

# CUSTOMIZABLE EMBEDDED SYSTEM ARCHITECTURES

*Peter Petrov*

*Alex Orailoglu*

University of Maryland at College Park  
CSE Department  
Email: ppetrov@ece.umd.edu

University of California at San Diego  
CSE Department  
Email: alex@cs.ucsd.edu

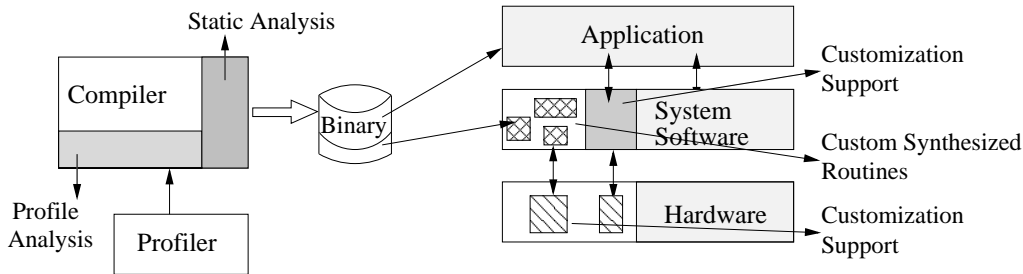
## ABSTRACT

We outline a framework for dynamic application customization for low-power and real-time embedded systems. The framework implements a *cross-layer application-customizable embedded systems platform*, in which the hardware, the system software, and the application tasks are fine-tuned in their *interaction* and *functionality* to the requirements of the program under execution. The traditional design approach has been to put together a set of general-purpose embedded processor cores, deploy a domain specific or a general operating system (OS) and map on it the set of program tasks. In all such systems, the main culprit for the energy inefficiency and the poor real-time guarantees is the general-purpose nature of the hardware architecture, the system software layer, and the interaction between them. The proposed architecture is capable of utilizing application information to boost the performance and lower the power consumption of the most important microarchitectural components such as instruction/data caches and the memory subsystem. We outline a design framework, including CAD support infrastructure and reprogrammable hardware support, for a dynamically customizable embedded system. We outline the underlying algorithms for compile-time extraction of the utilized application properties and we present the architectural principles of the hardware support. Experimental results confirm the efficacy of this novel embedded system architecture.

## 1. INTRODUCTION

The ever increasing integration densities and the affordable deployment of wireless connectivity have led to embedded applications for which a large number of data processing and communication functions are integrated into the same device. Such applications include distributed sensory nodes for environmental and industry data acquisition and control, wireless telephony with data access and manipulation capabilities, personal organizers with various multimedia, network, and security functions, home entertainment, and various others. These applications constitute sophisticated multi-tasking computing systems, which are frequently designed as complex and heterogeneous Multi-Processor Systems-on-a-Chip (MPSoC) [1, 2, 3]. Each processor core in such a system can be assigned multiple tasks for execution. Most of these systems, however, are extremely *energy constrained*, as they often operate on autonomous power sources. Even for many stationary devices with direct access to the power grid, such as set-top boxes, advanced network routers and gateways, energy efficiency is crucial due to the high cost of cooling and packaging. Furthermore, many of these high-end applications impose *real-time constraints*. For instance, many data acquisition and control systems need to process the incoming data samples with a certain speed and to react within the specification limits.

Embedded processors, deeply integrated and utilized for a multitude of SOC sub-systems, have already established themselves as the unquestionable leader in terms of volumes, electronic market share, and revenues, greatly surpassing their close relatives, the general-purpose processors. By virtue of their very nature, many general-purpose optimization techniques are either not directly applicable due to their non-trivial overhead, or just scratch the surface of otherwise promising optimization opportunities, being severely limited by their generality. The dramatic volumes in the embedded processor marketplace necessitate innovative approaches, particularly to surmount the obstacles posed by performance and power considerations in a diverse set of embedded processor market segments. Meeting the more stringent performance and power challenges in the embedded processor marketplace necessitates an ability to exploit the one aspect that differentiates embedded processors from general-purpose processors, namely, advance knowledge regarding their application context.



**Fig. 1.** Proposed framework for cross-layer customizations in embedded systems platforms

Embedded systems have been traditionally designed using general-purpose architectures and system software infrastructure in order to achieve flexible implementation, low design cost, and short time-to-market through the exploitation of off-the-shelf components and design reuse techniques [4, 5]. While good on average, these microarchitectural components are far from optimal in terms of energy and real-time guarantees [6, 7]. Due to their inherent multi-tasking nature and in order to improve the hardware utilization and product maintainability, the target application is usually partitioned into multiple tasks which are scheduled to run on a single or multiple processors. The system software structure, however, follows the principles of general-purpose OS in its interaction with the application tasks and its implementation and assumptions regarding the hardware utilization. The hardware architecture, the OS, and the application layers are separated, each making various general assumptions regarding the layers above and below. The interaction between these layers is also defined in a general-purpose way so as to achieve separation and generality.

The novelty of the envisioned framework is the development of embedded systems, where the *interactions* and the *functionality* of the *architecture*, the *operating systems*, and the *application* layers are fine-tuned and customized to the needs of the particular program or a set of application tasks. This customization is achieved in an automatic way through the help of compile/link-time application analysis together with dynamic support from the OS and the hardware architecture. The dynamic support at the system software layer is achieved through the utilization of pre-designed system software extensions or compiler-generated custom routines, which are registered with the kernel when the program is loaded for execution. The hardware is augmented with software programmable structures such as small memories or sets of registers, where certain application information is stored by the system software or application and utilized dynamically at run-time. Figure 1 presents the conceptual organization of the proposed platform and system software infrastructure.

The elaborate exploitation of application knowledge and its subsequent utilization as a driving force for customizing the embedded processor microarchitecture constitute a conceptual framework that establishes new grounds, enabling a plethora of opportunities for significantly improving both system power and performance [8, 9, 10]. We show that customizable microarchitectural components, such as instruction/data caches capable of utilizing precise application knowledge regarding data reuse regularities can achieve significant performance improvements through miss rate reductions [8], while information regarding the particular memory layout can be exploited in a way that leads to highly optimized tag operations with the concomitant drastic power reductions [9]. Application information regarding data memory footprints can be utilized to efficiently implement virtual memory support, which is not only energy-efficient, but also provides for time-deterministic address translation operations [11, 12]. In all these cases, an application-specific customizable microarchitecture takes informed decisions as to how to handle various architecture-specific actions. As this framework enables the microarchitectural utilization of global application properties identified off-line by compile/link time algorithms, traditional mainstream compiler algorithms remain unaffected.

Since flexible and cost-efficient implementations together with ease of maintenance are the primary advantages buttressing the wide acceptance of processor-centric system designs, it is of paramount importance that the customization support be designed as a microarchitecturally reprogrammable implementation capable of dynamic recustomization. This reprogrammability is achieved on a microarchitectural level, rather than through gate-level approaches such as FPGAs, thus achieving cost-efficient performance and power improvements. We propose an architecture, capable of utilizing application-specific information in a programmable way; hence the ability to recustomize in a post-manufacturing fashion helps effectively cover diverse applications with no need for spinning new silicon, an important advantage in terms of flexibility for a number of high-performance embedded systems. Consequently, the proposed customization architecture constitutes a unified microarchitectural solution capable of



Solving these problems of performance and power simultaneously is a rather difficult task for a general-purpose cache controller.

Cache interference and pollution problems can be resolved in an application-specific environment, wherein more precise information about the inherent reusability can be provided to the cache controller. If the memory instructions were to be grouped according to the inherent reuse characteristics amongst them and each group subsequently mapped to a dedicated cache partition, behaving in the same way as a distinct cache, all conflicts would be obliterated and redundant tag manipulations completely avoided. The size of each cache partition can thus be reduced to no larger than the minimal sufficient size to exploit the inherent reuse for that particular group of instructions.

## 2.1. Partitioning overview

The proposed partitioning analysis utilizes information about the type of reuse of each reference. A formal methodology for determining the reuse type of array references with affine indices is presented in [16]. Since the methodology we propose utilizes information about reuse type in a loop nest, we briefly review the relevant terminology.

By isolating a group of load/store instructions from other unrelated and possibly interfering groups of memory references, the detrimental effects of cache conflicts can be minimized. The grouped instructions can be considered as a set composed of a *leading* reference and several *trailing* references. The *leading* reference does fetch data from memory but misses only once per cache line. All the *trailing* references invariably hit in the cache. Consequently, no tag operations are needed for the trailing references within a partition. Furthermore, if the data access is single strided, the leading reference would miss only in the beginning of a cache line. Therefore, in this case no tag operations are needed for the leading reference as well.

```

for i=1 to N
  f(A[i],A[i+1],A[i+4],A[i+7],A[i+8],A[i+12]);
  g(B[i],B[i+2]);
  h(C[i],C[i+3]);
end for

```

**Fig. 3.** Example of temporal reuse.

## 2.2. Algorithm Overview

We capture the information about the inherent reuse for a particular loop dimension by constructing a Data Reuse Graph (DRG). Each node in the DRG corresponds to a particular load/store instruction or to an already formed group. The edges in the DRG represent data reuse between the corresponding nodes. Each edge is annotated with the particular type of reuse it represents. Additionally, an integer  $k$  is associated to every temporal reuse denotation, representing the number of iterations needed to exploit the temporal reuse denoted by the edge. The number of iterations in turn determines the cache volume needed to exploit the reuse.

The optimal cache partition size, CV (cache volume), varies depending on the reuse type. It is evident that a spatial reuse necessitates only a single cache line. In the case of temporal reuse though, a fixed (but varying amongst memory instructions) number of cache lines are needed in order to exploit the reuse and prevent interference.

The objective of the partitioning algorithm is to group memory instructions with data reuse amongst them and map this group to a cache partition of appropriate size. From a DRG perspective, this implies selecting DRG edges and grouping the neighboring selected edges together into partitions. While each edge selected provides a constant benefit in capturing a reuse, the cost in terms of cache volume to accommodate an edge varies. Consequently, the objective of the algorithm is to maximize the number of selected DRG edges.

An example of data reuse can be seen in Figure 3. We assume a cache line size of one word to simplify the explanation. Figure 4a shows the DRG for the example in Figure 3. The number of cache lines needed to capture the corresponding group reuse is therein shown. The initially appealing solution of a direct, greedy approach is unfortunately inadequate in determining the optimal number of partitions. If we assume an available cache volume of 9 cache lines, a direct greedy approach leads to the result shown in Figure 4b. It is evident though that a solution consisting of a single partition covering all references to the array A but the last one of A[i+12] is superior in that it covers instead four reuses in A and furthermore utilizes all 9 cache lines exactly.



	8K DM p-cache		16K DM p-cache	
	#Misses	MR	#Misses	MR
swim	1,533,664	17.19%	1,257,046	14.09%
tri	173,044	22.92%	165,142	21.88%
ej	1,254,990	14.01%	1,245,372	13.99%
sor	8,690	2.13%	8,682	2.13%

Fig. 6. Partitioned cache miss-rate results

bits from the *cache index* are used to form the new index. The remaining most significant bits from the cache index are replaced by a constant in the newly formed *cache partition index*. The value of this constant determines the *offset* of the cache partition in the original cache. The above reasoning evinces that each cache partition with size  $2^n$  is identified by a pair of numbers  $\{offset, cache\ partition\ index\ size(n)\}$ . The *cache partition index* is formed by concatenating the *offset* and the  $n$  least significant bits from the cache index.

The partition mapping identification is achieved by a hardware architecture utilizing two tables: the *Partition Mapping Identification Table* (PMIT) and the *Partition Identification Table* (PIT). The PMIT is used to define the mapping between load/store instructions and cache partitions. The PMIT is indexed using the least significant PC bits of the load/store instruction from the loop. The size of the PMIT corresponds to the total number of instructions within the loop nest. In practice the size of a loop nest in data intensive applications is rarely large, thus leading to implementations with a small number of entries. When a load/store instruction is decoded, the PMIT is indexed with the least significant bits of the PC. An entry in PMIT contains a value that represents an index into the PIT, which in turn contains a partition defining information. An additional bit, *tr*, is also stored in the PMIT entry to indicate whether the instruction is a *trailing* reference for the partition. The main purpose of this organization is to avoid associative lookups, which are expensive in terms of power. The tables described above are directly indexed; their size, negligible compared to the size of the tag memory arrays, ensures that no significant amount of energy dissipation is introduced. All memory references in a loop nest that are left unpartitioned by the reference analysis are mapped to a dedicated cache partition. This special partition is treated in the same way as the remaining cache partitions.

2.3.2. Computing the cache index

The lookup into the PMIT and PIT is the first step in determining the *cache partition index* and is performed early in the pipeline, thus not affecting the cache access time. Right after the load/store is decoded, the lookup is performed in parallel with the effective address calculation. Figure 5 shows the implementation of the *cache partition index* calculation. The Cache Index Template (CIT) and control signals  $C[i]$  are computed before the actual cache access pipeline stage using the partition information found in PIT. The CIT is defined as having the *offset* value in its most significant bits and zeroes in its  $n$  least significant bits resulting in control signals  $C[i]$  defined as  $C[i] = 1$  for  $0 \leq i < n$ , and  $C[i] = 0$  for  $i \geq n$ . The Effective Address Cache Index (EACI) is the traditional cache index field in the effective address. The Cache Index (CI) is computed using the simple combinatorial logic depicted in Figure 5. The delay of the two gates shown in this figure is the sole, evidently insignificant, increase in the path delay of the cache access data-path.

2.4. Experimental results

In our experimental evaluation of the partitioned cache architecture we have used the following benchmarks: *Swim benchmark* (*swim*), part of the SPEC95fp benchmark suit, characterized by a high cache miss-rate due to a large amount of interference; *Tri-diagonal system solver* (*tri*), a fundamental part of *tomcatv* SPEC95fp benchmark and a major contributor to the high miss rate for the *tomcatv* benchmark, with matrix size of 128x128; *Extrapolated Jacobi-iterative method* (*ej*) on a 128x128 grid; *Successive over-relaxation* (*sor*) on a matrix with size 256x256.

Figure 6 shows the results for the partitioned cache. Within each cache partition, the number of conflict misses is reduced to zero. This follows directly from the way the cache partitions are formed. Consequently, only the cold misses for the references within the partitions need to be considered in terms of cache miss behavior. Furthermore, one can observe that only the *leading* reference of the partition exhibits cold misses, due to its inherent role in bringing data into the partition for subsequent spatial reuse for itself and temporal reuse for the *trailing* references.



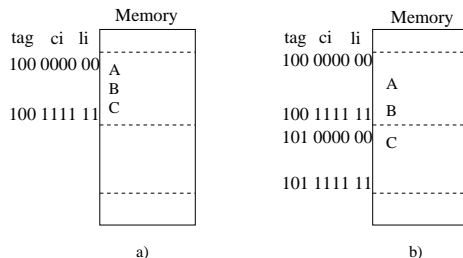


Fig. 8. Memory layouts

case there will be no conflicts within the arrays of data in the cache; no need to perform any tag operations exists, consequently, as the tag part of the addresses is identical. Figure 8a depicts a configuration in which the dataset spans two embedded 0-tag regions. In this case there *will* be conflicts in the cache; nonetheless, the tag fields of two conflicting addresses will differ on average by an exceedingly small number of bits and in the particular example differ only in the least significant bit. If the least significant bit of the tag for a given tag region is 0, then the address tag associated with the subsequent tag region in the memory will differ only in the least significant bit, which will be 1. Such pairs of 0-tag regions are denoted as *1-tag regions*; a *k-tag region* is defined in a similar manner as a group of two  $(k-1)$ -tag regions. It is evident for the example in Figure 8b that only checking the least significant tag bit is enough to determine whether there is a cache miss or hit. Therefore, only the least significant tag bit needs to be read from the tag arrays, while all the remaining bitlines can be gated, thus saving an appreciable amount of power. This tag reduction technique relies on the observation that if all the data being accessed by an application lie within a *k-tag region*, then the *k* least significant tag bits suffice for complete cache conflict identification and therefore can be used as *active tags*.

### 3.2. Unconstrained tag reduction (UTR) technique

The *BTR* technique identifies the smallest *k-tag region*, within which the accessed data set resides and essentially treats this tag region as if it is the entire memory space. This interpretation is valid since the most significant tag bits after the  $k^{th}$  bit are identical for all the tags within the *k-tag region*, while the *k* least significant bits provide complete tag resolution. Yet, as the example in Figure 9a shows, the smallest *k-tag region* does not necessarily provide the minimal number of least significant tag bits for tag resolution. Figure 9 shows three different memory layouts accessed from a particular application loop. The 0-tag regions in which the blocks reside are shown on the left side for each memory configuration.

The data blocks in Figure 9a reside within a 5-tag region, while it is evident that only the 2 least significant tag bits suffice for distinguishing all three tags shown. Selection of the 2 least significant tag bits in this example illustrates that the tag bits to be eliminated are not restricted to being identical as long as the tag bits utilized suffice in completely distinguishing all the 0-tag regions. The *Unconstrained Tag Reduction (UTR)* technique identifies the *minimal* number of least significant tag bits, which are necessary to completely distinguish all the tags generated by the application hot spot and uses these bits as active tag bits.

Figure 9b illustrates a case for which the data lies within four consecutive 0-tag regions, i.e. four different tags are to be generated by the application hot spot under consideration. The *BTR* technique would have selected all 5 tag bits as active bits, since the data set clearly resides within a 5-tag region. Yet, it is evident that in any case of four consecutive tags, the two least significant tag bits suffice for cache conflict identification. It can be observed that these two bits differ for all the tags within such a memory region and therefore these two tag bits provide a full resolution for the set of memory addresses referencing this memory region. In general, if the data set resides within *n* consecutive 0-tag regions, to distinguish all *n* different tags, the  $\lceil \log_2 n \rceil$  least significant tag bits suffice, as any number of adjacent least significant bits taken from a sequence of contiguous binary numbers would in turn form a sequence of distinct, contiguous and yet shorter binary numbers.

The example in Figure 9a that we examined already suggests that the *UTR* technique can handle such cases quite efficiently. Yet further analysis is needed to establish the benefits of the *UTR* in this more general situation. Figure 9c shows another such data layout. The first part of the data occupies two 0-tag regions, while the rest of the code resides in another two 0-tag regions, but separated from the first two. If one considers the unused memory space between the two groups of data as part of the entire group, then the data spans a total of 32 0-tag regions, thus necessitating a total of five least significant tag bits for cache conflict identification. It is evident that this is a



the *UTR* might be far from the optimal one. The *GTR* technique allows the *selection of arbitrary tag bits*, so as to achieve a solution comprising of minimal number of tag bits with complete tag resolution.

### 3.4. Hardware support

The tag array in the cache subsystem is typically implemented as an SRAM array, possibly divided into multiple banks. The SRAM data array contains *wordlines* for each tag data and a *bitline* for each bit within the tag word. By eliminating most of the bitline precharge and discharge operations, our approach greatly reduces the energy dissipation in the tag SRAM array. This is achieved by gating the bitlines according to the minimal number of tag bits required to check for cache conflicts. The sense amplifiers for the disabled bitlines are gated as well. Furthermore, the tag comparator cells are gated in order to perform the comparison only on the required tag bits.

The number of tag bits for each loop needs to be determined before entering the loop, so that the appropriate number of bitlines are enabled. Since this number is fixed for the loop, it can be stored in a special control register, the *Tag Enable (TE)* register, before entering the loop. Each bit in the TE directly corresponds to an enable signal of bitline and sense amplifiers. The default value of this register specifies that all tag bitlines are enabled. The actual value of this register is used to determine the number of bitlines to enable.

### 3.5. Experimental results

Our experimental results indicate the power reductions obtained for the tag arrays of instruction and data caches. In our experimental studies we have used the media benchmarks *adpcm*, *g721*, *gsm*, *epic*, *jpeg*, *mpeg* [17] and an additional benchmark, namely, the *mp3* encoder benchmark. In this paper we present the power reduction results for the GTR technique only. Extended experimental results can be found in [9, 18].

Figure 11 shows the results obtained by applying the *GTR* technique on the data cache for a set of benchmarks. The first triplet of rows shows the number of tag bitlines per application hot-spot, the energy consumption of the tag arrays after using these reduced tags, and the percentage of the energy reduction with respect to the baseline data. The data presented in the first three rows corresponds to a 32k direct-mapped Icache. The subsequent two triplets of rows present the same information but for 2 and 4 way set-associative configurations.

## 4. VIRTUAL MEMORY SUPPORT

Paged virtual memory [19] is a concept providing an elegant and efficient solution for the complex problems of memory allocation, code/data relocation, and sharing in multi-tasking environments. Many modern embedded processors, such as the *Intel XScale* and the *ARM720T*, offer hardware support for virtual memory support in the form of *Memory Management Unit (MMU)*. Typically, it is the responsibility of the system software to interact with the hardware MMU and to implement the system memory management policy.

When virtual memory support is present, the program accesses a virtual address space partitioned into pages, which are referred to as *virtual pages* and are identified by their *Virtual Page Number (VPN)* which constitutes a large fraction of the virtual address most significant bits. During each memory access a translation is needed to map the VPN into a *Physical Page Number (PPN)*. The *Translation Lookaside Buffer (TLB)* is a hardware cache responsible for capturing the most recently used *Page Table Entries (PTE)* for dynamic virtual address translation with no intervention of the system software. TLB misses typically result in trapping into the OS where the missing PTE is retrieved from the *page table* maintained by the kernel. This page table traversal can be implemented either in hardware or executed as a system software routine. The MMU/TLB is usually implemented as a highly associative cache structure so that misses are minimized, which, in turn, results in significant amount of power consumption. It has been shown through direct measurements [20, 21] that around 17% of the total on-chip power for the StrongARM and the Hitachi SH-3 is contributed by the TLB.

A traditional address translation hardware employs a highly-associative buffer, usually referred to as a Translation Lookaside Buffer (TLB), to cache the most frequently used translation entries from the page table. High-associativity is needed due to the very high cost of misses in that buffer. The high-associativity of the translation buffer, however, is extremely *power consuming* and still does not guarantee a conflict-free translation lookup, thus exhibiting *poor real-time guarantees*.



	adpcm	g721	gsm	epic	jpeg	mpeg	mp3
Init Part	2	2	3	4	4,7	5,3,5	8,11,7,7,12
Part	1	1	1	3	1,1	3,2,4	3,2,2,1,2
Index	{2}	{2}	{2}	{3,6,7}	{9}	{1,6,7}	{3,5,4}
					{9}	{1,7}	{5,5}
						{7,7,5,1}	{5,5}
							{5}
							{5,5}
E(4sa)	0.08	5.37	7.86	0.87	1.08	55.6	39.0
Red(%)	62.0	62.0	62.0	62.0	54.3	57.9	58.9
E(8sa)	0.08	5.37	7.86	0.87	13.0	62.6	42.6
Red(%)	78.3	78.3	78.3	78.3	74.0	75.8	76.5

Fig. 15. Energy reductions of DTT

## 4.2. Experimental results

Our experimental study has been performed on a set of widely used embedded applications from the Mediabench [17] set of benchmarks. The simulation is performed with the SimpleScalar toolset. Through benchmark simulation and analysis, the hot-spots and their virtual memory layout are identified. The final power consumption is computed by summing up the energy for all the VPN to PPN translations including the energy needed for the hardware.

Figure 15 shows the energy dissipation for the proposed methodology with a default general-purpose D-TLB of 64 entries. The first row presents the number of initial partitions for each hot-spot at the beginning of the merging phase of our algorithm. The next row shows the number of VPN partitions after applying the merging algorithm. The improvements in terms of minimizing the total number of complete VPN partitions per hot-spot is evident from this data. The next row in the table presents the dimension  $m$  of each VPN partition. This number indicates the DTT volume needed to handle the VPN partition. The last two pairs of rows report the energy achieved by the proposed approach and the percentage improvement compared to the baseline TLB architecture. The first pair of rows present the data for the proposed approach with a 4-way set-associative default D-TLB accessed outside the hot-spots, while the second pair of rows corresponds to an 8-way set-associative default D-TLB. The first row for each pair shows the energy consumption in mJ for the proposed technology, while the second row shows the energy reduction in percentage against the baseline case. It can be observed that the energy reductions are in the range of 54% to 79%. It is evident that the proposed technique consistently achieves high energy reductions even for complex benchmarks, such as *mpeg* and *mp3*, with multiple hot-spots and large VPN partitions.

## 5. CONCLUSIONS

In this paper, we have presented a novel, application-specific customization approach for embedded systems. Increasing processor performance, reducing power consumption, and improving real-time execution guarantees have been identified as essential goals towards achieving cost-efficient and flexible system implementations. This customization framework uses a novel approach for transferring application information to the processor micro-architecture and exploiting it dynamically. The ability to re-customize the processor microarchitecture in field is a significant advantage that preserves the flexibility of general-purpose processors. The methodology is evaluated on real-life applications and significant performance and power improvement are shown.

Customizing the processor core with application-specific information promises to be a powerful technique towards higher performance and lower power consumption. It allows the processor-centric implementation paradigm and its concomitant advantages to be extended to large classes of complex hardware/software codesign systems implementing various modern applications.

## 6. REFERENCES

- [1] W. Wolf, “The Future of Multiprocessor Systems-on-Chips”, in *DAC*, pp. 681–685, June 2004.
- [2] C. Rowen, *Engineering the Complex SOC. Fast, Flexible Design with Configurable Processors*, Prentice Hall, New Jersey, 2004.
- [3] P. Cumming, “The TI OMAP Platform Approach to SoC”, in *Winning the SOC Revolution*, Kluwer Academic Publishers, 2003.

