Multi-Level On-Chip Memory Hierarchy Design for Embedded Chip Multiprocessors

ABSTRACT

Two trends, increasing importance of memory systems and increasing use of chip multiprocessing, motivate conducting research on memory hierarchy optimization for chip multiprocessors. One of the interesting research topics along this direction is to design an application-specific, customized, software-managed on-chip memory hierarchy for a chip multiprocessor. Another important issue is to optimize the application code and data for such a customized on-chip memory hierarchy. This paper proposes an integer linear programming (ILP) based solution to the combined problem of memory hierarchy design and data allocation in the context of embedded chip multiprocessors. The proposed solution uses compiler analysis to extract data access patterns of parallel processors and employs integer linear programming for determining optimal on-chip memory partitioning across processors and data allocations across memory components. Our experimental results show that the application-specific on-chip memory hierarchies designed using this approach are much more energy efficient than conventional (pure shared or pure private) on-chip memories and those designed by a prior work that partitions memory space across parallel processors without designing a multi-level hierarchy.

1. Introduction and Motivation

One of the critical components of an embedded computing system is its memory architecture. There are several reasons for this. First, many embedded applications are data intensive and make frequent memory references. As a result, a significant portion of execution cycles is spent in memory accesses. Second, this large number of accesses also contribute to a large fraction of overall power consumption [5]. Third, since on-chip memory structures constitute a significant portion of overall chip area in embedded designs, they are highly vulnerable to transient errors, making them an important target for reliability optimizations. Finally, many security leaks are exploited through manipulation of memory space. Based on these observations, there have been numerous proposals in the last two decades for optimizing memory behavior of embedded systems and applications. Note that, as embedded applications become more complex and process increasingly large data sets, the role of the memory system will be even more important in the future.

Chip multiprocessing is becoming a promising way of utilizing ever-increasing number of transistors in computer architectures. There are at least three trends that motivate for chip multiprocessing in embedded computing domain. First, computer architects are finding it increasingly difficult to wring more performance out of single-processor based approaches. Second, multi-core architectures promise to simplify the increasingly complex design and validation work of processors. This in turn lowers chip costs and reduces time-to-market. Third, many embedded applications from image/video processing domain lend themselves very well to parallel computing. Therefore, they can take advantage of parallel processing capabilities provided by chip multiprocessors.

These two trends, growing importance of memories and increasing use for chip multiprocessing, form a strong motivation for conducting research on memory synthesis and optimization for chip multiprocessors. One of the interesting research topics along this direction is to design an application-specific, software-managed, on-chip memory hierarchy. Another important issue is to optimize the application code and data for such a customized on-chip memory hierarchy. This paper proposes a solution to the combined problem of energy-efficient on-chip memory hierarchy design and data allocation in the context of embedded chip multiprocessors.

The proposed solution targets array-based embedded applications from the domain of image/video processing. It is based on integer linear programming (ILP) and thus determines the optimal on-chip memory hierarchy for a given application, under the assumed cost model. It needs to be noted that the optimality of a memory system can be defined based on the goal of the optimization. In this paper, our goal is to design an software-managed on-chip memory hierarchy for chip multiprocessors and allocate data across the components of this hierarchy in such a manner that the energy consumption on the memory system (during execution of the application) is minimized. The proposed solution is very general and can come up with any on-chip memory hierarchy. In the most general case, the resulting memory system will have both private components (i.e., memories that are exclusively accessed by a single processor) and shared components (i.e., memories that are shared by multiple processors). Also, in the most general case, each processor can see a different memory hierarchy (in terms of both the number of the components in the hierarchy and their sizes). Figure 1 illustrates four different example on-chip memory designs. (a) and (b) correspond to conventional pure shared memory based and pure private memory based designs, respectively. (d) is a typical hierarchy design returned by our ILP-based approach proposed in this paper. (c) is a design that would be returned by an approach that partitions the on-chip memory space across a single level (not a hierarchy).

We implemented the proposed approach using a commercial linear solver and performed experiments using ten embedded benchmark codes. The experimental results show that the proposed ILP-based approach (1) comes up with on-chip memory hierarchies that are much better (in terms of energy efficiency) than conventional memory designs based on pure shared memory or pure private memories, and (2) performs better than customized memory architectures that perform optimal memory space partitioning across processors without designing a hierarchy. The results also indicate that, for the least energy consumption, memory space partitioning and data allocation should be handled at concert. We also found that the solution times taken by our ILP-based approach are not excessive and are within tolerable limits.

This paper is organized as follows. A discussion of the related work is given in Section 2. The details of our ILP-based approach are given in Section 3, and an experimental evaluation is presented in Section 4. The paper is concluded in Section 5.

2. Related Work

Memory system design and optimization has been a popular area of research for the last two decades. We can roughly divide the work in this category into two classes: memory synthesis/design and software optimizations. Most of the prior work on memory synthesis/design focus on single processor based systems [19, 1, 20, 3, 15, 17, 13, 5]. In comparison, Abraham and Mahlke focus on an embedded system consisting of a VLIW processor, instruction cache, data cache, and second-level unified cache. A hierarchical approach to partitioning the system into its constituent components and evaluating each component individually is utilized. Meftali et al [11] focus on the memory space allocation (partitioning) problem, which they formulate based on an integer linear programming model. Their solution permits one to obtain an optimal distributed shared memory architecture, minimizing the global cost to access the shared data in the application and the memory cost. The effectiveness of the proposed approach is demonstrated by a packet routing switch example. Gharsalli et al [6] present a new methodology for embedded memory design for application-specific multiprocessor system-on-chips. Their approach facilitates the integration of standard memory components. Further, the concept of memory wrapper they introduce allows automatic adaptation of physical memory interfaces to a communication network that may have a different number of access ports. The main difference between our work and these prior efforts is that our work is very general. Although we focus on application-specific memory design, the proposed solution strategy (based on ILP) is applicable to any array-based embedded application. In addition, we also address the data allocation problem along with the memory space partitioning problem. [10] proposes a dynamic memory management scheme for chip multiprocessors. Our

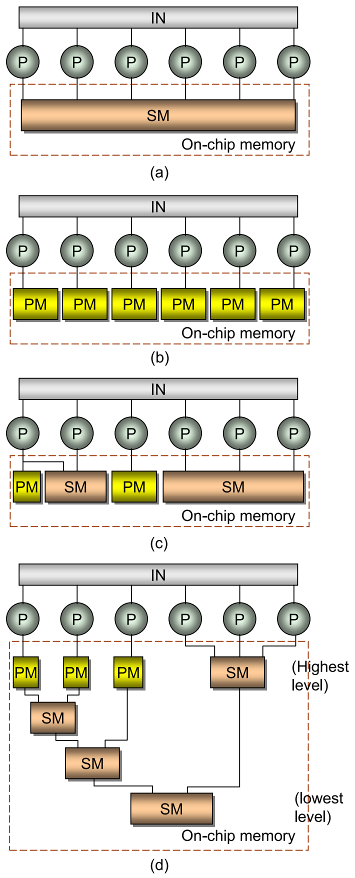

Figure 1: Different on-chip memory designs. (a) Pure shared memory architecture. (b) Pure private memory architecture. (c) Single level based design. (d) Hierarchy based design. SM = shared memory; PM = private memory; IN = interconnection network.

work is also more general than the one in [10], as the latter does not design a hierarchy but only partitions the available memory space across multiple processors.

On the software side, the prior work considered the optimal use of the available SPM space. Panda et al [14] present an elegant static data partitioning scheme for efficient utilization of scratch-pad memory. Their approach is oriented towards eliminating the potential conflict misses due to limited associativity of on-chip cache. This approach benefits applications with a number of small (and highly reused) arrays that can fit in the SPM. Benini et al [2] discuss an effective memory management scheme that is based on keeping the most frequently used data items in a software-managed memory (instead of a conventional cache). Kandemir et al [8] propose a dynamic SPM management scheme for data accesses. Their framework uses both loop and data transformations to maximize the reuse of data elements stored in the SPM. They enhance their approach in [9]. Cooper and Harvey [4] present two compiler-directed methods for software-managing a small cache for holding spilled register values. Hallnor and Reinhardt [7] propose a new software-managed cache architecture and a new data replacement algorithm. The main difference between these studies and our work is that, in addition to data allocation, we also focus on customized memory hierarchy design.

3. Problem Formulation

3.1 Operation of On-Chip Memory Hierarchy and Memory Management Granularity

The on-chip memory hierarchy returned by our approach operates as follows. When a processor executes a memory operation, the memory component that holds the corresponding data is accessed (and its access latency and energy consumption is incurred). Recall that the on-chip memory space in our framework is software managed. If the required data is not located in a memory component connected to the requester processor and it resides in a remote (but on-chip) memory component, it is accessed from there (this typically incurs a higher energy consumption than a local access). If this also fails, the data is accessed from the off-chip memory.

Table 1: The constant terms used in our ILP formulation. These are either architecture specific or program specific.

Constant	Definition
N_P	Number of processors
N_{MC}	Number of possible memory components
N_L	Number of levels in the memory hierarchy
S_M	Total on-chip memory size
N_B	Number of data blocks
S_B	Size of a data block
N_{Ph}	Number of program phases
$L_{size}(i)$	A function that returns the maximum possible size for i^{th} level

In our approach, the data transfers across the memory components take place at phase (epoch) boundaries. Specifically, the program is divided into phases and profiled. The profile data gives us the data blocks (considering all array data in the application) accessed by each processor at each phase, and an estimation of the number of accesses to each data block by processors. This information is then passed to the ILP solver (which will be discussed in detail shortly), which in turn determines the locations of data blocks for each phase and optimal on-chip memory hierarchy. After that, this information is passed to the compiler which modifies the application code accordingly to insert explicit data transfer calls. In this paper, our main focus is on ILP formulation of the problem and an experimental evaluation of the resulting framework. We do not focus on the details of the code modification phase, which is rather straightforward. Also, we do not address the problem of optimal phase partitioning or data block size selection.

3.2 ILP Formulation

ILP provides a set of techniques that solve those optimization problems in which both the objective function and constraints are linear functions and the solution variables are restricted to be integers. The 0-1 ILP is an ILP problem in which each (solution) variable is restricted to be either 0 or 1 [12]. It is used in this paper for determining the multi-level on-chip memory hierarchy for a chip multiprocessor architecture and data allocation. Table 1 gives the constant terms used in our ILP formulation. We used *Xpress-MP* [21], a commercial solver, to formulate and solve our ILP problem, though its choice is orthogonal to the focus of this paper. The goal of our ILP formulation is to minimize the energy consumption in the memory subsystem of a chip multiprocessor. Specifically, our solution decides how on-chip memory components are being shared among processors, what the size of each memory component is, and what level in the hierarchy each memory component is located.

Our objective is to minimize the energy consumption within the memory subsystem, and this can be achieved by dividing the available memory space in such a way that each processor can access the most frequently used data (by itself) from the closest memory location of the right size. By doing so, we can reduce the energy cost of accessing lower levels of the memory or accessing a remote memory component. There are $N_{MC} = 2^{N_P} - 1$ memory components possible for any level in the hierarchy, each of which can be accessed by different set of processors (N_P is the number of processors). For example, if there are two processors, there are possibly three on-chip memory components. These are m_1 (accessed by only processor 1), m_2 (accessed by only processor 2), and m_3 (accessed by both processors 1 and 2). Note that, we are not restricted to have only one copy of each memory component type[1], that is it is possible to have the same type of memory component at different levels of the memory hierarchy, except that these same memory component types at different levels should get larger as we move from the higher level to the lower level. This follows

[1] A "type" in this context corresponds to a memory component accessed by a specific set of processors.

from the fact that having a smaller or equal size memory component at a lower level in a hierarchy does not provide any energy benefits.

We assume for now that the entire data manipulated by the application can be stored in the on-chip memory hierarchy (we relax the assumption in Section *3.2.1*). We use an access matrix A as an input to hold the number of accesses made by each processor to each data block. Mathematically, we define:

- $A_{p,ph,b}$: the number of accesses to data block b by processor p during program phase ph[2].

Our approach uses 0-1 variables to specify the size of each on-chip memory component (C) possible. Specifically,

- $C_{l,u,m}$: indicates whether memory component u at level l is of size m.

We use M to identify the location of a data block during the course of execution. More specifically,

- $M_{ph,b,l,u,m}$: indicates whether data block b at program phase ph is located in memory component u of size m at level l.

Note that both $C_{l,u,m}$ and $M_{ph,b,l,u,m}$ are 0-1 variables whose values we want to determine. E is used to capture the energy consumption of a processor at a specific program phase. That is,

- $E_{p,ph}$: captures the energy consumed by processor p during program phase ph.

Since the available on-chip memory space is limited, the following constraint must be satisfied:

$$\sum_{i=1}^{N_L} \sum_{j=1}^{N_{MC}} \sum_{k=0}^{S_M} k \times C_{i,j,k} = S_M. \tag{1}$$

In the above expression, memory components are allowed to have sizes of 0, which indicates that these memory components do not actually exist (in the final hierarchy designed).

The next constraint indicates that a memory component can have one and only one size:

$$\sum_{i=0}^{S_M} C_{l,u,i} = 1, \quad \forall l, u. \tag{2}$$

As we go to the higher levels in the hierarchy, the sizes of the memory components decrease due to space and cost issues. We use the following expression to set a maximum capacity limit for each level in the on-chip memory hierarchy:

$$\sum_{i=1}^{N_{MC}} \sum_{j=0}^{S_M} j \times C_{l,i,j} \leq L_{size}(l), \quad \forall l. \tag{3}$$

Although it is possible to come up with different functions to obtain the level size constraint, in our current implementation, we use the following expression for $L_{size}(l)$:

$$L_{size}(l) = \lceil (S_M \times l) / \sum_{i=1}^{N_L} i \rceil. \tag{4}$$

A lower level memory component connected to a higher level memory component should be at least accessible by the processors of the higher level memory component. That is, the connections between higher and lower levels of memories should be in such a way that as the level of the memory component decreases, the number of processors that can access it should increase. For example, if a memory component at level 3 is connected to processors 1 and 2, it should not be connected to a memory component only connected to processor 1 at level 2. We use the term $subset(m_1, m_2)$ to indicate that the processors accessing memory component m_1 is a subset of the processors accessing memory component m_2. This is a matrix given as an input to the ILP solver, which represents all possible memory component pairs in mathematical terms.

$$C_{l,u,m} \leq \sum_{i=1}^{N_{MC}} \sum_{j=m}^{S_M} C_{l,i,j}, \quad \forall l, u, m : l \leq N_L - 1 \wedge subset(u, i). \tag{5}$$

As it is stated earlier, it is possible that a memory component type (for example a memory component accessed by processors 1 and 2) can appear at different levels in the hierarchy. In this case, the size of the memory component at a higher level should be less than the one at the lower level. This can be captured as follows:

$$C_{l_1,u,m_1} \leq 1 - C_{l_2,u,m_2}, \quad \forall l_1, l_2, u, m_1, m_2 : m_1 \geq m_2 \wedge l_1 \geq l_2 + 1. \tag{6}$$

Having defined the necessary constraints for the memory components in the hierarchy, we can now give the constraints for locating the data blocks within the memory components. We start by observing that a data block should be mapped to a single memory component. This is expressed as:

$$\sum_{i=1}^{N_L} \sum_{j=1}^{N_{MC}} \sum_{k=0}^{S_M} M_{ph,b,i,j,k} = 1, \quad \forall ph, b \tag{7}$$

If a memory component is used for a data block, that memory component should exist in the final hierarchy. Consequently, we have:

$$C_{l,u,m} \geq M_{ph,b,l,u,m}, \quad \forall ph, b, l, u, m. \tag{8}$$

The data stored in a memory component should be less than the size of the memory component. In mathematical terms, this can be captured as:

$$\sum_{i=1}^{N_B} M_{ph,b,l,u,m} \times S_B \leq m, \quad \forall ph, b, l, u, m. \tag{9}$$

Energy consumption of processor p at program phase ph can be calculated by summing up the energy consumptions due to data accesses performed by p:

$$E_{p,ph} = \sum_{i=1}^{N_B} \sum_{j=1}^{N_L} \sum_{k=1}^{N_{MC}} \sum_{l=0}^{S_M} M_{ph,i,j,k,l} \times A_{p,ph,i} \times AE(local(p,k), j, l, ports(k)) \quad \forall p, ph. \tag{10}$$

In the above expression, $A_{p,ph,i}$ returns the number of data accesses to data block i by processor p during program phase ph. There are four parameters that affect the energy consumption. These are data locality, the level of the memory component, the size of the memory component, and the number of ports the memory component has. In addition to accessing the non-local data (i.e., a data in a remote on-chip memory component), accessing a lower level memory component will require a higher interconnect energy consumption. Similarly the size of the memory component being accessed is another key parameter in energy consumption. As the number of processors sharing the memory component increase, the required number of ports will increase. This in turn will increase the energy consumption. We use function $AE(location, level, size, ports)$ to capture the energy consumption of a specific access. In the above expression, locality is denoted by using $local(p, k)$ which returns 1 if data block k is local to processor p; that is, it is stored in one of the memory components connected to processor p.

Having specified the necessary constraints in our ILP formulation, we next give our objective function:

$$\min \quad \sum_{i=1}^{N_P} \sum_{j=1}^{N_{Ph}} E_{i,j}. \tag{11}$$

Since $E_{i,j}$ denotes the energy consumption of processor i at program phase j, the total energy consumed by all processors throughout the entire program execution (i.e., over all program phases) can be calculated by summing them up.

[2]This matrix is filled after profiling and analyzing the application code.

Table 2: The number of data block accesses made by different processors.

Processor	D_1	D_2	D_3	D_4	D_5	D_6	D_7
Phase 1							
1	0	0	1	16	2	2	3
2	9	6	3	0	2	1	2
3	8	6	1	0	3	2	2
Phase 2							
1	12	0	2	0	1	3	2
2	0	7	1	8	2	2	3
3	0	7	3	9	2	1	2

3.2.1 *Incorporating Off-Chip Memory*

Recall that so far we assumed the entire data manipulated by the application can be stored in on-chip memory hierarchy. If we are to relax this assumption by allowing a smaller on-chip memory space than the total data size, then off-chip memory accesses have to be included in the total energy consumption expression. In order to do that, first, Expression (7) has to be replaced with:

$$\sum_{i=1}^{N_L}\sum_{j=1}^{N_{MC}}\sum_{k=0}^{S_M} M_{ph,b,i,j,k} \le 1, \quad \forall ph, b \tag{12}$$

In the above expression, a data block is ensured to reside in at most one on-chip memory component. That is, it is possible that a data block is not in one of the on-chip memory components. In addition to this modification, the energy consumed during off-chip memory accesses needs to be calculated. $O_{ph,b}$ in the expression below captures whether a data block is in the off-chip memory or not:

$$O_{ph,b} \ge 1 - \sum_{i=1}^{N_L}\sum_{j=1}^{N_{MC}}\sum_{k=0}^{S_M} M_{ph,b,i,j,k} \quad \forall ph, b. \tag{13}$$

The energy consumption due to off-chip accesses can then be calculated as follows:

$$OE_{p,ph} = \sum_{i=1}^{N_B} O_{ph,b} \times A_{p,ph,i} \times AE_{off} \quad \forall p, ph. \tag{14}$$

In this expression, AE_{off} denotes the energy consumption of a single off-chip data block access. Based on these modifications, the objective function given in Expression (11) is replaced with:

$$\min \quad \sum_{i=1}^{N_P}\sum_{j=1}^{N_{Ph}} (E_{i,j} + OE_{i,j}). \tag{15}$$

3.2.2 *Example*

We now give an example showing how our approach works in practice. We assume that there are 3 processors and 2 program phases with 7 data blocks, each with a size of 1K. The total on-chip memory space is assumed to be 7K with 2 levels. Table 2 shows the number of data block accesses made by different processors at each of the two phases.

In Figure 2 the memory hierarchy generated by our approach is shown. The data allocation for phase 1 is shown on the left-hand-side and that for phase 2 is shown on the right-hand-side. As can be seen from this figure, data block D_4 is privately used by processor P_1 during phase 1, whereas it is shared by processors P_2 and P_3 during phase 2. Since data blocks D_3, D_5, D_6 and D_7 are used by all the processors during both the phases, they are placed into a shared memory component residing at level 2. Note that these four data blocks are used by all the processors but not as frequently as the other data blocks.

4. Experimental Evaluation

Our goal in this section is two-fold. First, we want to show that it is possible to apply our ILP-based approach to realistic embedded application codes. For this purpose, we test our approach using ten real-world application codes extracted from the domain of embedded image/video processing. Our second goal is to compare our approach to alternate on-chip memory design schemes and check whether it really makes sense to employ this approach. Such an evaluation is important given that the ILP solvers usually take more time than fast heuristic solutions.

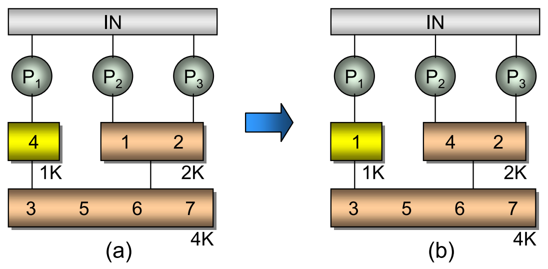

Figure 2: An example on-chip memory hierarchy design and data allocation. (a) After phase 1 and data allocation. (b) After phase 2. The numbers within the memory components are data block ids.

Table 3: Application codes and their important properties.

Benchmark	**Brief Description**	**Number of C Lines**	**Number of Memory References (M)**	**Memory Energy Consumption (mJ) Pure Shared**	**Memory Energy Consumption (mJ) Pure Private**
Compress	Digital Image Compression	127	625.2	196.2	183.1
Conv-Spa	Convolution in Spatial Domain	231	811.4	113.4	109.6
Filter	Triangular Filter	270	118.6	106.7	115.7
Laplace	Morphological Laplacian	438	109.6	310.3	301.0
LU-Decomp	LU Decomposition	86	227.3	256.7	280.3
Minkowski	Image Dilation and Erosion	455	839.5	576.8	602.2
Seg-02	Image Segmentation	622	107.1	605.2	584.7
Text	Texture Analyzer	1018	144.9	899.3	813.9
Img-Trans	Image Transformation	972	222.9	1142.8	1228.5
Verifier	Image Verifier	1353	196.3	986.7	976.9

4.1 Setup

Table 3 gives the important properties of the ten application codes used in this study. These benchmark codes are collected from several research groups who work on embedded image/video application development. The second column of this table gives a description of each benchmark and the next one shows the number of source code lines (all these codes are array-based and are written in the C language). The fourth column gives the number of memory references made by each application. Note that this number is an inherent characteristic of an application and is independent of the particular memory hierarchy used during execution. The next two columns give the memory energy consumption (i.e., the energy consumption due to memory references including both on-chip components and off-chip memory) for the pure private memory based and pure shared memory based on-chip memory architectures (see Figure 1) under the default (base) simulation parameters given in Table 4.

To conduct our experiments, we modified the Simics [18] tool-set (running on Solaris 9 operating system) and simulated the different schemes. The default configuration parameters are given in Table 4, and these are the values that are used unless explicitly stated/varied in the sensitivity experiments. The energy consumption values for the different memory components (i.e., the AE values used in our formulation) are obtained using the CACTI tool [16] under the 0.07μ process technology. The two most important factors that affect the energy consumption of a memory components are the size (capacity) and the number of ports. We modeled using CACTI memory components of different capacities and ports. A few necessary changes were needed to be made to the basic CACTI model to obtain the energy model for software-managed memory (as against conventional caches).

For each of the ten applications shown in Table 3, we performed experiments with five different schemes. Pure Shared (PS) and Pure Private (PP) are the two conventional schemes. In PS, there is a single monolithic on-chip memory space shared by all the processors. In PP, each processor has its private on-chip memory, i.e., the total on-chip memory space is divided equally among the parallel proces-

Table 4: Important base simulation parameters used in our experiments.

Parameter	Default Value
Number of Processors	8
Total On-Chip Memory Space	2MB
Data Block Size	128 bytes
Off-Chip Memory Capacity	32MB
Off-Chip Memory Access Latency	100 cycles
Bus Arbitration Delay	5 cycles
Replacement Policy	LRU
Maximum Number of Levels in Hierarchy	5

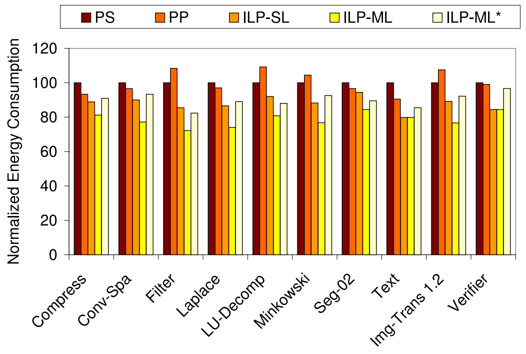

Figure 3: Normalized energy consumptions.

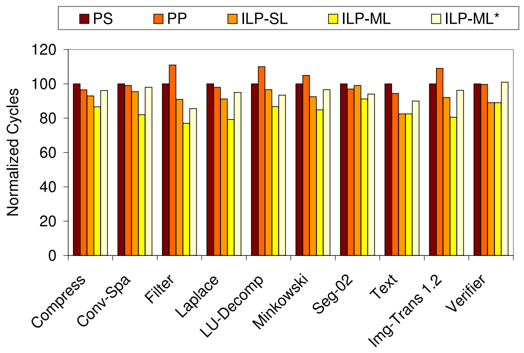

Figure 4: Normalized execution cycles.

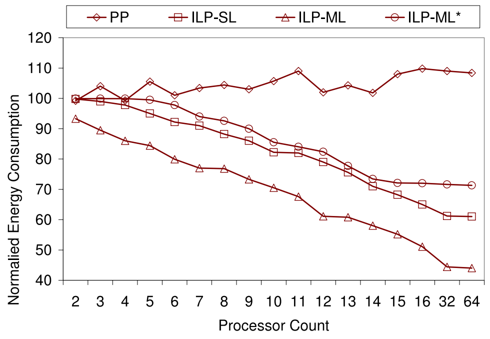

Figure 5: Impact of processor count.

sors. ILP-ML represents the ILP-based multi-level on-chip memory hierarchy design and data allocation strategy discussed in this paper. ILP-SL is similar to ILP-ML; the difference is that it uses only a single level (i.e., the available on-chip memory space is partitioned across the processors optimally without having a hierarchy, as exemplified in Figure 1(c)). It is important to emphasize that all these versions use optimized data allocation across memory components, which is determined using integer linear programming. Our goal in making experiments with the ILP-SL version is to isolate the energy benefits that are coming explicitly from memory hierarchy design. Our next scheme, ILP-ML* is similar to ILP-ML, except that it does not perform optimal data allocation. Instead, it allocates data based on the following heuristic. The program is profiled and the frequently accessed data by each processor is placed into the memory components which is directly accessible by that processor. The idea is to minimize the energy consumption spent in accessing frequently used data. The reason why we make experiments with this version is to isolate the benefits that are coming from optimal data allocation; that is, to demonstrate that optimal memory hierarchy design alone is not sufficient for achieving maximum energy reductions, and a unified approach that combines data allocation and memory hierarchy design is necessary.

Before presenting our energy savings, we want to remark on our ILP solution times. The largest solution time we experienced during our experiments was 5.5 minutes (for application Verifier). Therefore, since memory hierarchy design is an offline process, we can say that our solution times are not excessive.

4.2 Base Results

We start by presenting the energy and performance results with the schemes explained above. Figure 3 gives the energy consumption values with different schemes. All energy results are normalized with respect to the fifth column of Table 3. Our first observation from these results is that there is no clear winner between the PS and PP schemes. The relative performance of one scheme to another depends mostly on the amount of data shared by parallel processors and how these data are shared. When we look at the remaining three versions, we see that ILP-ML generates the best results for all the applications tested. The average energy reduction it brings is 21.3%. In comparison, the average energy reductions with the ILP-SL and ILP-ML* schemes are 12.1% and 9.9%, respectively. This result clearly emphasizes the importance of exploiting multiple levels (in designing memory hierarchy) and of optimal data allocation. In particular, when we compare the ILP-SL and ILP-ML schemes, we see that they generate the same results in only two benchmarks (Text and Verifier). On the other hand, the rest of the applications in our experimental suite take advantage of multiple levels. The reason why the ILP-ML* scheme performs poorly as compared to ILP-ML is lack of optimal data allocation. This means that optimal memory space partitioning and optimal data allocation should be carried out together if we are to minimize energy consumption.

While our memory partitioning and data allocation approach is designed for minimizing energy consumption rather than optimizing performance, it is also important to consider its performance. This is because the resulting memory design may not be acceptable if it causes a significant increase in original execution cycles (as compared to conventional memory designs such as PP or PS), even though it could reduce memory energy consumption significantly. The normalized execution cycles achieved by the different schemes are shown in Figure 4. While these results are not as good as those shown in Figure 3 from the perspective of our ILP-based approach, we still observe important benefits. In fact, the ILP-ML scheme brings nearly 15% improvement in original execution cycles over the pure shared memory based scheme. We also need to mention that, to reduce execution cycles even further, the ILP-based approach can be targeted, with some modifications, at minimizing execution cycles as well (instead of minimizing energy consumption), though this option is not evaluated in this paper. Still, when we consider the results shown in Figures 3 and 4 together, we see that the ILP-ML scheme is very useful from both performance and energy perspectives.

4.3 Sensitivity Analysis

In this section, our goal is to vary some of the default (base) values of the simulation parameters (shown in Table 4), and check how doing so affects our energy savings. We present our results for a single application only (Minkowski), due to space concerns. The first parameter we change is the number of processors (processor count), and the results are given in Figure 5. We see that all three ILP-based schemes evaluated take advantage of increased number of processors. The main reason for this behavior is the fact that, when the number of processor is increased, careful partitioning of on-chip memory space becomes more important, and all three ILP-based schemes allow us to achieve that to different extents.

The second parameter whose value we vary is the total on-chip memory size. The results are given in Figure 6. M/4 on the x-axis

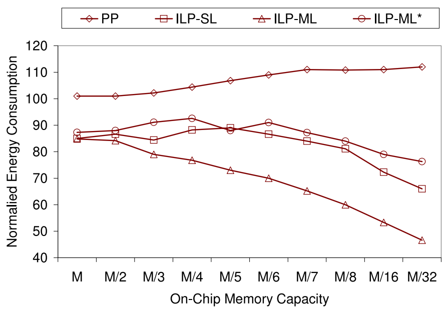

Figure 6: Impact of on-chip memory capacity.

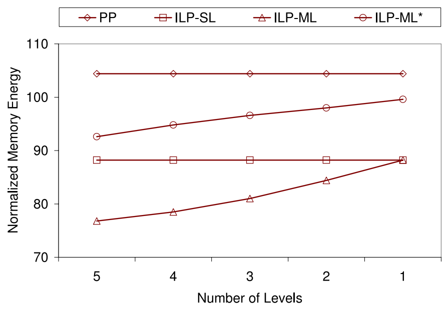

Figure 7: Impact of the number of levels in the memory hierarchy.

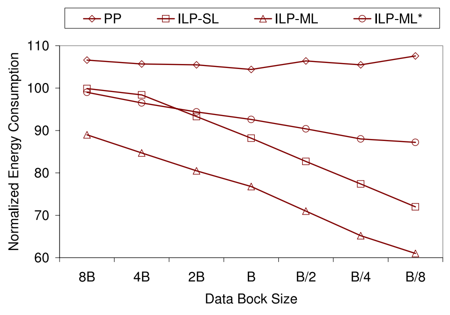

Figure 8: Impact of data block size.

of this graph corresponds to the default on-chip memory capacity used so far in our experiments (M is the total amount of data processed by this application). We see that all the three ILP-based schemes start to converge as the on-chip memory capacity is increased. This can be explained as follows. When the memory space is small, partitioning is important for minimizing energy consumption. However, when memory space is large, it becomes more of a problem of data allocation. Therefore, the benefits brought by the ILP-ML scheme are more emphasized with small memory sizes. This is in a sense an encouraging result though. This is because embedded applications are growing in terms of both complexity and data sizes, and the fact that our approach performs better with smaller memory spaces means that we can expect it to be even more successful in the future.

The third parameter we study is the number of levels in the on-chip memory hierarchy. Recall from Table 4 that in our experiments so far we allowed at most five memory levels. In the results shown in Figure 7 we give energy consumptions with different number of levels (from 1 to 5). Obviously, when we have only one level, ILP-ML reduces to ILP-SL. We see from these results the importance of working with large number of levels if it is possible to do so. This is because the energy savings reduce as we reduce the maximum number of allowable levels. However, not all the applications really use all the available levels. In fact, we observed during our experiments that, for any given application, there is a number of maximum levels beyond which we do not see any further energy reductions.

The last parameter we investigate is the data block size. Figure 8 gives the energy results with different block sizes (B represents the default block size, which is 128 bytes as given in Table 4). We see that both ILP-ML and ILP-SL take better advantage of smaller block sizes as compared to the ILP-ML* scheme. This is mainly because ILP-ML* does not employ optimal data allocation.

To summarize, our experimental analysis shows that the ILP-ML scheme performs really well from both energy and execution time perspectives. Also, our experiments with different parameters emphasize the importance of employing both optimal memory space partitioning and optimal data allocation (together in a unified setting) for the best energy savings.

5. Conclusions

One of the most important issues in designing a chip multiprocessor is to decide its on-chip memory organization. A poor on-chip memory design can have serious power and performance implications when running data-intensive embedded applications (e.g., those from the domain of embedded image/video processing). This paper proposes an integer linear programming (ILP) based solution to the combined problem of on-chip memory hierarchy design and data allocation across the components of the designed hierarchy. We measured the benefits of this approach by designing custom memory hierarchies for ten embedded applications. Our experience with this approach shows that, in order to minimize memory energy consumption, memory space partitioning and data allocation should be done in concert. The results also indicate that the memory hierarchies designed by this approach are much more energy efficient than all the alternate schemes evaluated in this work.

6. REFERENCES

[1] F. Angiolini, L. Benini, and A. Caprara. *Polynomial-Time Algorithm for On-Chip Scratch-Pad Memory Partitioning.* In Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems, San Jose, CA, 2003.

[2] L. Benini, A. Macii, E. Macii, and M. Poncino. Increasing energy efficiency of embedded systems by application-specific memory hierarchy generation. *IEEE Design & Test of Computers*, pages 74–85, April-June, 2000.

[3] Y. Cao, H. Tomiyama, T. Okuma, and H. Yasuura. *Data Memory Design Considering Effective Bit-width for Low-Energy Embedded Systems.* In Proceedings of the 15th International Symposium on System Synthesis, Kyoto, Japan, October 2002.

[4] K. D. Cooper and T. J. Harvey. Compiler-controlled memory. In Proc. *the International Conference on Architectural Support for Prog. Lang. and Operating Systems*, CA, November 1998.

[5] 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.* Kluwer Academic Publishers, 1998.

[6] F. Gharsalli, S. Meftali, F. Rousseau, and A. A. Jerraya. *Automatic Generation of Embedded Memory Wrapper for Multiprocessor SoC.* In Proceedings of the 39th Design Automation Conference, New Orleans, Louisiana, 1999.

[7] E. G. Hallnor and S. K. Reinhardt. A fully-associative software-managed cache design. In Proc. *International Conference on Computer Architecture*, pp. 107–116, Vancouver, British Columbia, Canada, 2000.

[8] M. Kandemir, J. Ramanujam, M. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh. Dynamic management of scratch-pad memory space. In Proc. *the 38th Design Automation Conference*, Las Vegas, NV, June 2001.

[9] M. Kandemir and A. Choudhary. *Compiler-Directed Scratch-Pad Memory Hierarchy Design and Management.* In Proceedings of the Design Automation Conference, New Orleans, LA, June 2002.

[10] M. Kandemir, O. Ozturk, and M. Karakoy. *Dynamic On-Chip Memory Management for Chip Multiprocessors.* In Proc. International Conference on Compilers, Architectures, and Synthesis for Embedded Systems, Washington D.C., September 2004.

[11] S. Meftali, F. Gharsalli, F. Rousseau, and A. A. Jerraya. *An Optimal Memory Allocation for Application-Specific Multiprocessor System-on-Chip.* In Proceedings of the International Symposium on Systems Synthesis, Montreal, Canada, 2001.

[12] G. Nemhauser, L. Wolsey. *Integer and Combinatorial Optimization*, Wiley-Interscience Publications, 1988.

[13] P. R. Panda and L. Chitturi. *An Energy-Conscious Algorithm for Memory Port Allocation.* In Proceedings of the 2002 IEEE/ACM International Conference on Computer-Aided Design, San Jose, California, November 2002.

[14] P. R. Panda, N. D. Dutt, and A. Nicolau. *Architectural Exploration and Optimization of Local Memory in Embedded Systems.* In Proceedings of the 10th international Symposium on System Synthesis, Antwerp, Belgium, September 1997.

[15] A. Ramachandran and M. F. Jacome. *Xtream-Fit: An Energy-Delay Efficient Data Memory Subsystem for Embedded Media Processing.* In Proceedings of the 40th Design Automation Conference, Anaheim, CA, June 2003.

[16] G. Reinman and N. P. Jouppi. *CACTI 2.0: An Integrated Cache Timing and Power Model.* Compaq, WRL, Research Report 2000/7, February 2000.

[17] W.-T. Shiue and C. Chakrabarti. *Memory Exploration for Low-Power Embedded Systems.* In Proceedings of the 36th Design Automation Conferences, New Orleans, LA, 1999.

[18] SIMICS Tool-set. http://www.virtutech.com/

[19] G. E. Suh, L. Rudolph, and S. Devadas. *Dynamic Partitioning of Shared Cache Memory.* Journal of Supercomputing, 2002.

[20] S. Udayakumaran and R. Barua. *Compiler-Decided Dynamic Memory Allocation for Scratch-Pad Based Embedded Systems.* In Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems, San Jose, CA, 2003.

[21] Xpress-MP, http://www.dashoptimization.com/pdf/Mosel1.pdf, May 2002.