Low-power Data Memory Communication for Application-Specific Embedded Processors[*]

Peter Petrov and Alex Orailoglu

Computer Science & Engineering Department
University of California, San Diego
(ppetrov,alex)@cs.ucsd.edu

ABSTRACT

We propose a novel customization methodology for power reduction on the communication link between an embedded processor and its data memory. We target the address bus and show how by utilizing application information about the memory references in the data intensive program loops, a power efficient address communication protocol can be established between the processor core and the data memory. The data memory controller thus generates the addresses for the various data streams with minimal run-time information from the processor engine, achieving significant power reductions on the address bus. An efficient reprogrammable hardware support is presented for enabling the proposed methodology. The experimental results demonstrate the efficacy of the approach for a set of data intensive applications.

Categories and Subject Descriptors

B.3 [**Hardware**]: Memory structures; B.4 [**Hardware**]: Input/Output and Data Communications; C.1 [**Computer Systems Organization**]: Processor Architectures; C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems

General Terms

Algorithms, Design, Experimentation, Performance

1. INTRODUCTION

Significant advances in VLSI process technology have made the utilization of system-on-a-chip design approaches highly attractive. Cost-efficient products, easy design reuse, and flexible implementation constitute some of the many SOC advantages. Embedded processor cores are being utilized widely in such systems in order to achieve better time-to-market, lower design cost, and easily reprogrammable implementations. However, the increased silicon integration, together with the ever increasing clock frequencies, leads to proportional increases in terms of power consumption.

At the same time, energy dissipation is becoming a prominent characteristic for a large number of important applications, such as hand-held and wireless devices. Less energy dissipation leads not only to longer battery life, but also enables larger die sizes. Consequently, techniques for minimizing system power consumption are of paramount importance for achieving high product quality. These techniques can be applied on various design abstraction levels, from circuit level to system architecture.

In a typical DSP or data intensive embedded environment, the interaction between the processor core and the on-chip/off-chip data memory or cache can be significant. Typically, in such environments the local working set is stored in a fast local scratch-pad memory, while a large off-chip data memory is used to bring the data to be processed from the system's environment. In the case of VLIW or SIMD processor engines with multiple load/store units, several data accesses are typically performed within a cycle. Consequently, a significant amount of the system power is consumed in the interaction between processor and data memory subsystem.

Hardware/software co-design techniques play a significant role in building complex SOCs comprising processor cores and dedicated ASIC modules. Processor cores are still the most viable solution in implementing these complex SOCs if special care is taken to obviate the fundamental disadvantages of reduced performance and high power consumption stemming from their general-purpose nature. Customizing the processor microarchitecture to particular application-specific needs has been shown to be an efficient technique for boosting processor performance and significantly reducing its power consumption [1].

In this paper, we propose an application-specific customization methodology for power reduction in the processor's communication to its data memory. The communication between processor and memory consists of the following operations. A processor load/store unit generates an effective address. This address is sent to the memory subsystem through the address bus along with certain control information. In the case of a store instruction, the data that needs to be stored is sent along to the memory on the data bus. If the transfer is a load, the memory subsystem reads in the data being referenced by the address and returns it to the processor through the data bus. It is well known that transferring addresses and data on long interconnect busses consumes a significant amount of power, due to the high capacitance of the lines [2].

We present a technique for minimizing the traffic on the address bus to the data memory subsystem. Application-specific information is statically extracted and dynamically transferred to the data memory controller prior to the execution of a major loop. Consequently, only an insignificant amount of information is transferred from the processor core to the data memory controller instead of a complete effective address, typically 32 bits long. The inherent reprogrammability of the proposed approach, extends its applicability to practically all areas of data-intensive embedded applications.

[*]This work is supported by NSF Grant 0082325.

2. RELATED WORK

Power optimization techniques at the circuit-level have been the dominant approach in designing energy efficient systems so far [3, 4]. However, in recent years, architecture-level approaches have attained popularity due to their ability to eliminate redundancies on a higher, microarchitectural level, thus resulting in even larger power optimizations [5].

The problem of minimizing the number of transitions on communication busses within a microprocessor-based system has been attacked recently by a number of research groups. The *Bus-Invert* method has been proposed in [6]. In this approach, the bus content is inverted if this leads to a smaller Hamming distance compared to the previous value on the bus. An additional bus signal informs the receiver whether the bus content is inverted or not. The approach is applicable to any communication bus, but its generality prevents it from achieving significant improvements on busses with highly correlated data. The *T0* approach [7] introduces an additional line to the instruction memory address bus in order to exploit the typical sequentiality of the instruction addresses. When this line is asserted, the memory controller computes the new address by incrementing the previous one. In [8] this technique is extended and the requirement for an additional redundant control line eliminated. The low-power encoding proposed in [9] utilizes self-organizing lists in order to achieve an optimal encoding for the most frequently accessed addresses. By utilizing a rather complex hardware implementation of self-organizing lists, the approach exploits the temporal and spatial locality of the addresses on both instruction and data address busses.

3. MOTIVATION

The recent research efforts for reducing bus switching activity in general-purpose processors [6, 8] have largely aimed at reducing the traffic on the address bus to the instruction memory as the typical incrementality of the instruction reference stream, violated only in the cases of control instructions, makes it highly amenable to power optimizations. While the address bus to the instruction memory is important in terms of power consumption, the data memory address bus plays a similar role in importance and contribution to the total system power. Nonetheless, no significant results have been achieved in minimizing the address traffic to the data memory subsystem, which in some cases, most notably for VLIW or vector processors, can even exceed in amount the traffic to the instruction memory. The typically limited knowledge regarding the application and its data memory reference patterns in general purpose processor architectures may account for the dearth of research efforts in this direction. Even in the case when some application information can be gathered, such as in the case of a typical DSP or data intensive application, where loop accesses through a number of arrays in quite a regular fashion following constant strides predominate, the end result on the data memory address bus consists of an intertwined, seemingly irregular mix, of these addresses.

It is the thesis of this paper that the utilization of application-specific knowledge within both the memory controller and the processor core can nonetheless result in quite efficient techniques for minimizing the amount of information that needs to be transferred from the processor core to the data memory subsystem. The research challenge resides in identifying effective means of communicating memory requests, utilizing application knowledge.

In typical data intensive applications, such as DSP or numerical analysis algorithms, the program execution is concentrated within a small number of heavily executed loops. These application loops frequently contain multiple nesting levels, reflecting the dimension of the problem being solved. The data being processed resides in large array structures and is commonly accessed using affine index calculations [10]. In order to eliminate or minimize to a large extent the amount of information that needs to be sent on the address bus to the data memory, application knowledge about the strides of the memory accesses as well as the execution order of these accesses is required on the data memory side. Knowledge of address strides for each load/store instruction and the relative order of execution within the data memory controller can help autonomously generate the next effective address with *no need of an effective address* sent by the processor, simply by adding the corresponding stride and processing the load/store request. Let's consider, for example, the following loop.

```
for i=1 to 100
  for j=1 to 100
    A[2i,j]=B[i,2j+5]+C[i+1,3j+1];
  endfor
endfor
```

The references to B and C are loads from the data memory, while the reference to A is a store. Nonetheless, all three references need to send an effective address to the data memory in order to perform their memory operation. Each of these three references constitutes a series of regular accesses with a constant stride. In the inner loop, for instance, the reference $A[2i, j]$ generates a sequence of incremental effective addresses, while the references to $B[i, 2j + 5]$ and $C[i + 1, 3j + 1]$ generate streams with strides 2 and 3, respectively. Executing the next iteration of the outer loop seemingly introduces a discontinuity to this pattern; nonetheless, it is easy to notice that fundamentally the regularity is not violated, but needs to be examined a bit more subtly by taking into consideration the two-level structure of the loop. The steady state loop execution strides as well as the outer loop execution strides in any case are fixed and their values can be easily computed during compile time by analyzing the index expression; in neither case do the multiplicative constants before either subscript effect the computation method, except for altering the constant value of the relevant stride. Consequently, this stride regularity can be easily identified and exploited by simply identifying the direction of the corresponding loop branches. Autonomous generation of these addresses necessitates transmission to the data memory controller of information regarding these strides and the point at which the outer loops are executed. While the effective addresses of these three memory accesses are intertwined on the address bus to the data memory, the order of their appearence is fixed and regular throughout the loop execution, ensuring their easy identification and communication.

Given the application knowledge about these references, i.e., their execution order and address strides, an enhanced data memory controller can utilize this additional information in order to generate the needed addresses internally, obviating completely the processor core responsibility to provide these addresses to the data memory subsystem. The only limited amount of information that the processor needs to send is the type of memory request, and an indication as to whether the innermost loop is in its steady state of execution or an outer loop is to be executed, so that the memory controller can access which set of strides to use for computing the next effective address. Therefore, by utilizing application information within the data memory controller, an efficient address communication scheme can be established, such that the memory controller is able to generate the appropriate effective addresses by itself by utilizing statically obtained information about the load/store order and their strides across the loop dimensions.

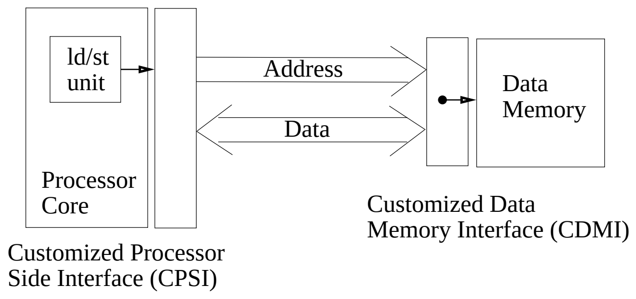
ld/st unit
Address
Data
Data Memory
Processor Core
Customized Data Memory Interface (CDMI)
Customized Processor Side Interface (CPSI)

Figure 1: General Architecture

4. CUSTOMIZED COMMUNICATION

The proposed methodology essentially transfers application information regarding the load/store instructions to the data memory controller, so that it can generate effective addresses by itself, thus obviating the need for receiving these addresses from the processor core. The general architecture of the proposed approach is depicted in Figure 1. The programmable interface associated to the data memory, the CDMI, is responsible for generating the addresses of the load/store instructions executed within the application loop. In the case of complete knowledge of the memory reference execution order and their strides, the traffic on the address bus is completely eliminated. The application information is provided to the CPSI and CDMI, as shown in Figure 1, by software before starting the loop execution.

If the CDMI is aware of the load/store execution order, no information whatsoever is needed from the processor to identify the memory reference. Knowing the starting memory reference, the CDMI can generate its effective address by adding its stride and automatically updating its state to the next memory operation for the application loop, thus obviating the need to send an address for the next instance of that memory operation.

Attaining such implicit communication of the required data addresses necessitates knowledge of the appropriate strides and of the initial values of the effective addresses for all the load/store instructions. These can be determined at compile time and either stored a priori or sent to the CDMI before entering the loop.[1] A table, traversed sequentially, contains the current effective addresses for the memory reference instructions, while the strides are stored in similar tables. The number of entries corresponds to the total number of memory reference instructions that can be handled by the proposed approach. If any load/stores remain uncovered, they can be executed in their usual way by sending the complete address to the data memory. Each time a load/store instruction is executed, the CPSI sends the stride index and asserts a special additional line denoted as NA for *not an address*, associated to the address bus, which signals the CDMI that the data memory request received does not contain an effective address. A fixed number of least significant address bus lines hold the *stride index*, an index that points to the appropriate stride array corresponding to the loop nesting level. Subsequently, the CDMI utilizes the information about this particular load/store and computes its effective address by simply adding its current address and stride. The address computed thus is used for accessing the data memory, while the CDMI stores it for utilization in the next loop iteration.

For the proposed customization methodology, we target data intensive loops with load/store effective addresses computed as affine linear functions of the loop indices. During compile-time, the strides for the load/store intructions are computed for subsequent run-time utilization within the CDMI. The compile analysis includes examination of the affine index expressions and identifying the strides for all the loop dimensions. The next code fragment shows an example of such an affine memory reference in its most general form.

```
for i1=k1 to n1, step s1
  for i2=k2 to n2, step s2
    for i3=k3 to n3, step s3
      A[a1*i1+b1,a2*i2+b2,a3*i3+b3]
      ...
    endfor
  endfor
endfor
```

The loop has three levels, with indices i_1, i_2, and i_3. The effective address EA of the reference from this code is computed as $EA = c_1 i_1 + c_2 i_2 + c_3 i_3 + c_4$, where the coefficients c_j depend on the array index and loop parameters a_l, b_l, s_l, n_l, and k_l, $l \in \{1, 2, 3\}$. Given this general representation, it is evident that the effective address changes with a fixed stride across the loop iterations. The stride can be statically computed in the case of fixed loop boundaries, or dynamically by software before entering the loop. Notably, the stride for the first iteration of the innermost loop differs from that of subsequent loop iterations. We denote the strides as L_3 for the steady case, while L_2 and L_1 correspond to the case of the first iteration of the i_3 loop after incrementing i_2 or i_1, respectively.

$$L_3 = c_3 s_3$$
$$L_2 = c_2 s_2 + c_3(k_3 - n_3)$$
$$L_1 = c_1 s_1 + c_2(k_2 - n_2) + c_3(k_3 - n_3)$$

Consequently, if the CDMI contains these stride values and is aware of which load/store instruction is currently requesting/writing a data, no address communication is needed from the processor. Since the memory controller has no information regarding loop execution, the processor needs to specify which stride is to be used for the address calculation.

The expressions for the strides L_j do not have to be recomputed during the loop iterations. Instead, the CDMI stores their values in a small SRAM array and utilizes them dynamically. The processor core, therefore, needs to send, through the CPSI, only a very short index to specify which stride array is to be used. If the architecture supports up to four levels of loop nesting, for example, then two bits would suffice. Noteworthy is that since the stride for the innermost loop dimension is used repetitively, while the other strides are used only in the beginning of the innermost loop, the number of bit transitions is zero in the steady case, and a quite small number of bit transitions on each start of loop nest i_3, constituting practically a zero overhead operation.

5. HARDWARE IMPLEMENTATION

5.1 Data memory side support

Figure 2 represents the architecture of the CDMI module. The Load/Store Table (LST) is used to store the current addresses for the memory reference instructions. An entry in the LST corresponds to a load/store instruction and contains its current address. The LST is addressed by the *LST Index (LI)* register. Initially, its value is reset to 0, so that it refers to the first entry in the LST, where the address of the first load/store instruction to be executed in the loop is stored. The strides for the memory references are stored

[1] In the case of variable loop boundaries, the strides need to be computed dynamically before entering the loop nest and subsequently sent to the CDMI.

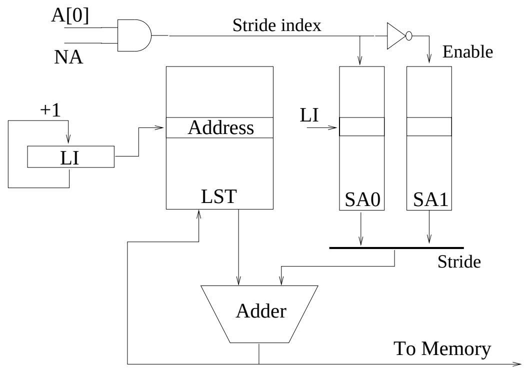

Figure 2: CDMI Architecture

in the *Stride Arrays (SA)*. For simplicity, figure 2 shows a CDMI architecture that supports only two strides (i.e., two loop nesting levels). In practice, the number of stride arrays can be increased by inserting additional stride arrays and controlling them with the stride information received by the CPSI with no practical impact on circuit hardware complexity or power consumption. After reading in the current address from the LST, the stride value is added to it and the resulting address is sent to the data memory for completing the access, while at the same time the LST entry is updated with the newly computed address. Subsequently, the LI is incremented to point to the next entry in the LST, which corresponds to the next load/store instruction to be executed by the application loop. When the LI reaches the index of the last load/store for the loop, it needs to reset its value so as to point to the first memory reference for the next loop iteration. This is easily achieved by an additional bit associated to the LST entries, which specifies whether the current entry corresponds to the last memory reference instruction for the loop. This simple policy of updating the LI register allows us to target all the memory reference instructions within the most frequently executed innermost loop with minimal hardware overhead. However, if load/stores outside the innermost level are to be targeted as well, then an additional modification to the LI register update logic is needed. The load/stores from the outer loop iterations need thereupon to be stored in the beginning of the LST. The starting indices in the LST for the load/stores of the outer loop nests need to be kept in separate registers for subsequent utilization. When the LI index reaches the entry corresponding to the last reference of the innermost loop, the LI register is updated to point to the beginning of the corresponding loop nest level depending on the stride index received from the CPSI. This is, of course, possible because the stride index completely identifies the loop nest level.

The proposed hardware support is extremely efficient in terms of area overhead and power consumption. The LST and SA tables are very small SRAM arrays (or alternatively can be implemented as small latch groups if deemed more power efficient) with depth of at most 32. The number of their entries corresponds to the total number of load/store instructions within the loop that can be targeted by the approach we propose; typically the number of static memory reference instructions within the application tight loops falls well within the bound of 32.

5.2 Processor side support

The CPSI hardware implementation is even simpler. Its fundamental goal is to send to the CDMI block the stride index. Since the stride index depends on whether any of the outer loop iterations have moved forward, the CPSI needs to obtain this information dynamically from the loop defining instructions. In the case of no special loop support in the instruction set, conditional branches are used to form the loop. In this case, the CPSI needs to monitor the direction of these branches to decide whether the stride index has changed. The following simple code example illustrates the structure of a three level loop nest.

```
L1:
L2:
L3:
  // Innermost loop code
branch L3  // B3
branch L2  // B2
branch L1  // B1
```

A taken B3 branch indicates an innermost loop in a steady state. If the branch B2 is encountered and taken, then the L_2 stride index needs to be sent to the CDMI, while if B1 is taken, then the stride in effect is L_1. Consequently, the processor side interface needs to detect these situations and act accordingly. The branches need to be identified and their direction observed. This can be easily implemented as three comparators (for supporting three level loop nests) which compare the content of three registers containing the branches' PC to the PC of the currently executing branch instruction. Their content is initialized before entering the loop together with setting up the CDMI module.[2]

Having a special zero-overhead loop instruction is a very common situation in almost every modern embedded processor. Typically, these instructions allocate a special counter register which is initialized with the number of loop iterations and decremented at each loop iteration. In this case, the CPSI logic can be implemented in a straightforward fashion by associating also the stride index value to the loop counter. Each time a loop counter is decremented, the corresponding stride index is to be utilized.

The methodology that we propose does not have to target the complete set of load/store instructions within a loop. In the case of a normal load/store instruction, the *Not an Address (NA)* line is deasserted and the complete effective address is sent to the data memory instead. Therefore, the CPSI needs to be able to distinguish between load/stores that are targeted by our approach and the rest. The most efficient way to perform this is to introduce an additional bit to the opcode of the load/store instructions that would specify whether this instruction is a special case being handled through low-power data address communication. For the instructions covered by our scheme, no effective address needs to be computed and no additional information, such as base register identifier or immediate offset values, needs to be carried with that opcode. This new opcode signals to the CPSI to assert the additional *NA* line on the address bus and to start the memory request as soon as the load/store instruction is decoded.

6. UNPREDICTABLE EXECUTION ORDER

Many application loops from the DSP and numerical analysis domain, such as *FFT*, *FIR*, and *convolution,* contain no control altering instructions other than the loop support code, resulting in the execution order of the load/store instructions being fixed throughout for these types of loops. The low-power data memory communication technique presented so far is directly applicable to such loops. If, however, the application loop contains control altering instructions, such as branches, then the CDMI needs additional information in order to identify the memory reference to be executed subsequently. A simple example is shown below.

[2]Since the frequency of execution of each loop can be easily identified, it can be exploited to devise an alternative scheme by storing the relevant frequencies of execution and counting up against them.

```
v1=A[i,j];
if (c) then
 v2=B[i+1,j];
else
 v2=B[i,j+1];
endif
```

In this code fragment, the load subsequent to $A[i, j]$ might be either $B[i + 1, j]$ or $B[i, j + 1]$, depending on the branch predicate. Therefore, the branch direction determines the order in which the memory references are executed. Consequently, the CDMI is unable in this case to follow this order autonomously with no information from the CPSI. The CPSI needs to send information to the CDMI regarding the identity of the load/store. The load/store identification information can be simply an index that will be used by the CDMI to access the tables with the application information. Consequently, instead of allowing the CDMI to update its state to the next load/store instruction by itself, this state can be explicitly specified by the processor side. In terms of the hardware specified in figure 2, this corresponds to just sending the LI value to the CDMI, instead of the interface updating this register by itself. For applications that target loops with control altering instructions, the CPSI needs to send the LI value for the load/store instructions, resulting in an expansion of the information to be communicated.

The consequent predicated execution of load/store instructions wreaks havoc with the necessary updating of the effective address stored in the LST to ensure its correspondance to the particular loop instance. The validity of the address information in the CDMI can be guaranteed through a special logic that ensures that all memory reference instructions handled by the CDMI are updated. Given the fact that the CDMI table has very few entries, assigning a "dirty" bit and sequentially updating the entries can be achieved by a simple and power efficient hardware.[3]

6.1 Hardware support for loops with control

As described above, in the case of a loop with control altering instructions, the CPSI needs to additionally transfer the index/name of the load/store instruction to the CDMI. This index is directly used by the CDMI to address the small LST table. Since the size of the LST is in the order of 16 to 32 bits, an additional 4 to 5 bits from the address bus need to be utilized. As described in the previous subsection, the new load/store instruction opcode does not need to carry information about its effective address. Therefore, the LI value for each instruction can be easily accommodated within the instruction opcode. Subsequently, when the instruction is decoded, the CPSI will send the LI value together with the stride index.

While at first glance this might seem to impose a non-trivial overhead compared to the case of the data-only loops, it delivers an interesting degree of freedom regarding the placement of instructions within the LST that can be exploited to obtain a highly power efficient encoding. Since the value of the LST index LI transmitted to the data memory is of *no* importance, as long as this index is the same for both CPSI and CDMI hence enabling the correct identification amongst individual memory access instructions, an efficient power encoding that utilizes the most frequent order of memory reference instruction executions can be employed. *Gray* code is a perfect candidate for such an encoding. In order to find the most frequent paths through the loop's *control-flow graph (CFG)*, the application under consideration is profiled. After that a Gray code can be utilized for the most frequently utilized load/store sequence. If the control structure of the loop is due to checking for some boundary/error cases throughout the application loop, then in practice the traffic of LI to the CDMI will add only a single bus line transition for each outer loop level iteration.

[3]As the CDMI needs to be able to update all the LST entries during the loop execution, special consideration needs to be paid to loop instances that can possibly execute in fewer cycles than the cycles needed for the LST update logic to accomplish its task. This special and very rare case can be easily identified at compile time through shortest path identification applied to the control flow graph and handled by reducing the number of load/stores in the LST.

The CDMI architecture for this case is very similar to the one shown in Figure 2 with the following new features. The index to address the LST in this case needs to be able to use the LI value sent by the processor side instead of the value of the LI register. The LST entries are complemented with an additional "dirty" bit, which specifies whether the entry contains the updated value of its current effective address. When an entry is read, if its "dirty" bit is clear, then the value of the entry is directly used to access the data memory; otherwise, the stride is added to this address and the newly computed address is used to complete the memory access.

A special hardware is needed for updating all the LST entries. It consists of a 4 to 5 bit counter that generates sequential indices to the LST, and hardware for updating the entries by adding the corresponding stride and for resetting the "dirty" bit. The update process is started at the beginning of the loop and all the "dirty" bits set at the end of the loop. In order to allow this update logic to work simultaneously with the memory access requests, an additional read line is needed from the memory structures (while the only write line is used by the update logic). Given the extremely small size of these tables (16 to 32 entries at maximum), the consequent impact on power and hardware complexity is extremely small.

7. TIMING AND POWER IMPLICATIONS

As processor performance strongly depends on the speed of interaction to the data memory, it is essential to ensure that the proposed solution does not adversely effect memory access time. It can be shown that no such adverse effect exists by noting that the CDMI has 2 cycles to perform its operations due to the fact that the effective address computation cycle, which was initially performed in the processor core, is now shifted to the data memory controller. Noteworthy is that adding the stride to the current address does not constitute an additional operation and consequently does not deteriorate the net power savings from the eliminated address transfers. Our approach simply performs this computation on the data memory side, disabling its execution in the processor core. Any code whose sole purpose is the computation of the load/store addresses can be eliminated from the application loop, furthermore.

In a typical pipelined embedded processor implementation, the load/store instruction execution is split into two cycles. Initially the effective address is computed and subsequently, a request is sent to the data memory. In the case of a load/store instruction utilizing the CPSI and CDMI blocks for low-power address transfers, the request to the memory subsystem can be sent at the beginning of the first stage, while the data is effectively returned at the end of the subsequent pipeline stage. Consequently, the transactions to the data memory are effectively pipelined and the address calculation logic in the CDMI adds no delay to the memory access time.

8. METHODOLOGY UTILIZATION FLOW

The proposed methodology utilizes information regarding the load/store instructions within the application loops and the way they generate effective addresses. Consequently, the methodology is based on: extensive compile/link time analysis for extracting this information; a reprogrammable hardware support for runtime utilization of the application information; and a way to transfer this information to the processor and data memory hardware support.

The first step in the methodology is profiling the application and identifying the major program loops. The load/store instructions within these loops are identified and their index functions analyzed and the strides extracted. In the case of a loop with control code, the load/store identification indices are determined according to their most expected execution order through the utilization of Gray encoding. Special code for initializing the LST and the Stride Arrays within the CDMI and the comparator registers of CPSI in the case of no special loop instruction support is inserted before the program loops. At run-time, just prior to entering the loop, this code prepares the hardware support for the customized data memory address transfer. The size of this code is relatively small and is executed only once before starting the application loop. Consequently, its overhead, both in code size and execution cycles, is negligible.

An important characteristic of the proposed customization hardware support is its inherent reprogrammability. Consequently, the methodology we propose is applicable across a significant range of applications and does not require spinning of new silicon in the case of functionality changes or late specification modifications.

9. EXPERIMENTAL RESULTS

In our experimental studies we have measured the effectiveness of the proposed approach by observing the reduction of the transitions on the address bus to the data memory. We have utilized seven benchmarks. The first five are important numerical and DSP kernels: *Matrix multiplication (mmul)* of matrices with size 256x256; *successive over-relaxation (sor)* [11] on a matrix with size 256x256; *extrapolated Jacobi-iterative method (ej)* [12] on a 256x256 grid; *fast discrete cosine transform (fdct)* kernel on a block of data with 8 samples; and the *energy loop in fast fourier transform (efft)* working on a block of 512 samples. The last two benchmarks are the speech coding (*adpcm_enc*) and decoding (*adpcm_dec*) applications [13], selected for their frequent utilization in many voice processing embedded applications. The first four applications consist of a single, data-only loop nest, while the *efft*, *adpcm_enc*, and *adpcm_dec* utilize a loop with a control structure. Consequently, for the *mmul*, *sor*, *ej*, and *fdct*, the basic CDMI and CPSI functionality for data-only loops has been utilized. The only information sent on the address bus lines to the CDMI consists of the stride index. In the case of the *adpcm* and *efft* benchmarks, a 4 bit LST index has been additionally provided to the CDMI, effectively utilizing the hardware support for loops with control structure.

In order to measure the number of transitions on the address bus to the data memory, we instrumented the application source code with logic that observes the address stream to the data memory and accumulates the Hamming distance between the consecutive addresses. The same instrumentation code was utilized to capture the number of the bus line transitions in the case of utilizing the proposed methodology. The instrumented application was compiled and executed on a PC workstation with Linux OS. The results obtained are reported in Figure 3.

The second column (#TR) in Figure 3 reports the total number of transitions in millions on the address bus to the data memory in the general case. It is notable that *mmul* generates, due to its higher execution complexity, a significantly larger number of transitions compared to the rest of the benchmarks. The next column (#TR Opt) of the table represents the minimized number of transitions after utilizing the methodology we propose. The fourth column (#Ref) shows the total number of memory reference instructions residing in the loop nest and targeted by the proposed methodology. Finally, the last column shows the reduction in percentage of the total number of transitions on the address bus. For the first four benchmarks containing data-only loops, the reductions consistently exceed 99%; this extremely high level of reductions has been anticipated, given the fact that only the two bit stride indices are sent and changed only on the outer loop iterations. Since the *efft* and *adpcm* benchmarks contain a loop with control altering instructions, additional traffic of the LST indices had to be introduced. Three out of all four references in *adpcm* were control independent for both the encoder and the decoder, while for *efft* three out of all eight references were subject to a predicate resolution. A straightforward Gray code encoding was utilized for these benchmarks and was subsequently considered in measuring the optimized number of bus line transitions.

	#TR*10^6	#TR Opt	#Ref.	Reduction(%)
mmul	203.06	261,633	3	99.88
sor	1.56	1,020	5	99.94
ej	46.91	15,360	11	99.97
fdct	0.041	256	6	99.37
efft	0.01	2,039	8	78.67
adpcm_enc	3.37	738,868	4	78.05
adpcm_dec	3.26	589,932	4	81.88

Figure 3: Data memory address bus transitions

10. CONCLUSION

In this paper we have presented a power optimization methodology that targets the address communication between an embedded processor core and its data memory. The number of transitions is greatly reduced by utilizing application-specific information regarding the order and address strides of the memory reference instructions within the major application loops. We have shown how by judiciously utilizing this information, the data memory controller can autonomously generate the effective address for most of the data references, thus obviating the need for address communication. The conducted experimental results confirm the expected significant reductions in terms of total number of power consuming transitions on the data memory address bus. The hardware microarchitecture is inherently reprogrammable, thus enabling the utilization of the proposed methodology for a wide range of applications.

11. REFERENCES

[1] P. Petrov and A. Orailoglu, “Performance and power effectiveness in embedded processors - Customizable Partitioned Caches”, *IEEE TCAD*, vol. 20, n. 11, pp. 1309–1318, November 2001.

[2] S. Ramprasad and N. R. Shanbhag, “A coding framework for low-power address and data busses”, *IEEE TVLSI*, vol. 7, pp. 212–221, June 1999.

[3] M. B. Kamble and K. Ghose, “Analytical energy dissipation models for low-power caches”, in *ISLPED*, pp. 143–148, August 1997.

[4] K. Ghose and M. B. Kamble, “Reducing power in superscalar processor caches using subbanking, multiple line buffers and bit-line segmentation”, in *ISLPED*, pp. 70–75, August 1999.

[5] N. Bellas, I. Hajj and C. Polychronopoulos, “Using dynamic cache management techniques to reduce energy in a high-performance processor”, in *ISLPED*, pp. 64–69, August 1999.

[6] M. R. Stan and W. P. Burleson, “Bus-invert coding for low-power I/O”, *IEEE TVLSI*, vol. 3, pp. 49–58, March 1995.

[7] L. Benini et al., “Asymptotic zero-transition activity encoding for address busses in low-power microprocessor-based systems”, in *7th GLSVLSI*, pp. 77–82, March 1997.

[8] Y. Aghaghiri, F. Fallah and M. Pedram, “Irredundant address bus encoding for low-power”, in *ISLPED*, pp. 182–187, August 2001.

[9] M. Mamidipaka, D. Hirschberg and N. Dutt, “Low power address encoding using self-organizing lists”, in *ISLPED*, pp. 188–193, August 2001.

[10] M. S. Lam, E. E. Rothberg and M. E. Wolf, “The Cache Performance and Optimizations of Blocked Algorithms”, in *4th ASPLOS*, pp. 63–74, April 1991.

[11] M. E. Wolf and M. S. Lam, “A Data Locality Optimizing Algorithm”, in *PLDI*, pp. 30–44, June 1991.

[12] S. Nakamura, *Applied Numerical Methods with Software*, Prentice-Hall, Englewood Cliffs, N.J., 1991.

[13] C. Lee, M. Potkonjak and W. H. Mangione-Smith, “MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems”, in *30th MICRO*, pp. 330–335, December 1997.