A Combined DMA and Application-Specific Prefetching Approach for Tackling the Memory Latency Bottleneck

Minas Dasygenis, *Member, IEEE*, Erik Brockmeyer, Bart Durinck, Francky Catthoor, *Fellow, IEEE*, Dimitrios Soudris, *Member, IEEE*, and Antonios Thanailakis

***Abstract*—Memory latency has always been a major issue in embedded systems that execute memory-intensive applications. This is even more true as the gap between processor and memory speed continues to grow. Hardware and software prefetching have been shown to be effective in tolerating the large memory latencies inherit in large off-chip memories; however, both types of prefetching have their shortcomings. Hardware schemes are more complex and require extra circuitry to compute data access strides, while software schemes generate prefetch instructions, which if not computed carefully may hamper performance. On the other hand, some applications domains (such as multimedia) have a uniform and known *a priori* memory access pattern, that if exploited, could yield significant application performance improvement. With this characteristic in mind, we present our findings on hiding memory latency using the direct memory access (DMA) mode, which is present in all modern systems, combined with a software prefetch mechanism, and a customized on-chip memory hierarchy mapping. Compared to previous approaches, we are able to estimate the performance and power metrics, without actually implementing the embedded system. Experimental results on nine well known multimedia and imaging applications prove the efficiency of our technique. Finally, we verify the performance estimations by implementing and simulating the algorithms on the TI C6201 processor.**

***Index Terms*—Computer-aided analysis, design methodology, memory architecture, performance, prefetching.**

I. Introduction

OVER THE last decade, significant advancements in the semiconductor industry have allowed for an unprecedented increase in processor performance. Continued exploitation of instruction level parallelism, microarchitectural innovations, deeper pipelines, and faster clocks are just a few techniques that have led to increased performance. Unfortunately, innovations in memory system design and technology have been unable to achieve the same rate of improvement. These facts have resulted in a continuously increasing gap between processor speed and memory latency, which plagues system developers, especially in application domains where memory accesses are dominant.

Multimedia computing and image processing are domains where memory accesses to large off-chip memories are very frequent. The continued success of media and communications applications have led researchers [1] to believe that these domains will make up over 90% of the workload on embedded and general purpose computers. Additionally, media and network processing are an integral part of many consumer products, including DVD players, cellular phones, and PDAs, so effective computing support is an area of importance for both general-purpose and embedded processors. Multimedia and image processing applications demand high-performance systems to be executed on, otherwise the real-time requirements are not going to be fulfilled, resulting in serious performance degradation. Simply increasing the number of memory ports will improve the performance but at a very high energy penalty for embedded systems.

The importance of the CPU-memory bottleneck has fueled a great deal of research work in this area, resulting in a multitude of approaches to help counteract this trend. Of these, some have proposed improving cache configurations by increasing cache size, tuning cache block size, or changing cache associativity. Other approaches involve software-controlled prefetching or software specifically optimized for cache behavior. Finally, other approaches involve multithreading, out-of-order execution, and usage of special access modes. All of these have been at least moderately successful in reducing memory stall time.

We tackle the problem of hiding the memory latency as part of a unified and formalized technique called Memory Hierarchical Layer Assignment (MHLA). This technique addresses the problem of optimizing the data assignments into memory layers, as well as the block transfers between memory layers. Our technique has a global view of the data reuse search space and selects the appropriate data copy candidates, together with the appropriate block transfer place (where in the code) and type (DMA or CPU controlled). This technique takes into consideration the limited lifetime of data arrays and temporal locality and finds Pareto-optimal tradeoff points in terms of on-chip area, performance, and energy consumption. It employs a prefetch mechanism combined with the DMA mode that most contemporary embedded systems contain.

Manuscript received August 17, 2004; revised May 14, 2005. This work was sponsored by a scholarship from the Public Benefit Foundation of Alexander S. Onassis and from the Marie Curie Host Fellowship project HPMT-CT-2000-00031.

M. Dasygenis, D. Soudris, and A. Thanailakis are with the VLSI Design and Testing Center, Department of Electrical and Computer Engineering, Democritus University of Thrace, Xanthi 67100, Greece (e-mail: mdasyg@ee.duth.gr; dsoudris@ee.duth.gr; thanail@ee.duth.gr).

E. Brockmeyer, B. Durinck, and F. Catthoor are with Design Technology for Integrated Information and Communication Systems (DESICS), Inter-University Micro-Electronics Center (IMEC), Heverlee B-3001, Belgium (e-mail: erik.brockmeyer@imec.be; bart.durinck@imec.be; francky.catthoor@imec.be).

Digital Object Identifier 10.1109/TVLSI.2006.871759

Although prefetching techniques have been proposed since the mid-1960s, and DMA transfers since the mid-1980s, to the best of our knowledge until now, nobody has systematically considered the beneficial opportunities of combining data reuse possibilities, in-place optimizations, and prefetching transfers using DMA, tailored to specific application domains. Even though prefetching is implemented in many compilers or architectures, the performance improvements are not as significant because they do not take into consideration all the optimization possibilities that the application may have. We illustrate that performance (and power) can be improved by the combination of optimizing the data memory hierarchy, prefetching and DMA tailored specifically for the application under consideration. We prove our claims using nine real-life applications of three different processing domains, and report performance improvement of up to 45%.

The rest of the paper is structured as follows. Section II reports on various ways of tackling the bottleneck that memory imposes to the system and describes in detail the various ways of prefetching. Section III introduces the target memory architecture, while Section IV briefly describes the DMA access mode. The next section (Section V) analyzes how prefetching is combined with DMA. The subsequent section (Section VI) discusses the demonstrator applications and experimental results, which prove the efficiency of our technique, concluding with Section VII, where we summarize our findings.

II. Hiding the Memory Latency From the Processor-Related Work

Among the various techniques used to improve memory performance, prefetching has always been one of the most studied and apparently promising method. For this reason, a plethora of different suggestions/implementations have been proposed by various authors for over 30 years.

Prefetching can be divided into three main classes. 1) Hardware prefetching [2]–[11], where the hardware alone decides what data to prefetch and when and where to prefetch the data. 2) Software prefetching [12]–[21], where the hardware supports a prefetching instruction to transfer off-chip elements to on-chip caches. The user, or compiler, then directs prefetching by inserting prefetching instructions into the code. 3) Integrated hardware/software prefetching [22]–[27], where prefetching is being achieved by the combination of software analysis (profiling) and specialized hardware circuitry.

Hardware and software prefetching are not novel techniques. Early work on parallel computing and multiprocessor systems have suggested and evaluated these techniques, in order to speed up execution of memory intensive applications on these systems. The common conclusion is that although prefetching may hide memory latency, it may as well hamper performance, if done inappropriately.

A. Hardware Prefetching

Hardware prefetching schemes are more complex and costly than software prefetching schemes. Furthermore, determining the stride and achieving the appropriate prefetching lookahead for a reference can only take place after a certain amount of execution. Therefore, there will often be more start-up cache misses in hardware schemes, relative to a software scheme, before the hardware scheme reaches the appropriate steady state. Finally, conservative implementation, in order to keep a general purposeness, and limited scope of the applications, are two major drawbacks of the hardware prefetching.

Hardware prefetching initially focused on simple alterations on the on-chip caches; researchers used to add extra circuitry to perform the prefetch function. Enger [2] was the first to propose such a scheme, which was prefetching a fixed number of cache pages. After some years, Bennet *et al.* [3] also proposed some hardware alterations to existing caches and introduced prefetching priorities. Lee *et al.* [4] also conducted another hardware prefetching research work, and reported limitations on existing prefetching techniques on shared memory multiprocessors; they suggested a hardware-controlled prefetching scheme, with the characteristics of long cache lines and instruction lookahead. Existing hardware prefetching schemes have also been studied in detail by Smith [5], who introduced the fetch bypass (or load through), and analyzed the cache memory pollution due to expelling cache lines, which are more likely to be referenced. Jouppi [6] addressed hardware prefetching by using one or more special caches (called stream buffers) to prefetch after a miss in the normal direct mapped cache. However, for large strides the prefetch proved to be inefficient. Baer and Chen [7] described a multiprocessor prefetching method based on branch prediction using a look up tup table that keeps track of the access history. Fritts [8] was one of the researchers that proposed a hardware scheme, with preliminary promising results, but with increased extra circuitry, prefetching to all levels of memory with no coherency control.

Hardware prefetching is still an open issue. Recently, Lusecky and Vahid [9] introduced a bus wrapper with a prefetch unit, in order to speed up execution, but limited inside the CPU core. Cucchiara *et al.* [10] has proposed a costly dynamic hardware prefetching scheme, which increases the memory bus accesses. Finally, Zhuang and Lee [11] have proposed the use of extra circuitry as a filtering mechanism for prefetches to avoid cache pollution, which may be inefficient for applications requiring large amounts of off-chip memories.

B. Software Prefetching

Apart from the hardware prefetching scheme, many researchers have suggested the use of software prefetching. Software prefetching schemes require less hardware support than hardware schemes. Therefore, a software scheme can have a cheaper implementation. A software scheme, however, must generate address calculation instructions and a prefetch instruction for each datum that needs to be prefetched. Therefore, code size increases both statically and dynamically. Most of these schemes are similar and they only differ in their level of sophistication and in where the prefetched data is placed.

The pioneer work in software prefetching was conducted by Porterfield [12]. In his dissertation, Porterfield presented a compiler algorithm for inserting prefetches. This was implemented as a preprocessing pass that inserted prefetching instructions into the source code. He recognized that this scheme was issuing too many unnecessary prefetches, and presented a more intelligent scheme based on dependence vectors and overflow iterations [13]. Since the simulation occurred at a fairly abstract level, the prefetching overhead was estimated rather than presented. Overall performance numbers were not presented. Also, the more sophisticated scheme was not automated (the overflow iterations were calculated by hand) and did not take cache line reuse into account.

The benefits of software prefetching have been recognized in the last decade, and for this reason most contemporary microprocessors support some form of `fetch` instruction that can be used to implement prefetching [14]–[16]. The implementation of a `fetch` can be as simple as a load into a processor register that has been hardwired to zero. Slightly more sophisticated implementations provide hints to the memory system as to how the prefetched block should be used.

The most extensive software prefetching study is by Mowry *et al.* [17]. Software prefetching in a multiprocessor context has been studied by Mowry and Gupta [18], in which they insert by hand the prefetching instructions. Another form of prefetching in multithreaded processors was introduced by Luk [19], in which he proposed a compiler algorithm for locality analysis and explicit pre-execution scheduling for some critical parts of the code, which may flush important data increasing the cache miss ratio.

Finally, it is worth mentioning that state of the art compilers, such as GCC 3.1 [20] and INTEL Compiler 8.0 [21], support software prefetching. Specifically, both compilers have a function `_builtin_prefetch` that minimizes cache-miss latency by moving data into a cache before it is accessed. The compiler or the user inserts these function calls and, if the target processor supports software prefetching, then the CPU may use these hints.

C. Hardware/Software Prefetching

Integrated software and hardware prefetching is another class of prefetching. Software prefetching relies exclusively on compile-time analysis to schedule fetch instructions within the user program. In contrast, the hardware techniques initiate prefetching opportunities at runtime without any compiler or instruction set support. Noting that each of these approaches has its advantages, some researchers have proposed mechanisms that combine elements of both software and hardware prefetching.

Gornish and Veidenbaum [22] were the first researchers to suggest combining the hardware and the software to outperform existing prefetching schemes. They described a variation on tagged hardware prefetching, where the compiler calculates at compile time what will be prefetched and when. Chen [23] was another researcher that suggested the use of a prefetch engine in a similar work. Xia and Torrellas [24] identified that the data cache is a bottleneck to multiprocessor systems and they suggested an integrated hardware/software technique to eliminate data cache misses. Chiou and other researchers at the Massachuchettes Institute of Technology (MIT) [25], addressed also the problem of prefetching in multilevel memories using dedicated prefetch engines. Wang *et al.* [26] proposed a cooperative hardware-software prefetching scheme called Guided Region Prefetching, where they reported that not every off-chip data should be prefetched, but only some data, avoiding cache pollution and keeping bus contention low, by prefetching when the memory bus is idle. Although their work is general, it is not application specific and data reuse or in-place optimizations are not handled. Furthermore, the accuracy is not high (52%–81%) and sometimes the aggressive prefetching displaces useful data.

Finally, Al-Sukhni *et al.* [27] suggested that to achieve more efficient prefetching, it should be tailored to the needs of the specific application.

To our knowledge, no one has combined the opportunities for data reuse and in-place optimizations with DMA-based prefetching in an automatizable exploration technique for real-life applications, where the steering does not need to be done by the designer. Moreover, our approach presents to the designer all possible tradeoffs in terms of energy or performance or on-chip memory space, taking into consideration the characteristics of every layer of the memory hierarchy, data reuse possibilities, in-place opportunities, contention of memory bus, and scheduling of block transfers. Our technique allows finding the optimal assignment, respecting the constraints that are imposed in a predictable way for both memories and hardware caches, as well as the optimal prefetching opportunities for every block transfer.

III. Target Memory Architecture

The MHLA technique that we present here is part of the **A** **T**oolbox for **O**ptimizing **M**emory **I**/O **U**sing geometrical **M**odels (ATOMIUM) tool suite [28], which supports the Data Transfer and Storage Exploration (DTSE) methodology. DTSE is a systematic, step-wise, system-level methodology for optimizing data dominated applications for memory accesses and memory consumption [29]. The main goal of the methodology is to start from the source code specification of the application (e.g., in the C language) and determine an optimized execution order for data transfers, together with an optimized memory mapping for the data (on a potentially predefined or customized memory organization). Concerning the target memory architecture, the tools of ATOMIUM are flexible; various multilevel memory architectures can be used, consisting of scratch pad memories and caches.

Although the MHLA technique is general and can be used in many memory architectures, we have used a specific architecture for our experiments. Our memory architecture is a simplified version of the one used in TMS230C6000 DSPs [30] (Fig. 1). We selected this platform because we also have a TIC6201 developer's board, which allowed us to compare our estimations with real measurements. This platform consists of a 'C6000 CPU core with two independent 32-bit data paths for accessing on-chip or off-chip memory, two blocks of on-chip memory, a program/data bus, a peripheral off-chip DMA controller and an external memory. On-chip memory varies and is one of the variables that our prototype tool explored during its execution. On the data bus, a data memory controller is introduced, which services requests to internal memory by either the CPU or the DMA. Off-chip and on-chip memory are used in a continuous memory map, which means that some

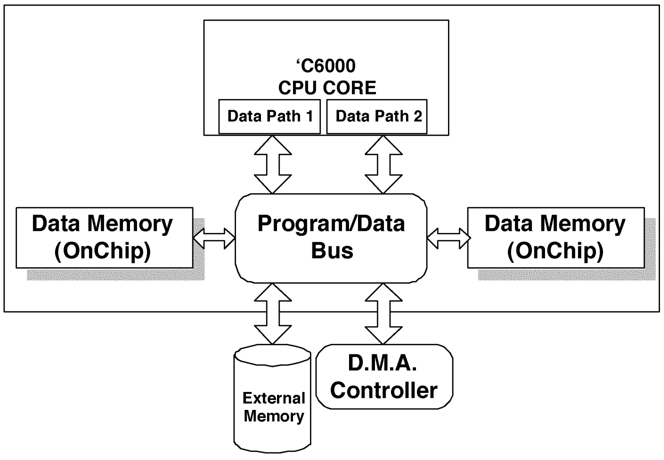


Fig. 1. In our MHLA technique, we are able to use a platform description file to describe many popular architectures. In this research work, we use the memory architecture of the TI C6201 as an exploration platform.

memory addresses are located in the on-chip space, some are reserved, and some are located off-chip.

IV. DMA Overview

Memory transfers between peripherals or off-chip memory and main processors can be achieved in many modes. One mode is the DMA [31] that is supported by a lot of contemporary architectures. In order to support this mode, a special hardware engine is used, which is called a DMA controller. We selected the DMA controller that is used in the C620x/C670x DSP series of Texas Instruments (TI), because we have at our disposal a TI C6201 DSP board. DMA controllers are similar between vendors, therefore, the estimations of our tool are applicable to non-TI-based architectures as well.

Initiating a DMA transfer is a bit complex and it depends on the architecture used. A number of registers have to be set and a DMA_START request has to be given to the DMA engine. The CPU can operate concurrently with the DMA as long as there are no data dependencies. Every DMA_START has an associated DMA_WAIT statement that is usually placed as late as possible, but no later than the place that the specific data transferred through DMA is required by the processor. Ideally the transfer would have been completed by the time the processor reaches the DMA_WAIT and, thus, no cycles will be spent stalling. However, putting the DMA transfer earlier, requires more on-chip memory space. This side effect arises from the increased life time of on-chip memory data, rendering in-place mapping techniques for memory compaction less effective [29]. In realistic cases, a complete removal of the stalls is not always possible, which means that some cycles are spent in DMA_WAIT statements. Alternatively, another thread could be initiated. Of course, interleaving of processing threads could also result in an overhead, so a tradeoff is present that is not further explored in this paper.

Usually, in current embedded systems, more than one DMA channel is available to the developer. The DMA channel with the lowest number (i.e., DMA0) is the channel with the highest priority, and *vice versa*. The highest priority means the DMA engine will first execute this block transfer and, next, the DMA with the lower priority.

V. Prefetching Combined With DMA

To alleviate the bottleneck of the memory latency, we propose a unified technique that incorporates many characteristics that are absent in all related work on prefetching. On one hand, these characteristics increase the flexibility of exploration of the memory assignments and scheduling, while on the other hand, extensively optimize the performance and energy consumption. In order to use our prefetch technique combined with DMA, two additional key points are to be decided: what are the prefetching candidates and how to effectively use the DMA mode for the block transfer of these candidates.

A. Prefetching Candidates

In order to specify the prefetch candidates, we put additional features into our prototype MHLA tool [32]. MHLA explores all different copy candidates of arrays and selects the most efficient ones for the given memory constraints, using accurate cost models of the ATOMIUM suite. In order to select the copy candidates, in-place optimizations are used to reduce memory size requirements. Arrays and copy candidates are assigned to specific memory layers in an exploration phase, estimating what the energy consumption and performance improvement and selecting the most efficient ones. The exploration process is guided through a number of heuristics, significantly decreasing the time required to reach optimum solutions. Not a single solution, but a pool of different solutions, is selected and presented to the designer in the form of Pareto points.

MHLA did not consider prefetching (or time extensions as we call it) when it was first presented [32]. This has forced us to add an important extra functionality to our technique, as well as to this prototype tool. In order to incorporate prefetching combined with DMA transfer, some features were added. Specifically, we added performance estimations of prefetching to block transfers, an experimental DMA model based on TI measurements, on-chip memory size estimation of prefetching, and life-time analysis of prefetch candidates. The MHLA tool takes advantage of temporal locality and limited lifetime of the memory architecture constraints, while taking into account the copy overhead, and explores all the possible different layer assignments.

Prefetch candidates or copy candidates (CC) can be identified automatically in the MHLA tool by using a geometrical data reuse analysis technique [33]. The output of this technique is a file which describes the size of every buffer, the number of accesses that can be captured by this buffer, and the number of accesses required to fill this buffer (called misses). The ratio of these two numbers is the reuse factor. This output can either be created automatically using the MHLA tool or can be created by the user manually. In our current work, prefetching candidates are limited to arrays that are reused. Thus, we select prefetching candidates for arrays that have a reuse factor greater than one. Of course, this is not a hard constraint and can be lifted in a future version, if we consider that arrays that are not reused have a reuse factor of one (every element is written once and is read once). From the list of all possible prefetch candidates (and their combinations), the selection is based using energy and performance heuristics, which will not be described here, due to lack of space.

B. Using DMA Transfers

Any selected prefetch copies can be implemented in the code in such a way that the CPU controls the transfer (using for-loop or the “memcopy” function), or it can be implemented in a DMA transfer mode. The second key point is to replace these memcopy commands with DMA commands. A DMA transfer consists of a DMA_START and a DMA_WAIT command. The DMA_START command instructs the DMA controller to start the DMA transfer for a given block size, from a specific off-chip (or on-chip) memory address to a specific on-chip (or off-chip) memory address. The DMA_WAIT command is a crucial point in the DMA implementation. In the DMA_WAIT command, the processor stalls, waiting for the DMA transfer to be completed. When the DMA transfer is completed, the processor can resume the execution of the application. The DMA_WAIT statement usually is placed after a computationally intensive nested loop. Thus, while the CPU is performing the computations, the DMA engine performs the memory transfer.

The core of the proposed approach is to hide the prefetch operation for the CCs from the CPU using DMA, and to select the best matching CCs. This means that the DMA transfer must take place in parallel with a CPU computational task. Thus, when the CPU finishes the computation and continues the execution of the program, the off-chip data that will be required will have been transferred to on-chip memory and available. This results in hiding the off-chip memory access latency. In our MHLA tool, we have incorporated two DMA transformations that can be used to increase the flexibility of the technique, and achieve greater performance gains: DMA priorities and DMA pipeline.

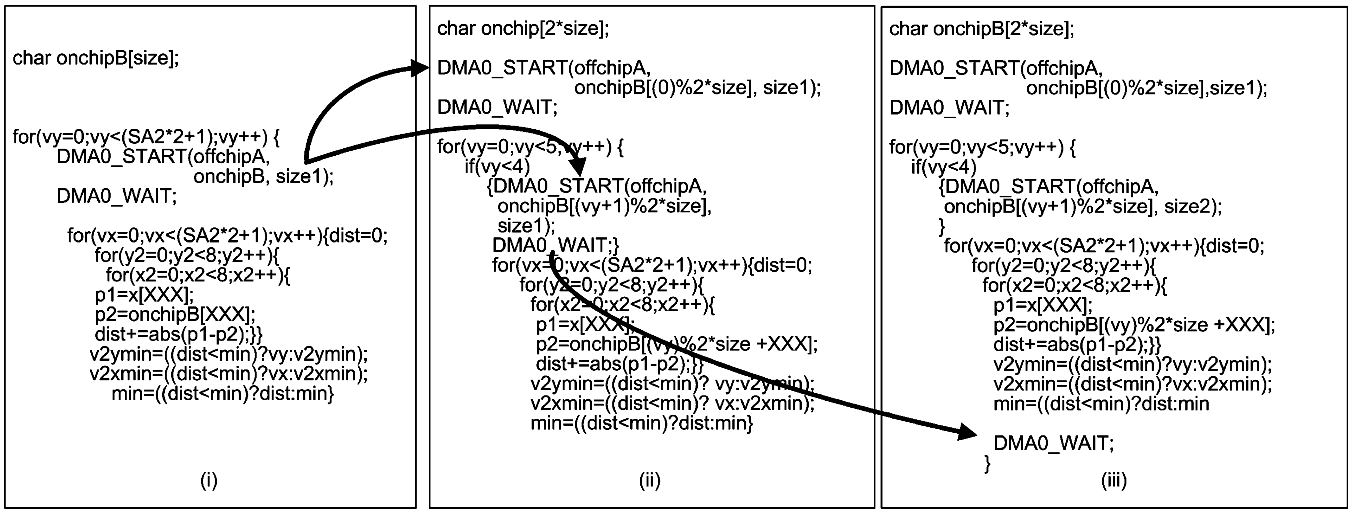
1) DMA Priorities: All contemporary DMA controllers support multiple DMA priorities. When it is required to prefetch more than one array, we have to use multiple DMA priorities. Assuming that we want to start a DMA [DMA(2)] transfer in advance, but in that loop another DMA transfer exists [(DMA(1)], which prohibits us to use the same priority (DMA transfers can not have the same priority simultaneously), and forces us to use different priorities. DMA transfer [DMA(2)] has to be initiated using a lower priority. As long as the DMA(1) is active the DMA(2) is suspended.

Generally speaking, DMA priorities enable us to schedule DMA transfers over long loops with low priorities, and in the nested loops of this loop to schedule DMA transfers with higher priorities. When the application enters the big loop, the DMA with low priority is activated. After some time, the program counter reaches a nested loop of the loop that has a DMA transfer. The first (low-priority) DMA is suspended, allowing the high-priority DMA of the nested loop to be completed. When the high-priority DMA completes, the low-priority DMA transfer resumes. DMA priorities are used to simplify DMA transfer scheduling and to allow greater exploitation of the loops. In our tool, we estimate the critical path of every loop and the time required for a DMA transfer, allowing us to schedule DMA transfer with loops of similar (or less) cycles.

In our technique, DMA priorities are used as follows. If we want to prefetch an array to on-chip layers, then we use the highest DMA priority. If another array needs to be prefetched in a coincident time period with a second array, then the array that belongs to the innermost loop has the highest priority. This is valid for two reasons. First, the array of the innermost loop is usually much smaller in size than the array in an outer loop, and thus, the DMA transfer of it will be much faster, which means the DMA channel (or, in other words, the DMA resource) will be released quicker. Second, arrays belonging to innermost loops are usually arrays that will be accessed much more often and in very short periods of time, than arrays belonging to higher loop levels. Thus, it is important for these arrays to be on-chip before they are accessed. A question that arises is, what happens when two arrays that need to be transferred using DMA belong to the same innermost loop. In that case, our architecture supports multiple concurrent DMA transfers which solves our problem. In the case that our architecture supports less concurrent DMA transfers, then we have to select what array will have the highest priority. In our technique, we believe that, in this specific case, the bigger the size of the array the higher priority it should have, because DMA usefulness is increased with the size of the DMA transfer (the bigger the DMA transfer the greater the performance savings). If all arrays have the same size then there is no point in selecting a specific one and we chose randomly. Note that the fact that both of the arrays belong to the innermost loop means that both of these arrays should be present by the time they are executed. If one of these arrays is not present in the on-chip memory, then the CPU will stall in the off-chip to on-chip access. This means that the selection of the arrays that will be fetched first (the one with the highest DMA priority) is not a crucial decision.

2) DMA Pipeline: Sometimes a DMA transfer can not be initiated early, due to existing data dependencies. In that case, DMA pipeline can be used to find another solution (which will increase on-chip memory size, but may improve performance). If the pipelined version is a new Pareto point, it should be included in the Pareto graph with the nonpipelined version.

DMA pipeline creates tradeoffs between on-chip memory size and performance, which we explain using an example (Fig. 2). In this example [Fig. 2(i)], a DMA operation transfers elements from an off-chip memory (off-chipA) to an on-chip memory (on-chipB of “size” bytes). In the subsequent loop, a dependency prohibits the masking of this DMA transfer by moving the DMA_WAIT statement after the loop. This code requires on-chip size of size (declaration of array $\mathrm{onchipB}$ [Fig. 2(i)]). The DMA transfer in this code is not done in parallel with a computational loop because a data dependency exists on array $\mathrm{onchipB}$. In order to break this dependency, loop pipelining is employed and $\mathrm{onchipB}$ is doubled in size [Fig. 2(ii)]. Moreover, in the same figure, we illustrate that the first iteration of the DMA transfer is unrolled in order to create the DMA pipeline. This results in the situation, when the CPU computes the motion vectors of the current iteration, the DMA operation transfers elements of the next iteration. The dependency is now broken and the DMA_WAIT statement is moved to the end of this loop nest [Fig. 2(iii)]. Again, this DMA0_wait statement has to be evaluated and measured in terms of CPU cycles spent stalling there. A successive DMA hiding operation will have zero cycles spent waiting at the DMA_WAIT



```
char onchipB[size];

for(vy=0;vy<(SA2*2+1);vy++) {
    DMA0_START(offchipA,
                    onchipB, size1);
    DMA0_WAIT;

    for(vx=0;vx<(SA2*2+1);vx++){dist=0;
      for(y2=0;y2<8;y2++){
        for(x2=0;x2<8;x2++){
      p1=x[XXX];
      p2=onchipB[XXX];
      dist+=abs(p1-p2);}}
     v2ymin=((dist<min)?vy:v2ymin);
     v2xmin=((dist<min)?vx:v2xmin);
        min=((dist<min)?dist:min}

                         (i)
```

```
char onchip[2*size];

DMA0_START(offchipA,
                  onchipB[(0)%2*size], size1);
DMA0_WAIT;

for(vy=0;vy<5;vy++) {
   if(vy<4)
     {DMA0_START(offchipA,
       onchipB[(vy+1)%2*size],
      size1);
     DMA0_WAIT;}
     for(vx=0;vx<(SA2*2+1);vx++){dist=0;
        for(y2=0;y2<8;y2++){
          for(x2=0;x2<8;x2++){
           p1=x[XXX];
           p2=onchipB[(vy)%2*size +XXX];
          dist+=abs(p1-p2);}}
         v2ymin=((dist<min)? vy:v2ymin);
         v2xmin=((dist<min)? vx:v2xmin);
         min=((dist<min)?dist:min}

                         (ii)
```

```
char onchipB[2*size];

DMA0_START(offchipA,
                  onchipB[(0)%2*size],size1);
DMA0_WAIT;

for(vy=0;vy<5;vy++) {
    if(vy<4)
      {DMA0_START(offchipA,
       onchipB[(vy+1)%2*size], size2);
      }
      for(vx=0;vx<(SA2*2+1);vx++){dist=0;
         for(y2=0;y2<8;y2++){
           for(x2=0;x2<8;x2++){
            p1=x[XXX];
            p2=onchipB[(vy)%2*size +XXX];
            dist+=abs(p1-p2);}}
            v2ymin=((dist<min)?vy:v2ymin);
            v2xmin=((dist<min)?vx:v2xmin);
            min=((dist<min)?dist:min

            DMA0_WAIT;
       }
                         (iii)
```

Fig. 2. Sometimes data dependencies prohibit prefetching. In these cases, DMA pipeline can be used to break the dependencies and initiate in advance the prefetching transfer with some penalty in terms of on-chip memory size.

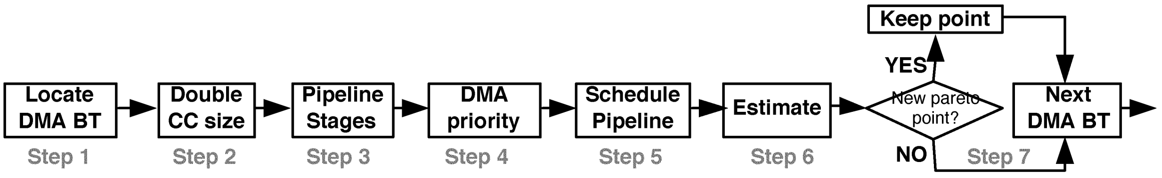


Fig. 3. MHLA can estimate whether a DMA pipeline is beneficial for performance, with some penalty in on-chip memory size, in seven steps.

statement. Finally, the first DMA0_wait statement, which is the prepipeline context, can not be masked due to dependencies.

The implementation of the DMA pipeline, in the MHLA technique, is done in seven steps (Fig. 3). After the selection of all copy candidates and the allocation of on-chip size, and the decision of where DMA transfers will be used, another phase follows, which target performance improvements through the DMA pipeline. Note that in the prototype tool MHLA, no algorithmic transformations occur but the performance and the on-chip memory size are estimated. First, we locate the DMA transfer that could not be moved earlier due to data dependencies. The DMA pipeline is used only on DMA block transfers that have data dependencies and, thus, they can not be moved earlier. This DMA transfer will belong to a nested loop of a specific loop-level (e.g., i) and will transfer elements required for the loop that follows in the subsequent level (e.g., $i+1$). Second, we double the size of the CC that the DMA transfer copies data. In our current implementation, we only consider pre-fetch and not post-write, thus, DMA transfers are used only to transfer elements from off-chip to on-chip memory. Third, we split the DMA operation in the pipeline stages: a prologue phase that can not be moved and transfers the initial data on the first half of the on-chip array; and in a main body, similar with the first, that transfers all the elements of the subsequent iteration of that nested loop level to the on-chip array (to the other half of this array). Note that the size of the on-chip array has been doubled and, thus, it can hold data required for two iterations. Fourth, we check whether a DMA operation is already done in loop $i+1$. If another DMA transfer is done in loop $i+1$ then the priority of the new DMA operation must be lower. It is logical, but we have also validated this with experiments, that the greater the level of the nested loop level the DMA operation is scheduled, the higher the priority it should have. Fifth, we schedule the stages and we put the pipelined DMA operation in the beginning of the loop $i+1$ and the DMA_WAIT in the end of the loop $i+1$. Sixth, we compute the old and new performance. We estimate both the cycles of the DMA operation, using in-house mathematical formulas, and the critical cycles needed for the computational loop, using functions from the tool Storage Bandwidth Optimization [34]. In our technique no algorithmic transformations have been done; only the prototype tool MHLA is used. The cycles needed for the pipelined loop will be the cycles needed for the prologue phase added with the higher number of cycles of the DMA operation or execution cycles of the computational loop (note that they are happening in parallel). In that case, we know the performance and on-chip memory size before and after DMA pipeline. In the seventh step, we compare the estimated performance with and without pipeline. If we have a new Pareto point, then we add this to the list of Pareto points and MHLA continues working with the next DMA block transfer (BT), until all DMA transfers have been examined.

The position of the DMA_WAIT statement is crucial to the system performance because if the DMA_WAIT statement is placed early, the DMA transfer will be incomplete and CPU will stall waiting for the transfer to be completed. If the DMA_WAIT statement of array A is placed after the read access of this array, then the DMA transfer of elements of A from off-chip to on-chip may not have finished by the time the elements from the off-chip memory are needed, resulting in accessing invalid memory positions. The position of placing the DMA_WAIT statement together with the DMA_START statements are very crucial and affect performance directly.

C. Estimating Performance and Energy Improvement

Our MHLA technique carries out a fast estimation of the performance and energy improvement and can quickly determine the optimal implementations. Every implementation is described by the selection and assignment of the arrays and the prefetching candidates into the multilayer memory architecture. In order to perform the estimations, we use some profiling information (number of accesses for every array and where in the code they are happening). This profiling information has been extracted using the ATOMIUM tools. Once we profile the application, we use the MHLA tool to estimate various assignments into the memory layers. Note that no actual modifications are done to the code. The estimations are obtained using some user-defined energy and performance functions. These functions should follow two basic principles: 1) an off-chip access is more expensive, both in energy and in time, than an on-chip access, and 2) a bigger memory is slower and less energy efficient per access than a smaller one. Concerning the energy estimations, we are using a modified version of cache access latency and power estimation tool (CACTI) [35], which has been slightly altered according to data sheets from our industrial partners.

Generally speaking in this context, we do not care to provide energy and timing estimations closely to hardware implementation, because we perform a faster exploration. In this high-level exploration, only relative numbers and gains are required. As it was proven by the experimental results, if an estimation pinpoints one solution as better than another, then this will also be valid when the solutions are implemented. The estimated and measured numbers will differ but the optimal solutions will be the same.

Our tool can quickly explore all different tradeoffs for a given memory size in terms of performance and energy consumption. In order to achieve this, it uses heuristics which pinpoint the solutions to be estimated, along with estimators to compute the energy and performance gains. If either the energy consumption or the performance, is better than any other previous solution (for the same memory size), then this is a new Pareto-optimal point. The exploration continues until either all the possible combinations have been explored, or a user defined time-limit has passed.

VI. Demonstrators and Experimental Results

Our experiments were performed using real-life applications of motion estimation, video encoding, and image processing domain. Specifically, our demonstrator applications belong to the following three categories: (*A*) Motion Estimation (ME) Kernels: *1*) full search (FS) [36]; *2*) hierarchical search [37]; *3*) parallel hierarchical one-dimensional search [36]; *4*) 3-step logarithmic search (3SLOG) [36]; *5*) spiral search [38]. *B*) Video Encoding: *6*) quadtree structured difference pulse code modulation (QSDPCM) [39]. *C*) Image Processing Algorithms: *7*) cavity detection [40]; *8*) wavelet [41]; and *9*) three-dimensional shape reconstruction (3-DR) [42].

Nine applications of different domains have been selected as test vehicles to evaluate our technique: five multimedia algorithms, one video encoder, and three image processing algorithms. The ME algorithms are fundamental multimedia algorithmic cores used in many embedded systems. The QSDPCM is an interframe compression technique that uses a hierarchical search and implements a full video encoder. This application has similar complexity with MPEG2 and it is not a simple code. In addition, we examine cavity detection, which belongs to the medical image processing domain. It is used to detect tumor cavities in computed tomography pictures of the brain. Furthermore, we examine one of the subalgorithms of a 3-D image reconstruction algorithm [42], which is 600.000 lines of C++ code. Finally, we include in our test vehicles a 5-level 9/7 tap row/column inverse wavelet transformation algorithm.

Concerning the experimental setup, all the estimations and measurements are performed using typical and representative parameters of these applications. Our experiments for the motion estimation algorithms are carried out using the luminance component of QCIF (144×176 pixels), CIF (288×352 pixels), and D1 (576×720 pixels) frames of the sequences of two successive frames with the name "Akiyo," "Tennis," and "Barb," respectively. A reference window is selected to include 15×15 pixels $(2p + 1 \times 2p + 1)$, where $p = 7$ is the search space. Our experiments for the cavity are carried out using two medical images of a human brain; one at 640×400 pixels and one at 1280×800 pixels. The 3-D image reconstruction used as input pictures of 640×480 pixels. Finally, the input to the wavelet is a picture named "lena" with dimensions 512×512 pixels.

We used our prototype tool to explore different allocations and assignments of arrays to memory layers, yielding all the tradeoffs in terms of performance, power dissipation, and on-chip memory size. Additionally, we validated our estimations using an instruction set simulator, which reported the same performance improvements. Finally, in order to further analyze and confirm our findings, we ported one application (QSDPCM) to a hardware development board, which confirmed again our initial estimations.

A. MHLA Estimations

The MHLA prototype tool that implements the presented technique, performs an automatic analysis and exploration to discover all the performance, energy consumption, and on-chip memory size tradeoffs. Energy and performance estimations are done only for the memory accesses and not for the CPU. Memory energy dissipation is the dominant contributor in the overall system characteristics, and it is believed to be 50%–80% of the overall system power consumption [29]. Thus, optimizing the memory energy consumption results in gains for the whole system. This is also true for the performance of the system. We have not yet developed models for the CPU power consumption and performance and, thus, we can not estimate the system's power consumption and performance. It is expected though, that the memory optimization will optimize to a great extent the embedded system.

The outputs of the MHLA tool are a number of optimum implementations (tradeoffs) for specific architectural platform descriptions. For example, considering the wavelet application

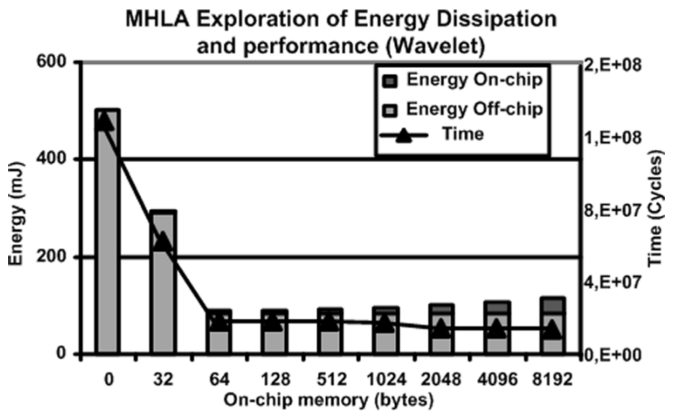


Fig. 4. While the increase of on-chip memory reduces overall energy consumption, after some application-dependent memory size, the energy consumption starts to increase without performance gains. In Wavelet, this is noticed after 64 bytes.

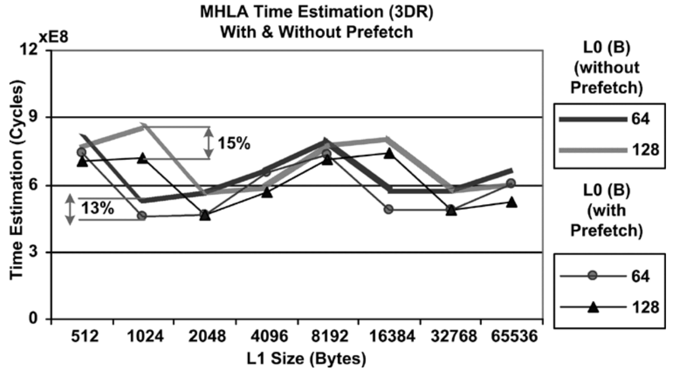


Fig. 6. Taking into consideration the prefetching opportunities, enables further optimization of the application. Here, we see performance estimations of 3-D reconstruction for different L0, L1 sizes, with and without prefetching.

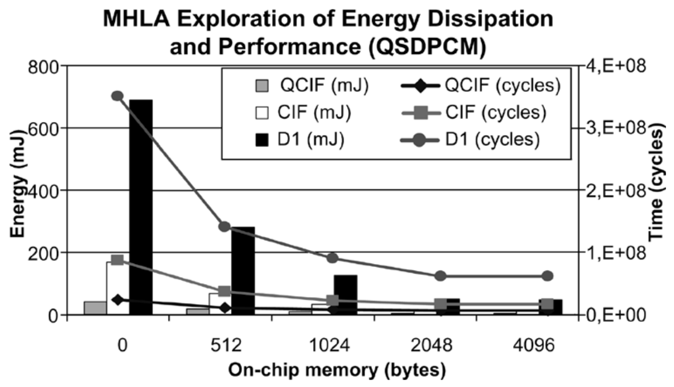


Fig. 7. Energy and performance improvement is similar for different sizes of input frames.

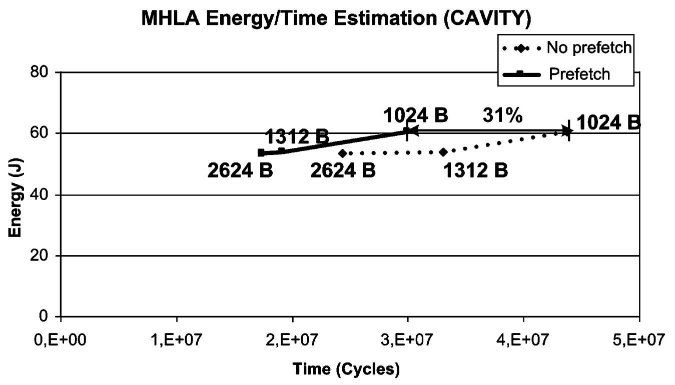


Fig. 5. MHLA estimates tradeoffs between performance and energy consumption for different memory assignments, for using prefetching/DMA or not. The use of prefetching on CAVITY shows that, for the same energy consumption, prefetching can boost performance up to 31%.

after a fast exploration of different memory assignments, MHLA estimates the various tradeoffs (Fig. 4), helping the designer decide what implementation suits better for his product. In our case, we found out that further increase of the L1 size of 64 bytes does not improve performance, since all important arrays and copy candidates have been assigned to on-chip memory. Increasing the on-chip memory size will improve performance by some cycles, but it will deteriorate power consumption. For increased on-chip memory sizes, the power consumption of on-chip memory is becoming a dominant power contributor.

The MHLA tool also supports prefetching with DMA (or time extensions as we call it), which improves the performance even further. Even though optimum assignments are found and reported to the user, enabling prefetching (where this is possible) increases the performance gains for the same power consumption. The performance/energy estimations on CAVITY (Fig. 5), show the optimum implementations in terms of performance for an architecture with and without using prefetch. If prefetch is enabled, then performance is increased from 10% up to 30% for the same energy dissipation. We gain in performance because the block transfers are scheduled in parallel with processing loops and are implemented using DMA transfer. The energy consumption stays the same because our tool estimates power based only on accesses to the memory hierarchy. Power consumption spent in CPU or in DMA controller is not modeled. However, it is anticipated that using DMA for block transfers, power savings can occur, as it will be discussed later in this section.

Similar results exist for 3-DR, where we performed an exhaustive exploration for different on-chip L0 and L1 sizes (Fig. 6). Estimations show that prefetching increases 3-DRs performance up to 15% for the given memory characteristics.

In order to investigate the effect of input images of different sizes, we used MHLA to estimate the performance and energy dissipation of our selected applications for different commonly used frame sizes. We selected the most common; QCIF (176× 144), CIF (352 × 288), and D1 (720 × 576). The estimations reveal that the energy and performance relative gains are similar for the same on-chip memory (of course the absolute energy dissipation and performance numbers are different). For example, in the QSDPCM (Fig. 7), using an on-chip memory size of 1024 bytes has an average performance gain of 74.74% and an energy gain of 81.95% for input frame of D1 dimensions, compared with an architecture without any on-chip memory. The same gain is achieved for CIF input frame (energy and performance gain of 81.88% and 75.46%, respectively), and QCIF input frame (energy and performance gain of 82.50% and 75.30%, respectively).

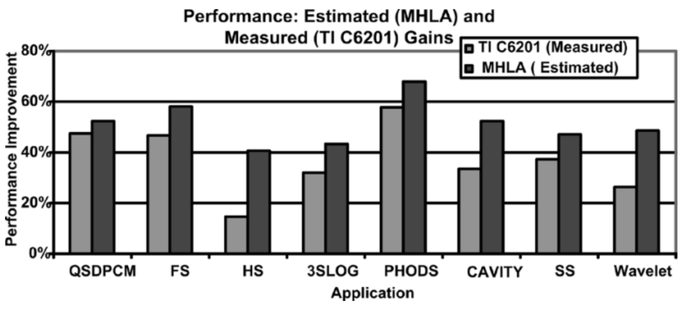


Fig. 8. Measured gains on TIC6201 validate the fast MHLA estimations and show the benefits of using prefetching on an optimum memory assignment.

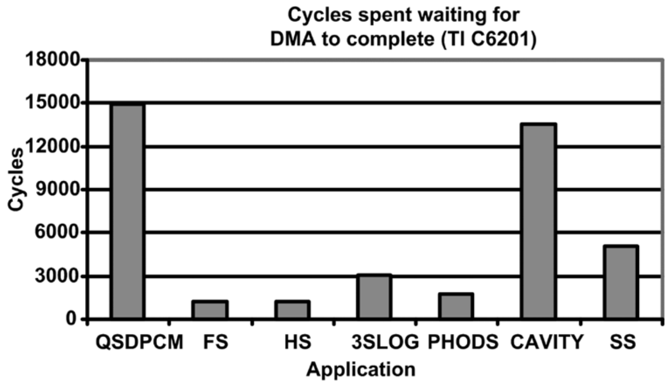


Fig. 9. CPU stall cycles in DMA_WAIT statements are negligible compared to the total cycles of every application, showing that prefetching has been hidden effectively.

B. Simulating MHLA Solutions on TI C6201

In order to evaluate the quality of the estimated results, we used one popular instruction set simulator (ISS); Code Composer Studio [30] for simulating the TI C6201, which provides a very detailed simulation (it simulates DMA transfers). This is a design environment for embedded processors based on TI. Performance measurements that are obtained through the simulation with this IIS are very close to real measurements, and provide a precise method for profiling applications that will be ported to TI processors. The drawback of using this embedded development suite, is the actual long execution time (simulation) of the image.

The TIC6201 measurements are taken as follows. Initially, MHLA performs an exhaustive exploration of an application for varying on-chip sizes, for our specific architecture description. The output is the list of arrays and copy candidates that are placed on-chip and the arrays that are placed off-chip, together with the performance (and power) estimation of this memory allocation. In order to compare the estimations with simulated measurements, we are implementing these solutions by making algorithmic transformations to the C code of every application. Thus, for every application, different (transformed) versions are created, which have the same functionality. The transformations involve the insertion of additional arrays, namely copies, which are copies of arrays that are placed on the off-chip memory. The transfer of data elements from off-chip to on-chip is done using DMA instructions, specific for the processor that we are using. We also performed loop unrolling and pipeline in the transformed algorithms, in order to better utilize the DMA mode. The applications are compiled using normal compiler flags(−O2).

The estimated results obtained from MHLA are compared with the implemented and simulated (measured) results, obtained by the IIS (Fig. 8). The figure shows relative performance improvement of both estimated and simulated results. Evidently, our technique overestimates the performance improvement that can be achieved for a specific memory assignment. This is due to the fact that MHLA considers only the time spent in memory accesses (on-chip or off-chip), which even though it is the dominating factor in the overall performance, it is not the only one factor. Control overhead or other CPU delays also plays a small part.

The general conclusion is that the real measurements follow the estimated results of MHLA. Even though the performance improvement is not the same, when MHLA estimates an optimum solution, this is indeed an optimum implementation in reality, because it is validated by the IIS. The values depicted (Fig. 8) show the percentage of the performance improvement, compared with the original (1-performance_optimized/performance_original). PHODS, FS, and QSDPCM have the highest average measured performance improvement (57%, 46%, and 47%, respectively), because they have computational intensive loops that can effectively hide the DMA transfers from off-chip to on-chip layers that are happening at the same time. Furthermore, significant improvements are measured on all of our test benches, indicating that prefetching is most beneficial when prefetching is done after a thorough exploration of different memory allocation and assignments of the multidimensional signals of our applications.

A measure of the efficiency of the DMA mode is the cycles of the CPU that are spent in stalling during the DMA_WAIT statement. An efficient DMA implementation depends on the minimization of the cycles spent in the DMA_WAIT (Fig. 9). It can be easily seen that the number of cycles spent is indeed very low; for example, QSDPCM application has only 14 954 cycles in CPU stalling during the DMA_WAIT, which is a very small percentage of the total cycles in the optimized application (<1%). If DMA wait cycles are significant, then it means that DMA/prefetch transfers are not hidden from the processors, and that they are scheduled with processing loops that take much fewer cycles compared to the DMA prefetch that is happening in parallel with them.

C. Prefetching on TI Board

In order to further validate our claims that prefetching provides significant benefits when it is done in a general framework of memory optimization, we have actually implemented a solution that MHLA has pinpointed as one of the optimum for the QSDPCM application, and uploaded the code to a TI C6201 development board that was at our disposal. This cumbersome procedure leads us to take real implemented performance measurements (Fig. 10), which again validate our claims. In addition, this task enabled us to see the different performance improvements that are achieved by using prefetch or DMA. Specifically, after performing some initial loop transformations according to

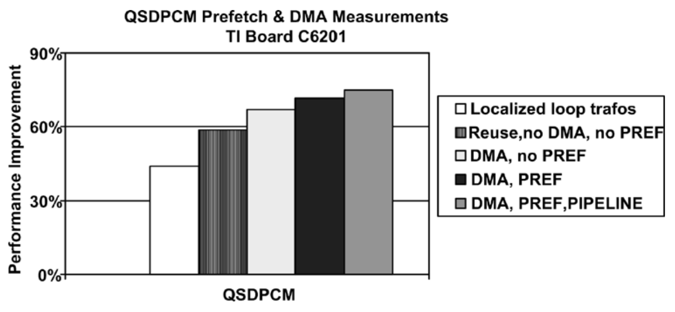


Fig. 10. Significant gains can be achieved using prefetching, DMA, and pipeline. On QSDPCM, gains up to 75% were measured on a TI C6201 development board.

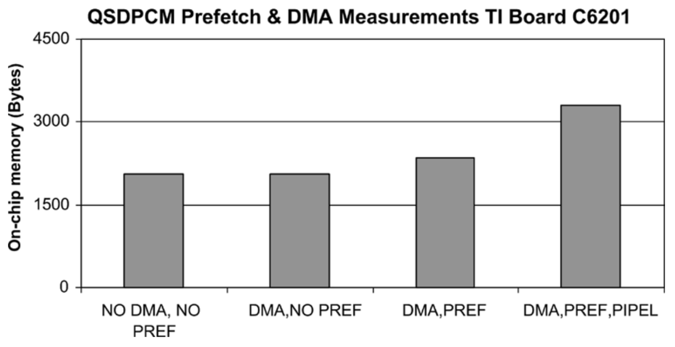


Fig. 11. Although prefetching and pipeline are beneficial to performance, they slightly increase the on-chip memory size. On QSDPCM, a performance boost of 75% is achieved by pipeline and prefetching combined with DMA, which increases the on-chip memory size by 25%.

the ATOMIUM [29] framework and introducing some copy candidates which were pinpointed by MHLA, we achieved a gain of 43.89% over the original (out of the box) code. Prefetching some copy candidates from off-chip to on-chip memory, allowed us to achieve an additional 15% performance increase. Transfers from off-chip to on-chip memory were CPU controlled (using "for-loops"). In the next step, we made all the transfers of the copy candidates to be DMA transfers, which boosted QSDPCM performance by 9%. We combined DMA and prefetch on the next step and this resulted in an additional 5% performance increase. Finally, in order to break some data dependencies, we used pipeline in some cases, which resulted in 75% performance improvement over the original out of the box code. Concerning the compilation process, we used the standard optimization flag−O2.

Using prefetching or pipelining prefetching has an impact on the on-chip memory size (Fig. 11). Assigning some arrays and CCs to on-chip layers, the on-chip memory size is 2048 (no prefetch or DMA). If DMA is used to speed up the transfers, on-chip memory remains the same but performance is improved (Fig. 10). Using prefetching, or in other words transferring off-chip data to on-chip data before they are requested, where data dependencies allowed it, increased the on-chip memory size to 2348 bytes. Finally, further increase in performance was obtained when loop pipeline was incorporated to break some dependencies, allowing us to prefetch large block transfers much earlier. This increased on-chip memory used in 3304 bytes, but allowed us to achieve a 75% performance boost.

All these tradeoffs are estimated in advance using the MHLA technique in a fast and accurate way. It is not necessary for the designer to implement his applications, which is very costly in terms of effort, in order to see all these different choices. MHLA finds all these different choices. Different choices lead to different performance, on-chip memory requirements and power consumption. Thus, the designer can study the global tradeoff curve, and decide the implementation that best fits his constraints in an early phase of his design.

The MHLA technique optimizes the performance and energy consumption of the memory architecture. In multimedia and image processing domains, the memory architecture is the dominant part in both energy consumption and performance [29]. Thus, the optimization of the memory architecture, in these applications, leads to optimization of the whole system. Concerning the performance optimization, this was verified by the TI Code Composer I.S.S. and by the TI board.

Our optimized allocation and assignment solutions, for the memory hierarchy, optimize the system concerning the energy consumption. Even though we do not have any quantitative way to prove this, we can support our claim as follows. First, in memory intensive applications, the energy consumption optimization of the dominant contributor (memory) is very likely to lead to overall optimization of the system. Second, TI has published an application report [43] that analyzes the power consumption on the TI devices and peripherals. This report shows that the C6201 core consumes 49% of total power consumption of the system for high- or low-activity models, while peripherals (such as the DMA controller) consume 1% of the total power consumption and, hence, have nearly no overhead when activated. This means that decreasing the execution cycles on the core of a given application utilizing the DMA mode has a strong direct impact on the overall power consumption. It is known that fewer cycles spent on an application execution, for the same energy cost/cycle, translates to fewer mW dissipated by the system. We are now unable to provide accurate numbers of the power reduction, due to the lack of information on the TI core breakdown, but it can be safely assumed that a performance increase of 45% leads to an energy decrease to implement the same task. Given these two factors, we believe that the power consumption of the whole system is reduced when prefetching with DMA is used.

In the case that hardware prefetchers are used in the specific applications, it is expected that the efficiency will be reduced because the hardware schemes perform only prefetching and not any algorithmic modifications. These applications have a lot of dependencies, which can only be broken using specific algorithmic transformations (i.e., loop pipeline), diluting the effectiveness of pure hardware prefetching. Without performing e.g., software pipelining, the elaborated hardware, such as the DMA controllers, or data movers, or hardware prefetchers, would have abysmal efficiency. The key contribution of this paper is that software prefetching, which is pipelined and performed using DMA, exhibits a significant performance and energy improvement. Of course, it is possible for someone with a good knowledge of his hardware prefetcher, to rewrite his code upfront to help the hardware prefetchers perform better. Though the effort that is required for this is great, hardware prefetching has

TABLE I
MHLA PREFETCHING IS MORE BENEFICIAL THAN PREFETCHING USING A HARDWARE CACHE

QSDPCM	Bytes	MHLA	d	a(dpf=1)	m(dpf=1)	m(dpf=4)	t(dpf=1)
QCIF	512	27,51%	10,81%	14,87%	12,40%	12,81%	12,69%
CIF	512	26,11%	14,74%	19,53%	17,26%	17,11%	17,42%
D1	512	26,54%	10,59%	14,41%	12,33%	12,79%	12,47%
QCIF	1024	5,54%	5,74%	7,80%	6,38%	6,39%	6,47%
CIF	1024	8,05%	10,92%	13,76%	12,22%	12,16%	12,25%
D1	1024	8,08%	7,31%	9,21%	8,28%	8,04%	8,31%
QCIF	2048	3,43%	3,06%	4,27%	3,42%	3,36%	3,47%
CIF	2048	3,09%	7,03%	9,61%	8,27%	8,13%	8,30%
D1	2048	2,12%	2,30%	2,92%	2,50%	2,48%	2,51%
QCIF	4096	1,22%	1,71%	2,54%	1,98%	1,91%	2,01%
CIF	4096	3,26%	6,03%	8,38%	7,20%	7,07%	7,22%
D1	4096	1,04%	1,28%	1,69%	1,44%	1,37%	1,44%
CAVITY							
640x400	512	10,28%	15,18%	32,14%	31,69%	31,69%	31,69%
800x600	512	10,25%	15,18%	32,14%	32,03%	32,03%	32,03%
640x400	1024	10,28%	14,73%	30,35%	31,13%	30,35%	30,13%
800x600	1024	10,25%	14,73%	30,36%	30,19%	30,24%	30,19%
640x400	2048	6,83%	14,51%	29,46%	29,35%	39,35%	29,35%
800x600	2048	10,25%	14,51%	29,46%	29,38%	29,41%	29,38%
640x400	4096	0,00%	14,40%	29,02%	28,96%	28,96%	28,96%
800x600	4096	6,82%	14,40%	29,02%	28,92%	28,92%	28,97%

the benefit of improving most of the applications without any further effort. If we start putting effort on algorithmic transformations, then we lose this significant benefit.

An application specific prefetching scheme is more beneficial than using a prefetching scheme with general applicability. We verified this assumption by running a series of tests using Dinero IV [44] cache simulator to analyze the execution traces of all the applications for various input sizes. Dinero can simulate various popular prefetching techniques, like always (A), miss (M), and tagged (T) for different prefetching distances (DPF). The "always prefetching" prefetches DPF blocks in every cache access (hit or miss), while the "miss prefetching" prefetches DPF blocks after a miss cache access and the "tagged prefetching" prefetches DPF blocks only after the first miss. In our measurements we also include the demand (D) fetch, which corresponds to the normal cache operation (without prefetch). In order to compare the cache and MHLA prefetching, we used the percentage of off-chip accesses with an on-chip memory (cache or scratchpad) versus the off-chip accesses without any on-chip memory. It can be easily concluded that the lower this percentage is, the better the energy and performance. Table I reports some of our measurements for different sizes of on-chip memory and input frames. In the column labeled MHLA, we inserted the estimations of the MHLA tool using prefetching, while in the following columns we put the Dinero Cache measurements for different cache parameters. Similar conclusions exist for all the other applications. These numbers reveal the following. 1) Hardware cache prefetching significantly increases the off-chip accesses, meaning that a large number of unnecessary off-chip accesses are being done. 2) For very small memory sizes (512 bytes), MHLA is not as efficient as a cache that does not use prefetching, which is expected, therefore, prefetching requires on-chip memory size. 3) The application specific prefetching is superior to these schemes of hardware prefetching which do not have a global view of the application. 4) For a given on-chip memory size the input dimensions do not significantly affect the gains. 5) There are cases where hardware prefetching is totally wrong (e.g., CAVITY), where we see that prefetching significantly increases the off-chip accesses (compared to the same cache parameters without prefetching). 6) Finally, these applications benefit more by using a simple cache, than by using a cache with prefetching. Concluding, we found out that application specific prefetching is superior to hardware prefetching, which is due to the fact that MHLA has a global view of the application, does not make unnecessary prefetches, and can fine tune the scheduling of the prefetches.

VII. CONCLUSION

The memory hierarchy plays an important role and should be used and exploited efficiently. One of the key obstacles in achieving high-memory performance utilization is memory-access latency. As a result, various techniques have been devised to hide memory access latency. This paper illustrates that the performance (and power consumption) of multimedia and image processing applications, which are characterized by a uniform access pattern, can be significantly improved by using the DMA mode, that most contemporary systems have, combined with the prefetch mechanism. We showed that prefetching should be done in a framework that optimizes memory hierarchy, memory allocation, and assignment. Nine real life applications are used as test vehicles for evaluation. The measurements show that it is possible to mask the prefetch latency, using the DMA mode, with a cost overhead of increased usage of on-chip memory space. This results in applications that perform faster and consume less energy to execute the same task. The technique presented here has been automated at IMEC in a prototype tool called MHLA, a part of the ATOMIUM framework.

References

[1] J. Fritts, "Multi-level memory prefetching for media and stream processors," in *Proc. Int. Conf. Multimedia Expo (ICME)*, 2002, pp. 101–104.

[2] T. A. Enger, "Paged control store prefech mechanism," *IBM Tech. Discl. Bull.*, vol. 7, no. 16, pp. 2140–2141, Dec. 1973.

[3] B. T. Bennet and P. A. Franaczek, "Cache memory with prefetching of data by priority," *IBM Technical Disclosure Bulleting*, vol. 18, no. 12, pp. 4231–4232, May 1976.

[4] R. L. Lee, P. C. Yew, and D. H. Lawrie, "Data prefetching in shared memory multiprocessors," in *Proc. Int. Conf. Parallel Process.*, 1987, pp. 28–31.

[5] A. J. Smith, "Sequential program prefetching in memory hierarchies," *IEEE Computer*, vol. 11, no. 12, pp. 7–21, Dec. 1978.

[6] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *Proc. Int. Symp. Comput. Arch.*, 1990, pp. 363–373.

[7] J.-L. Baer and T.-F. Chen, "An effective on-chip preloading scheme to reduce data access penalty," in *Proc. Supercomputing*, 1991, pp. 176–186.

[8] J. Fritts, "Multi-level memory prefetching for media and stream processors," in *Int. Conf. Multimedia Expo (ICME)*, 2002, pp. 101–104.

[9] R. Lysecky and F. Vahid, "Prefetching for improved bus wrapper performance in cores," *ACM Trans. Des. Automat. Electron. Syst.*, vol. 7, no. 1, pp. 58–90, Jan. 2002.

[10] R. Cucchiara, A. Prati, and M. Piccardi, "Improving data prefetching efficacy in multimedia applications," *Multimedia Tools Appl.*, vol. 20, no. 2, pp. 159–178, Jun. 2003.

[11] X. Zhuang and H.-H. S. Lee, "A hardware-based cache pollution filtering mechanism for aggressive prefetches," in *Proc. IEEE Int. Conf. Parallel Process. (ICPP)*, 2003, pp. 286–293.

[12] A. K. Porterfield, "Software methods for improvement of cache performance on supercomputer applications," Ph.D. dissertation, Rice University, Houston, TX, 1989, Tech. Rep. CRPC-TR89009.

[13] D. Callahan, K. Kennedy, and A. Porterfield, "Software prefetching," in *Proc. 4th Int. Conf. Arch. Support Prog. Lang. Oper. Syst. (ASPLOS)*, 1991, pp. 40–52.

[14] H. Glenn, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel, "The microarchitecture of the Pentium 4 processor," *Intel Technol. J.*, vol. Q1, pp. 1–10, 2001.

[15] K. Yeager, "The MIPS R10000 superscalar microprocessor," *IEEE Micro*, vol. 16, no. 2, pp. 28–40, Apr. 1996.

[16] V. Santhanam, E. H. Gornish, and W. C. Hsu, "Data prefetching on the HP PA-8000," in *Proc. 24th Int. Symp. Comput. Arch. (ISCA)*, 1997, pp. 264–273.

[17] T. Mowry, M. Lam, and A. Gupta, "Design and evaluation of a compiler algorithm for prefetching," in *Proc. ACM 5th Int. Conf. Arch. Support Program. Lang. Oper. Syst.* , 1992, pp. 62–73.

[18] T. Mowry and A. Gupta, "Tolerating latency through software-controlled data prefetching," *J. Parallel Distrib. Comput.*, vol. 12, no. 2, pp. 87–106, Jun. 1991.

[19] C.-K. Luk, "Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors," in *Proc. 28th Int. Conf. Comput. Arch.*, 2001, pp. 40–51.

[20] GNU Project, GNU Operating System-Free Software Foundation. [Online]. Available: http://www.gnu.org

[21] Intel, Intel Corp. [Online]. Available: http://www.intel.com, 2005

[22] E. H. Gornish and A. V. Veidenbaum, "An integrated hardware/software scheme for shared-memory multiprocessors," in *Proc. Int. Conf. Parallel Process.*, 1994, pp. 281–284.

[23] T. Chen, "An effective programmable prefetch engine for on-chip caches," in *Proc. 28th Int. Symp. Microarch.*, 1995, pp. 237–242.

[24] C. Xia and J. Torrellas, "Improving the data cache performance of multiprocessor operating systems," in *Proc. 2nd IEEE Symp. High-Performance Comput. Arch. (HPCA)*, 1996, pp. 85–94.

[25] D. Chiou, S. Devadas, J. Jacos, P. Jain, V. Lee, E. Peserico, P. Portante, L. Rudolph, G. E. Suh, and D. Willenson, "Scheduler-Based Prefetching for Multilevel Memories," Lab. Comput. Sci., MIT, Boston, MA, Group Memo 444, 2001.

[26] Z. Wang, D. Burger, K. S. McKinley, S. K. Reinhardt, and C. C. Weems, "Guided region prefetching: A cooperative hardware/software approach," in *Proc. 30th Ann. Int. Symp. Comput. Arch.*, 2003, pp. 388–400.

[27] H. Al-Sukhni, I. Bratt, and D. A. Connors, "Compiler-directed content-aware prefetching for dynamic data structures," in *Proc. 12th Int. Conf. Parallel Arch. Compilation Tech. (PACT)*, 2003, pp. 91–102.

[28] IMEC [Online]. Available: http://www.imec.be/design/atomium/, 2005

[29] F. Catthoor, S. Wuytack, E. D. Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle*, Custom Memory Management Methodology, Exploration of Memory Organization for Embedded Multimedia System Design*. Boston, MA: Kluwer, 1998.

[30] *"Texas Instrument Code Composer Studio Manuals,"* Texas Instruments, Dallas, TX, 1999.

[31] *"TMS320C620x/C670x DSP Program and Date Memory Controller, Direct Memory Access (DMA) Controller,"* ver. A, Texas Instruments, Dallas, TX, 2004.

[32] E. Brockmeyer, M. Miranda, H. Corporaal, and F. Catthoor, "Layer assignment techniques for low energy in multi-layered memory organizations," *Proc. Des. Automat. Test Eur.*, pp. 1070–1075, 2003.

[33] I. Issenin, E. Brockmeyer, M. Miranda, and N. Dutt, "Data reuse analysis technique for software-controlled memory hierarchies," in *Proc. Des. Automat. Test Eur.*, 2004, pp. 202–207.

[34] S. Wuytack, F. Catthoor, G. D. Jong, and H. J. D. Man, "Minimizing the required memory bandwidth in VLSI system realizations," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 7, no. 4, pp. 433–441, Dec. 1999.

[35] P. Shivakumar and N. Jouppi, "CACTI 3.0: An Integrated Cache Timing, Power and Area Model," COMPAQ, Palo Alto, CA, WRL Res. Rep. 2001/2, 2001.

[36] V. Bhaskaran and K. Konstantinides*, Image and Video Compression Standards*. Norwell, MA: Kluwer, 1998.

[37] M. Nam, J.-S. Kim, R.-H. Park, and Y. S. Shim, "A fast hierarchical motion vector estimation algorithm using mean pyramid," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 5, no. 4, pp. 344–351, Aug. 1995.

[38] T. Zahariadis and D. Kalivas, "A spiral search algorithm for fast estimation of block motion vectors," in *Proc. 8th Euro. Signal Process. Conf. EUSIPCO*, 1996, pp. 1079–1082.

[39] P. Strobach, "QSDPCM—A new technique in scene adaptive coding," in *Proc. 4th Eur. Signal Process. Conf. (EUSIPCO)*, 1988, pp. 1141–1144.

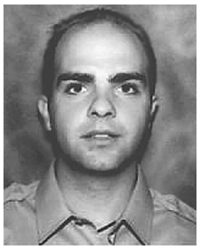
[40] K. Danckaert, F. Catthoor, and H. D. Man, "Platform independent data transfer and storage exploration illustrated on a parallel cavity detection algorithm," in *Proc. ACM Conf. Parrallel Distrib. Process. Tech. Appl.*, 1999, pp. 1669–1675.

[41] G. Lafruit, L. Nachtergaele, B. Vahnhoof, and F. Catthoor, "The local wavelet transform: A memory-efficient, high-speed architecture optimized to a region-oriented zero-tree coder," *Integr. Comput.-Aided Eng.*, vol. 7, no. 2, pp. 89–103, 2000.

[42] M. Proesmans, L. V. Gool, and A. Ossterlinkck, "One shot active 3D shape reconstruction," in *Proc. 13th Int. Conf. Pattern Recognit.: Appl. Robot. Syst. (ICPR)*, 1996, pp. 336–340.

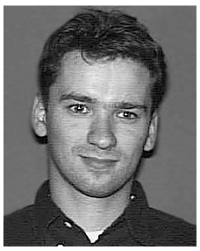
[43] "Power Consumption Summary," Texas Instruments, Dallas, TX, SPRA486C, 2002.

[44] J. Edler and M. Hill, Dinero IV Trace-Driven Uniprocessor Cache Simulator (1997) [Online]. Available: http://www.cs.wisc.edu/markhill/DineroIV/



Minas Dasygenis (M'06) was born in Thessaloniki, Greece, in 1976. He received the diploma in electrical and computer engineering and the Ph.D. degree from the Democritus University of Thrace, Greece, in 1999 and 2005, respectively. His diploma thesis was honored by The Technical Chamber of Greece and Ericsson Hellas.

He has published more than 20 papers in international journals and conferences and he has been a principal researcher in three European research projects. His research interests include low-power VLSI design of arithmetic circuits, residue number system, embedded system design, DSPs, hardware/software codesign, and IT security.

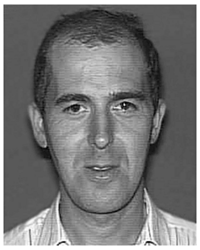


Erik Brockmeyer received the M.S. degree in electrical engineering from the University of Eindhoven, Eindhoven, the Netherlands, in 1998. He completed his M.S. thesis on MPEG-4 in the System Exploration for Memory and Power (SEMP) group. This group is part of the DESICS Division of the Inter-University Micro Electronics Center (IMEC), Heverlee, Belgium.

He has worked on automation of various steps of the Data Transfer and Storage Exploration (DTSE) script. His past work focuses on mapping an application efficiently to the multi levels of a memory hierarchy. Currently, his work is shifting towards multi processor systems with a shared distributed memory subsystem.

Bart Durinck received the M.Sc. degree in computer sciences from the Univerteit Gent, Heverlee, Belgium, in 1997.

In 1997, he joined ICOS Vision Systems, Belgium, where his main task was the development of DSP image processing and real-time system software. In 2002, he joined the Inter-University Micro-Electronics Center (IMEC), Heverlee, Belgium, where his focus was on the development of prototype design tool for memory hierarchy aware optimizations.



Francky Catthoor (F'05) received the M.Eng. and Ph.D. degrees in electrical engineering from the Katholieke Universiteit Leuven, Leuven, Belgium, in 1982 and 1987 respectively.

In 1983–1987, he became a Researcher in the area of VLSI design methodologies for Digital Signal Processing, along with Prof. H. De Man and Prof. J. Vandewalle, as a Ph.D. Thesis Advisor. Since 1987, he has headed several research domains in the area of high-level and system synthesis techniques and architectural methodologies, all within the Design Technology for Integrated Information and Telecom Systems (DESICS-formerly VSDM) Division at the Inter-University Micro-Electronics Center (IMEC), Heverlee, Belgium. In 1989, he became an Assistant Professor in the Electrical Engineering Department at the Katholieke Universiteit Leuven, Heverlee, Belgium, and became a Full Professor (part-time) in 2000. His current research activities are in the field of architecture design methods and system-level exploration for power and memory footprint within real-time constraints, oriented toward data storage management, global data transfer optimization and concurrency exploitation. His major target application domains are real-time signal and data processing algorithms in image, video, and end-user telecom applications, and data-structure-dominated modules in telecom networks. Platforms that contain both customized architectures (potentially on an underlying configurable technology) and (parallel) programmable instruction-set processors are targeted. Also deep-submicron technology issues are included.

Dr. Cathoor was an Associate Editor for the IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS from 1995 to 1998, and for the IEEE TRANSACTIONS ON MULTIMEDIA from 1999 to 2001. In 2002, he was an Associate Editor for the ACM TRANSACTIONS ON DESIGN AUTOMATION FOR EMBEDDED SYSTEMS (TODAES) and in 1996 he became an Editor for *Kluwer's Journal of VLSI Signal Processing*. In 1997, he became a Member of the Steering Board for the VLSI Technical Committee of the IEEE Circuits and Systems Society. In 1999 he served on the IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS steering board. He was the Program Chair in 1997 and the General Chair in 1998 for the IEEE *International Symposium on System Synthesis* (ISSS). He was also the Program Chair and Main Organizer of the 2001 IEEE *Signal Processing Systems* (SIPS) *Conference*. He received the Young Scientist Award from the Marconi International Fellowship Council in 1986. He is a Fellow at the Inter-University Micro-Electronics Center (IMEC), Heverlee, Belgium.

Dimitrios Soudris (M'92) received the diploma and Ph.D. degrees in electrical engineering from the University of Patras, Achaea, Greece, in 1987 and 1992, respectively.

He is currently an Assistant Professor in the Department of Electrical and Computer Engineering at the Democritus University of Thrace, Xanthi, Greece. He was leader and principal investigator in numerous research projects funded from the Greek Government and Industry, as well as the European Commission (ESPRIT II–III–IV and fifth and sixth IST). He has served as General Chair and Program Chair for the International Workshop on Power and Timing Modeling, Optimization, and Simulation (PATMOS). His research interests include low-power design, parallel architectures, embedded systems design, and VLSI signal processing. He has published more than 140 papers in international journals and conferences.

Dr. Soudris received an award from INTEL and IBM for the project results of LPGD #25256 (ESPRIT IV). He is a member the VLSI Systems and Applications Technical Committee of IEEE CAS and the ACM.

Antonios Thanailakis was born in Greece on August 5, 1940. He received the B.Sc. degree in physics and electrical engineering from the University of Thessaloniki, Thessaloniki, Greece, in 1964 and 1968, respectively, and the M.Sc. and Ph.D. degrees in electrical engineering and electronics from the University of Manchester (UMIST), Manchester, U.K., in 1968 and 1971, respectively.

He has been a Professor of Microelectronics in the Department of Electrical and Computer Engineering, Democritus University of Thrace, Xanthi, Greece, since 1977. He has been active in electronic device and VLSI system design research since 1968. He was leader for carrying out research and development projects funded by Greece, EU, or other organizations on various topics of Microelectronics and VLSI Systems Design (e.g., NATO, ESPRIT, ACTS, STRIDE). His current research activities include microelectronic devices and VLSI systems design. He has published a great number of scientific and technical papers, as well as five textbooks.