controller, which includes a clear flow between the three mechanisms. Moreover, it includes simple local conditions to be checked in each stage to perform the transitions between the different states that represent the proposed mechanisms, namely exploiting the information generated by the compiler during its operation onto the source code.

4 Experimental Setup and Results

The proposed unrolling mechanisms have been evaluated using the modified CRISP platform previously described in Section 2. The summary of the simulated architecture's characteristics is shown in Table 1.

Table 1. Simulated architecture

Number of Functional Units	4
Size of the Shared Register File	128
Size of the Local Register File	32
Size of the Unroll Bank	64

There is a reduced area overhead due to the use of the extra bank of registers (i.e. local unroll bank). However,

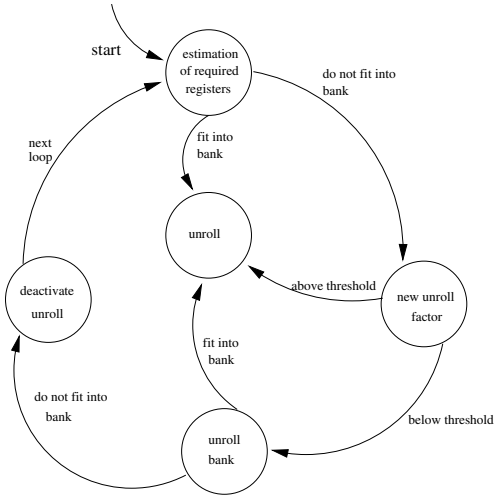

Figure 2. Loop unrolling mechanism

the energy savings that are obtained using this technique, compensate for the impact on area.

The simulations have analyzed the effect of the proposed HW/SW approach for supporting suitable unrolling techniques on several multimedia applications adapted to MPSoCs. Our case studies are the followings ones:

- BTPC: a general-purpose image coding scheme suitable for compression of all kinds of images. BTPC is designed to perform both lossless and lossy compression, and to be effective for both photos and graphics. It is also suitable for compressing multimedia images, which integrate two or more types of visual material.
- QSDPCM: an inter-frame compression technique for video images. It involves a hierarchical motion estimation step, and a quadtree based encoding of the motion compensated frame-to-frame difference signal.
- ADPCM: a waveform codec which, instead of quantizing the speech signal directly, like PCM codecs, quantizes the difference between the speech signal and a prediction that has been made of the speech signal.
- CJPEG: tool which compresses the named image file, or the standard input if no file is named, and produces a JPEG/JFIF file on the standard output.
- EPIC: an experimental lossy image compression utility designed for extremely fast decoding on conventional hardware, at the expense of slower encoding and a slight degradation in compression quality. The compression algorithm is based on a critically-sampled non-orthogonal (imperfect-reconstruction) dyadic wavelet decomposition and a combined run-length/Huffman entropy coder.

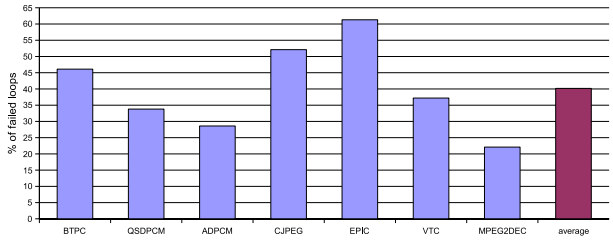

Figure 3. Percentage of failed loops (baseline case)

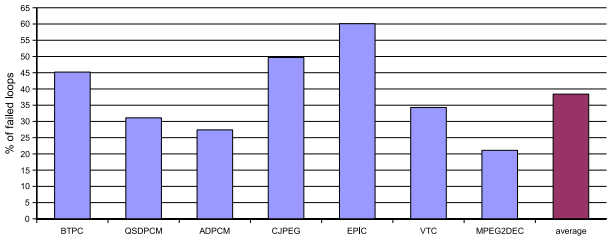

Figure 4. Percentage of failed loops (worst case estimation)

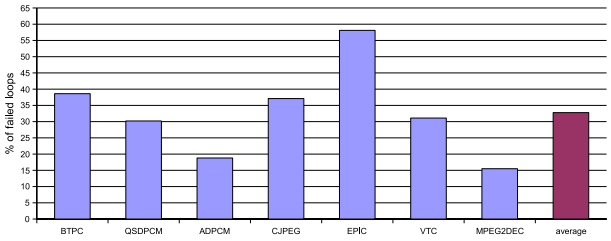

Figure 5. Percentage of failed loops (post compilation)

- VTC: is the algorithm used in MPEG-4 to compress visual textures and still images. It is based on the discrete wavelet transform, scalar quantization, zero-tree coding and arithmetic coding.
- MPEG2DEC: a test program for libmpeg2. It decodes mpeg-1 and mpeg-2 video streams, and also includes a demultiplexer for mpeg-1 and mpeg-2 program streams.

The simulations evaluate the percentage of *failed loops* in several applications and loop unrolling mechanisms. In our notation, the term failed loops refers to loops that cannot be mapped in just one bank of the register file. In that case (every loop can be mapped in just one bank) the gains in power consumption achieve up to 75% compared to the regular case. According to this, a reduction in the number of failed loops enables us to turn a bigger region of the register file into the low-power state during the execution of the benchmark. Therefore, the higher percentage of failed loops, the lower the energy savings that can be obtained. As a consequence, 0% of failed loops means that the energy saving reaches the 75%, while a 100% of failed loops means that no energy savings are achieved.

In the first set of experiments, used for comparison purposes, we use the baseline architecture. It consists of a banked implementation of the register file, with unrolling mechanism enabled and an unroll factor selected by default by the compiler. The percentage of failed loops for this architecture is depicted in Figure 3. These failed loops cannot be mapped into one register file bank due to the register demands, and thus energy waste occurs. The percentage of failed loops reaches 40%; Hence, the energy savings obtained by the banked architecture comes to 45% of the total dissipation in the register file.

In the second set of experiments, we have defined the suitable unroll factor selected by the compiler assuming a worst-case scenario. Hence, the maximum unroll factor initially reported by the compiler is assumed to be the effective factor. Figure 4 shows the results of these simulations regarding the number of failed loops after applying the modification of the unroll factor. As it can be seen, there is not a big improvement with such technique because the estimation performed by the compiler differs from the real execution of the loop unrolling. In fact, the percentage of failed loops is around 38%, which means that the energy savings is just 46.5%, only slightly better than the baseline architecture.

In the third set of simulations we have explored the effect of modifying the unroll factor after a preliminary compilation phase. In this case, the unroll factor is modified after retrieving the effective one used by the compiler. After that, a second compilation phase is launched. As Figure 5 shows, this technique achieves better results with just a negligible overhead due to the second compilation phase. In this case, the 33% of failed loops achieved with such technique enables 50.25% of overall energy savings.

In addition, we have run a fourth set of experiments, where the most time-consuming loop without data dependence among processors is selected to be mapped into the local unroll bank of registers. Figure 6 shows the percentage of failed loops when the unroll bank of registers is employed. In our case studies, up to 56.25% of energy savings are accomplished. Nevertheless, they also indicate that the possible energy savings that can be obtained with this technique depend very much on the possibility to find a data-independent loop, and its weight in the overall execution. Therefore, future work in this matter is needed to develop appropriate strategies to use this mechanism for each type

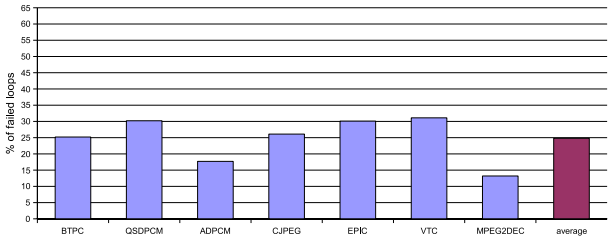

Figure 6. Percentage of failed loops (unroll bank)

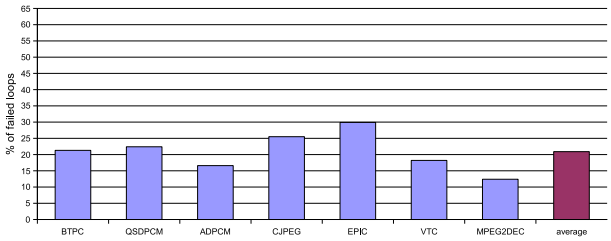

Figure 7. Percentage of failed loops (deactivation)

of considered application.

Finally, we have evaluated in a final set of experiments the deactivation of the loop unrolling mechanism for all those loops that still cannot be mapped into neither the local register file nor the unroll bank (see Figure 7). The energy savings obtained with this approach are obviously higher (60%) because a larger number of loops can be mapped into just one bank of the register file (20% of failed loops). However, the extensive use of this approach could negatively impact MPSoC's performance due to the underuse of resources. Future research work in this field is required to estimate the impact of this technique and how to minimize its implications on performance.

5 Conclusions

MPSoCs represent a new challenge in system and power-aware design. Currently, compiler technology is not mature enough to support the architectural extensions and capabilities of these devices. The work presented in this paper has analyzed the effect of different loop unrolling mechanisms in a proposed banked architecture of the register file for MPSoC systems conceived with low power constraints.

Our results have shown how the careful selection of the unroll factor, the efficient use of an extra bank of registers, and the deactivation of the loop unrolling performed by the compiler in certain extreme cases, can decrease the percentage of failed loops (i.e. in our notation, the loops that cannot be mapped in just one register file bank). As a result of decreasing this percentage, the energy consumption of the device is equally reduced since the rest of the banks in the register file of the MPSoC architecture can be set to a low power state without significant performance penalties.

Several future research lines have also been drawn. Our current work is focused in the development of effective register allocation algorithms to improve the energy behavior of the register file in MPSoCs, which include policies to dynamically (de)activate the use of unrolling mechanisms.

Acknowledgements

This work is partially supported by the Spanish Government Research Grants TIC2003-07036 and TIC2002-0750.

The authors thank Praveen Raghavan from IMEC for providing the benchmarks and the baseline platform used in this work.

References

[1] J. Abella and A. Gonzalez. On reducing register file pressure and energy in multiple-banked register files. In *Proceedings of ICCD*, 2003.

[2] C. Akturan and M. F. Jacome. FDRA: A software-pipelining algorithm for embedded VLIW processors. In *Proceedings of ISSS*, pages 34–40, 2000.

[3] C. Akturan and M. F. Jacome. Caliber: A software pipelining algorithm for clustered embedded VLIW processors. In *Proceedings of ICCAD*, pages 112–118, 2001.

[4] J. L. Ayala and M. López-Vallejo. Improving register file banking with a power-aware unroller. In *Proceedings of PARC*, 2004.

[5] J. L. Ayala, M. López-Vallejo, and A. Veidenbaum. Energy-efficient register renaming in high-performance processors. In *Proceedings of WASP*, 2003.

[6] J. L. Ayala and A. Veidenbaum. Reducing register file energy consumption using compiler support. In *Proceedings of WASP*, 2002.

[7] F. Barat, M. Jayapala, T. V. Aa, R. Lauwereins, G. Deconinck, and H. Corporaal. Low power coarse-grained reconfigurable instruction set processor. In *3th International Conference on Field Programmable Logic and Applications*, 1st - 3rd Sept. 2003, in Lisbon, Portugal, 09 2003.

[8] J. L. Cruz, A. Gonzalez, and M. Valero. Multiple-banked register file architectures. In *Proceedings of ISCA*, 2000.

[9] P. B. et al. Early-stage definition of lpx: A low power issue-execute processor. In *Proceedings of PACS*, 2002.

[10] K. Flautner et al. Drowsy caches: Simple techniques for reducing leakage power. In *Proc. of Int. Symp. Computer Architecture*, 2002.

[11] M. Kandemir, N. Vijaykrishnan, and M. J. Irwin. Compiler optimizations for low power systems. pages 191–210, 2002.

[12] J. P. Koen, K. Langendoen, and H. J. Sips. Application-directed voltage scaling. *IEEE Transactions on Very Large Scale Integration (TVLSI)*, 11(5):812 – 826, October 2003.

[13] I. Park, M. D. Powell, and T. N. VijaykumarV. Reducing register ports for higher speed and lower energy. In *Proceedings of MICRO*, 2002.

[14] Philips nexperia - highly integrated programmable system-on-chip (mpsoc), 2004. http://www.semiconductors.philips.com/products/nexperia/.

[15] 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.

[16] A. Seznec, E. Toullec, and O. Rochecouste. Reducing register ports for higher speed and lower energy. In *Proceedings of MICRO*, 2002.

[17] St nomadik multimedia processor, 2004. http://www.st.com/stonline/prodpres/dedicate/proc/proc.htm.

[18] Ti's omap platform, 2004. http://focus.ti.com/omap/docs/.

[19] A. Vicentelli and G. Martin. A vision for embedded systems: Platform-based design and software. *IEEE Design and Test - Special Issue of Computers*, 18(6):23–33, November 2001.

[20] M. Viredaz and D. Wallacha. Power evaluation of a handheld computer. *IEEE Micro*, 23(1):66–74, January 2003.

[21] V. V. Zyuban and P. M. Kogge. The energy complexity of register files. In *Proceedings of ISLPED*, 1998.

[22] V. V. Zyuban and P. M. Kogge. Inherently lower-power high-performance superscalar architectures. *IEEE Transactions on Computers*, 50(3):268–285, March 2001.