Published in IET Computers & Digital Techniques
Received on 17th September 2009
Revised on 7th January 2010
doi: 10.1049/iet-cdt.2009.0089

ISSN 1751-8601

On-chip memory space partitioning for chip multiprocessors using polyhedral algebra

O. Ozturk[1] *M. Kandemir*[2] *M.J. Irwin*[2]

[1]*Department of Computer Engineering, Bilkent University, Bilkent, 06800, Ankara, Turkey*
[2]*Computer Engineering Department, The Pennsylvania State University, University Park, PA 16802, USA*
E-mail: ozturk@cs.bilkent.edu.tr

Abstract: One of the most important issues in designing a chip multiprocessor is to decide its on-chip memory organisation. While it is possible to design an application-specific memory architecture, this may not necessarily be the best option, in particular when storage demands of individual processors and/or their data sharing patterns can change from one point in execution to another for the same application. Here, two problems are formulated. First, we show how a polyhedral method can be used to design, for array-based data-intensive embedded applications, an application-specific hybrid memory architecture that has both shared and private components. We evaluate the resulting memory configurations using a set of benchmarks and compare them to pure private and pure shared memory on-chip multiprocessor architectures. The second approach proposed consider dynamic configuration of software-managed on-chip memory space to adapt to the runtime variations in data storage demand and interprocessor sharing patterns. The proposed framework is fully implemented using an optimising compiler, a polyhedral tool, and a memory partitioner (based on integer linear programming), and is tested using a suite of eight data-intensive embedded applications.

1 Introduction

The ability to pack millions of transistors into a single core enables chip multiprocessor design, where multiple processor cores reside in the same chip. An important advantage of this architecture is that it reduces the cost of interprocessor communication from both performance and power perspectives as this communication does not need to go over off-chip buses. A potential drawback is the increased cost of off-chip accesses as compared to interprocessor communication. In particular, multiple cores can try to use the same set of buses/pins to go off-chip, and this can put a high pressure on the memory subsystem. Consequently, embedded software designers for chip multiprocessors prefer on-chip communication over off-chip memory accesses.

Conventional on-chip memories in chip multiprocessors can be of two types: shared or private. In the shared case, all processors share the same on-chip memory space. Only one processor can access a shared memory location at a time. Synchronisation is done through shared memory. Data sharing among processors depends on the speed of memory access, and can be fast if the number of processors is small. But, the memory bandwidth is limited. An increase in the number of processors without an increase of bandwidth causes bottlenecks. In the private case, each processor can only access its own private memory. Data are shared across an on-chip communication network using message passing. Synchronisation is also achieved by message passing. Memory and bandwidth are scalable with the number of nodes but data sharing among processors is slowed down due to the latency of interconnection networks. There are pros and cons for each type of memory. In the private memory case, the memory units are closer to a processor, not requiring shared bus accesses in the common case, thereby reducing access latency as well as bus contention. A drawback of this approach is potential duplication of hot data in multiple on-chip memories, leading potentially to an underutilisation of the available aggregate on-chip memory space. Clearly, the density of duplication is determined by the degree of data sharing between the parallel processors. In addition to this problem, working with equal-sized private memories can also introduce load balancing problems; that is, while some

processors need more private memory space than available to them, the rest may not use available private space allocated to them. While a shared on-chip memory solves these two problems to a certain extent, it typically costs higher access latency and higher power consumption per access, and leads to high contention on the interconnection network.

In an attempt to come up with a hybrid solution that takes good characteristics of these two solutions while leaving out the bad ones, customised on-chip memory design for chip multiprocessors is first proposed that execute array-based data-intensive embedded applications. Customisation in this context brings two non-uniformities. First, each processor can have a different size private memory, and second, some processors can share a common memory while the others do not. Fig. 1 illustrates private, shared, and a sample hybrid on-chip memory architecture. Our first goal in this paper is to determine a suitable hybrid memory hierarchy for a given application, i.e. application-specific software-managed memory design for chip multiprocessors. To achieve this objective, our approach proceeds in two steps:

- We estimate the amount of data accessed by each processor and shared between different processors. We do this by formulating the set of accessed/shared elements using Presburger formulas, and counting them (using a polyhedral tool). The collected statistics are then fed into the second step.

- We divide the available on-chip memory space into shared and private parts according to magnitudes of shared and private data sets. After this division, we decide how the private parts should be distributed across the processors, and the shared parts across the processor pairs. The outcome of this step is a hybrid memory architecture, where in the most general case some processors share memory whereas the others do not.

We evaluated the hybrid memory designs obtained through our approach from both performance and power perspectives. In addition, we compared them to pure private and shared architectures. When running the same set of array-based data-intensive embedded applications with the same code optimisations, our results indicate that the proposed hybrid memory design methodology leads to much less power consumption than the conventional memory architectures. These results not only show that such hybrid memory architectures are promising for chip multiprocessors, but they also open up opportunities for novel compiler optimisations in the future that could take non-uniform memory sizes of the processors and irregular data sharing into account.

While such specialised hybrid architectures generate very good results from both power and performance angles, the resulting design may not be extremely flexible because the data access pattern of the entire application is captured in a single memory configuration. This is particularly true when relative memory demands and data sharing patterns of processors change from one portion of the code to another during the course of execution. In such cases, a fixed

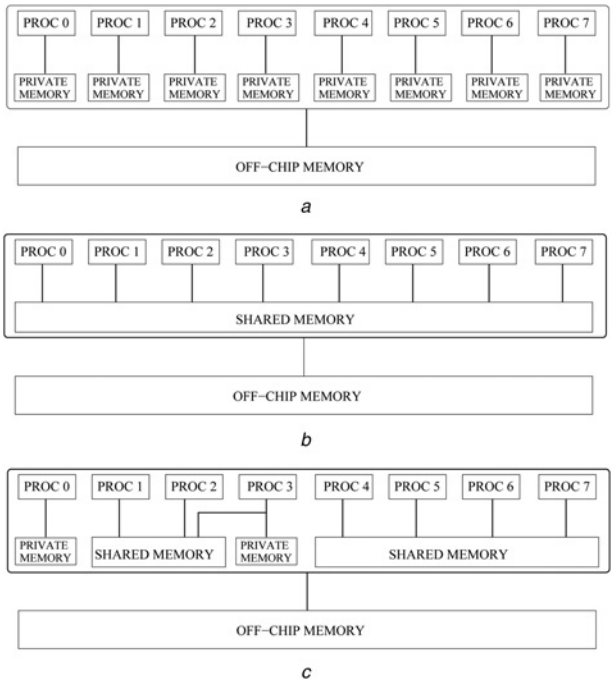

Figure 1 *Private a, shared b and hybrid c on-chip memory architectures*

(static) memory configuration may not be optimal for individual program phases. Motivated by this argument, a reconfigurable software-managed on-chip memory design for chip multiprocessors is also proposed here. The idea is to change the memory partitioning dynamically between the processors at runtime when execution moves from one phase of computation to another.

To achieve this dynamic memory partitioning, we divide the memory spaces into slices of equal sizes, and perform a new slice allocation (across processors) at each phase of the computation. A slice allocation indicates how the slices are assigned to processors. Then, the question is to decide the slice allocation for each programme phase. In our proposal, this is handled by an analysis of the application code based on a polyhedral tool that uses Presburger arithmetic. Specifically, an optimising compiler analyses each program phase with the help of the polyhedral tool, and decides the most suitable on-chip memory partitioning for it. It subsequently inserts special instructions in the code to switch from one memory configuration to another when the execution moves from one phase to another. We compared this dynamic memory management strategy with both classical memory architectures (pure shared and pure private on-chip memories) and an application-specific architecture that uses the same memory configuration for the entire program execution. Our experimental evaluation indicates that the proposed technique is very effective in practice and leads to much less energy consumption than all the alternatives tested.

The rest of this paper is organised as follows. Section 2 discusses our static approach and Section 3 discusses our

dynamic scheme. An experimental evaluation of the proposed schemes is presented in Section 4. Section 5 discusses related work. Section 6 concludes with a summary and an outline of the future work.

2 Application-specific static design

In this section, we discuss our formulation of the problem that determines a customised on-chip memory for a given application. Note that, here the focuses is on array-based, loop-intensive embedded programme. Array/loop-intensive applications are frequently used in embedded image/video processing. Such an application is typically composed of a set of loop nests; each loop nest accesses a set of arrays. For these applications loop bounds are either known or profile data are available. An optimising compiler can analyse these loop-intensive applications with regular data access patterns.

2.1 Presburger formulation

Presburger formulation is a class of logical formulas which can be built from affine constraints over integer variables, the logical connectives ($\vee$, $\wedge$, and $\neg$), and the existential and universal quantifiers ($\exists$ and $\forall$). In this work, we employ a polyhedral tool called the Omega Library [1] to manipulate integer tuple relations and sets, which are described using Presburger formulas.

In order to partition the memory space among the processors, we need two types of information: the amount of private data accessed by each processor, and the amount of data shared across the processors. Our solution to obtaining these is based on representing the data accessed by each processor and each processor pair using Presburger formulas and counting them. After the sets containing Presburger formulations are obtained, we generate code to count the number elements in each set. The resulting numbers (for private and shared data) are then fed to our memory partitioning algorithm.

2.1.1 Representation of parallelism information: We represent parallelisation information for a given nest using a vector referred to as the parallelism vector ($\boldsymbol{\gamma}$). In this vector, entries correspond to loops in the nest (from outermost to innermost), and each entry can be 1 or 0. If an entry is 1, this means that the corresponding loop is parallelised; otherwise, the entry is set to 0. For example, for a nest with four loops, $\boldsymbol{\gamma} = (1\ 1\ 0\ 0)^{\mathrm{T}}$ indicates that the first two loops are parallelised, whereas the iterations of the inner two loops are to be executed sequentially. There are different ways of obtaining the $\boldsymbol{\gamma}$ vector for each loop. First, we can use automatic loop parallelisation theory [2] and identify the loops whose iterations could be executed in parallel. Alternately, we can allow the programmer to explicitly mark the parallel loops, as is also common in high-end parallel computing community [3, 4]. This user-assisted parallelisation is typically supported with directives/pragmas, enabling users to actively assist the compiler in the parallelisation process. Our approach, which is described in the rest of this here, is independent of the parallelisation strategy employed.

2.1.2 Representation of iteration and data sets and counting them: A loop space contains all the iterations executed by the loop and is represented by a set whose boundaries are defined by loop limits. Consider, for example, the loop nest given in Fig. 2. We can represent the iteration space of this loop nest using the following Presburger formulation

$$\mathcal{L} = \{(i_1, i_2, i_3) \mid L_1 \le i_1 \le U_1 \wedge L_2 \le i_2 \le U_2 \wedge L_3 \le i_3 \le U_3\}$$

The set of loop iterations that will be executed by a processor is determined by parallelism vector ($\boldsymbol{\gamma}$) as well as the scheduling strategy adopted. In this paper, we employ a static block scheduling, where each processor is assigned a set of successive loop iterations. Assuming that we have $\boldsymbol{p} = (1\ 0\ 0)^{\mathrm{T}}$ for the loop nest in Fig. 2 and that we have four processors (PROC0 through PROC3), the set of loop iterations that will be executed by processor c (where $0 \le c \le 3$) can be determined as

$$\mathcal{K}(c) = \left\{(k_1, k_2, k_3) \mid \left(L_1 + \frac{c(U_1 - L1 + 1)}{4} \le k_1 < L1 + \frac{(c+1)(U_1 - L1 + 1)}{4} - 1\right) \wedge (L_2 \le k_2 \le U_2) \wedge (L_3 \le k_3 \le U_3)\right\}$$

We assumed (for simplicity) that $(U_1 - L_1)$ is evenly divided by four. To specify the set of data elements accessed by each processor, we need to consider each array reference within the loop. Continuing with our example, for processor c, we have the following sets

$$\mathcal{S}(c, A[i_2, i_3 + i_1]) = \{(a_1, a_2) \mid \exists (i_1, i_2, i_3) \quad \text{such that} \quad (i_2 = a_1 \wedge i_1 + i_3 = a_2) \wedge (i_1, i_2, i_3) \in \mathcal{K}(c)\}$$

$$\mathcal{S}(c, A[i_2 + i_3, i_3 - i_2]) = \{(a_1, a_2) \mid \exists (i_1, i_2, i_3) \quad \text{such that} \quad (i_2 + i_3 = a_1 \wedge i_3 - i_2 = a_2) \wedge (i_1, i_2, i_3) \in \mathcal{K}(c)\}$$

$$\mathcal{S}(c, A[i_2 + 2, i_3 + i_1 - 1]) = \{(a_1, a_2) \mid \exists (i_1, i_2, i_3) \quad \text{such that} \quad (i_2 + 2 = a_1 \wedge i_1 + i_3 - 1 = a_2) \wedge (i_1, i_2, i_3) \in \mathcal{K}(c)\}$$

$$\begin{aligned} &for(i_1 = L_1; i_1 \le U_1; i_1 + +)\\ &\quad for(i_2 = L_2; i_2 \le U_2; i_2 + +)\\ &\quad\quad for(i_3 = L_3; i_3 \le U_3; i_3 + +)\\ &\quad\quad\quad \cdots = A[i_2, i_3 + i_1] + A[i_2 + i_3, i_3 - i_2] + A[i_2 + 2, i_3 + i_1 - 1]; \end{aligned}$$

Figure 2 *Example loop nest*

IET Comput. Digit. Tech., 2010, Vol. 4, Iss. 6, pp. 484–498
doi: 10.1049/iet-cdt.2009.0089

As a result, the total set of data items (array elements) accessed by processor c can be expressed as

$$\mathcal{S}_{\text{total}}(c) = \mathcal{S}(c, A[i_2, i_3 + i_1]) \cup \mathcal{S}(c, A[i_2 + i_3, i_3 - i_2]) \cup \mathcal{S}(c, A[i_2 + 2, i_3 + i_1 - 1])\}$$

where $\cup$ denotes the set union.

To capture data sharing between processors, we need to specify the set of common elements accessed by each pair of processors. Returning to our current example, let us consider two processors (c and c'). The array elements shared between them can be written as

$$\mathcal{C}(c, c') = \{(a_1, a_2) \,|\, (a_1, a_2) \in \mathcal{S}_{\text{total}}(c) \wedge (a_1, a_2) \in \mathcal{S}_{\text{total}}(c')\}$$

Informally, for an element to belong to set $\mathcal{C}(c, c')$, it must be accessed by both the processors (i.e. c and c'). Consequently, the private data elements for processor c (i.e. those that are accessed only by processor c) can be computed as

$$\mathcal{P}(c) = \mathcal{S}_{\text{total}}(c) - \bigcup_{\forall c' \text{s.t.} c' \neq c} \mathcal{C}(c, c')$$

However, for our memory space partitioning approach, we also need to find the number of elements in sets $\mathcal{P}(c)$ and $\mathcal{C}(c, c')$. To do this, we need to be able to enumerate the elements in these sets. For this purpose, we employ the following strategy. Using the codegen utility provided by the Omega Library [1], we first generate a code (typically a nested loop sequence) that iterates over the elements in a given set. Then, we place a counter (variable) in this code, and find the number of times the counter is updated, which gives us the number we want. In other words, what we do is to create a code and execute it. While executing such a code fragment for each set of interest can take time, we believe that this overhead is within the tolerable limits. Nevertheless, we also quantify this overhead for the benchmark codes used in this study.

Note that, if we have multiple nests and arrays in the application, we determine $\mathcal{P}(c)$ and $\mathcal{C}(c, c')$ considering all the nests and arrays. Specifically, let $\mathcal{C}(c, c', N_i, A_j)$ be the set of the elements of array A_j shared by processors c and c' in nest N_i. Based on these, we can write

$$\mathcal{C}(c, c') = \bigcup_i \bigcup_j \mathcal{C}(c, c', N_i, A_j)$$

and

$$\mathcal{P}(c) = \mathcal{S}_{\text{total}}(c) - \bigcup_{\forall c' \text{s.t.} c' \neq c} \mathcal{C}(c, c')$$

The first one of these equalities, determine the total shared elements between processors c and c' across all nests and arrays in the application. The second one, on the other hand, gives us the set of data elements private to processor c, assuming that $\mathcal{S}_{\text{total}}(c)$ is the total number of elements (from all arrays and in all nests) accessed by that processor.

2.2 Memory partitioning

2.2.1 Base case: Let us use P to denote the number of CPUs in our chip multiprocessor. We assume that the on-chip memory space is divided into L units (also called slices) of equal size. We define a module (also called bank) as a number of units put together; that is, the units are the building blocks for the modules. Each module can be shared or private. If it is private, it can be accessed by only one processor (its owner). If it is shared, any number of processors can access it, as determined by our approach. We proceed in two steps:

- In the first step, we divide the available memory space between private and shared modules. Let $\mathcal{P}_{\text{total}} = \sum_{c=0}^{P-1} |\mathcal{P}(c)|$ and $\mathcal{C}_{\text{total}} = \sum_{c=0}^{P-1} \sum_{c'=0, c'>c}^{P-1} |\mathcal{C}(c, c')|$. That is, $\mathcal{P}_{\text{total}}$ and $\mathcal{C}_{\text{total}}$ hold the total number of private elements and shared elements, respectively. Using these two numbers, we allocate L_{p} units of memory for the private data and L_{c} units of memory for the shared data, where L_{p} and L_{c} are defined as follows

$$L_{\text{p}} = \left\lfloor \frac{\mathcal{P}_{\text{total}}}{\mathcal{P}_{\text{total}+\mathcal{C}_{\text{total}}}} \times L \right\rfloor \quad \text{and} \quad L_{\text{c}} = \left\lceil \frac{\mathcal{C}_{\text{total}}}{\mathcal{P}_{\text{total}} + \mathcal{C}_{\text{total}}} \times L \right\rceil$$

Note that $L = L_{\text{p}} + L_{\text{c}}$. Note also that, this allocation makes sense as it gives more memory to whichever type of data (private or shared) dominates the other.

- In the second step, we focus on L_{p} and L_{c} separately, and divide them across the processors and processor pairs respectively. Specifically, the private memory allocated to processor c, denoted L_{p_c}, is computed as

$$L_{\text{p}_c} = \left\lfloor \frac{|\mathcal{P}(c)|}{\mathcal{P}_{\text{total}}} \times L_{\text{p}} \right\rfloor$$

Similarly, the shared memory allocated to processors c and c' is calculated as

$$C_{\text{p}_{c,c'}} = \left\lfloor \frac{|\mathcal{C}(c, c')|}{\mathcal{C}_{\text{total}}} \times L_{\text{c}} \right\rfloor$$

It is to be noted that, we have

$$L' = \left\{ \sum_{\forall c} L_{\text{p}_c} + \sum_{\forall c} \sum_{\forall c' \text{s.t.} c' > c} C_{\text{p}_{c,c'}} \right\} \leq L$$

Here, L' is the number of units (slices) that are actually partitioned. If $L' < L$, our current implementation gives the extra units $(L - L')$ to the smallest module (bank). If there exist more than one such module, the selection among them is arbitrary.

2.2.2 Extension to higher-order data sharing: In this subsection, we discuss how our approach can be

extended to capture higher order data sharing. A drawback of the scheme presented so far is that it allows a memory module to be shared by at most two processors. Consequently, if three or more processors share significant amount of data, the approach presented above may not be particularly efficient since it creates a separate shared module for each pair of processors that share data. There are at least two ways of getting around this problem, which are discussed below.

(a) *Extended Presburger formulation-based approach:* In this approach, we extend the $\mathcal{C}(c, c')$ sets explained above to capture data sharings between three or more processors. As an example, $\mathcal{C}(c, c', c'')$ holds the data elements shared by processors c, c', and c''. Returning to our running example, we compute $\mathcal{C}(c, c', c'')$ as

$$\mathcal{C}(c, c', c'') = \{(a_1, a_2) \mid (a_1, a_2) \in \mathcal{S}_{\text{total}}(c) \wedge (a_1, a_2) \in \mathcal{S}_{\text{total}}(c') \wedge (a_1, a_2) \in \mathcal{S}_{\text{total}}(c'')\}$$

Then, in the memory partitioning part, we take into account $|\mathcal{C}(c, c', c'')|$ values, and compute $C_{\mathrm{P}_{c,c,c'}}$ (the amount of shared memory allocated to processors c, c' and c'') as well, in addition to $C_{\mathrm{P}_{c,c'}}$ and L_{P_c}. One can easily extend this approach to capture even higher levels of sharing (i.e. those between four processors and beyond). However, we do not further elaborate on the details of this process, as we have not yet automated this extension fully. Instead, we implemented the post-processing based idea explained in the following subsection.

(b) *Post-processing-based approach:* In this approach, starting with the memory configuration determined by the base approach described above in Section 2.2.1, we hierarchically build larger memories, by a process called module merging. We do this by merging the shared memory modules if they have a common processor assigned to them. Let us assume that we ran the base approach explained in Section 2.2.1, and determined a memory partitioning for our chip multiprocessor. We use the notation share(i, j, T) to indicate that processors i and j share common memory module T (resulted from our approach). Then, we merge share(i_1, j_1, T_1) and share(i_2, j_2, T_2) into share($i_1, j_1, j_2, T_1 + T_2$) if $i_1 = i_2$. Assuming that $T_3 = T_1 + T_2$, in the next step, we merge share(i_1, j_1, j_2, T_3) and share(k_1, l_1, T_4) (if there is such one) into share($i_1, j_1, j_2, k_1, T_3 + T_4$) if $l_1 = j_1$, and so on. Clearly, at the end of this process, we can reach a memory configuration with only a few modules, all of which are heavily shared (in addition to private modules). However, it is to be noted that, it is also possible to stop this module merging process before we reach the point at which no further merging is possible.

2.3 Discussion

We now discuss how our hybrid memory architecture operates. Note that there are three scenarios for the outcome of a memory access in the hybrid on-chip memory architecture:

- *Local hit:* When the processor finds the data in one of the memory modules it has access to.

- *Remote hit:* In this case, the lookup among its assigned module(s) fails, but the data are found in another (on-chip) module that is not assigned to the processor that issued the memory request.

- *On-chip miss:* In this case, the data are not in any of the on-chip modules, and requires an off-chip access. The access cost in this case will involve the cost of the off-chip access.

We also need to explain what happens in the case of a remote hit or on-chip miss. This largely depends on how the software manages the local memory space. Basically, when a processor accesses a data item that is not in the memory modules assigned to the processor, we have two options: either (1) we can use that data from its original location (i.e. we do not copy it to the modules that we have access to), or (2) we can bring the accessed data to one of the modules that we have access to. In our current implementation, we use the second approach as we optimistically expect that the data that has just been accessed will exhibit some temporal reuse in the near future, and thus, keeping it in close proximity will lower the power consumption when it is subsequently accessed. Then, the important question is where (i.e. to which module) should we bring the data. Ideally, if we could know how the data will be shared, we would make a good decision. But, since we do not keep track of runtime data sharing/movements, in this study, we select a random shared module that we have access to, and bring the data there if there is available space. If not, we select an alternate shared module that we have access to, and store the data there, and so on. If none of our shared modules has any available space, we return to the first module we checked and select a victim data from that module and send it to the off-chip memory. While different algorithms can be employed to decide the victim, our current implementation uses an LRU-based replacement scheme.

3 Application-specific dynamic design

In this section, we propose a dynamic memory partitioning scheme that changes memory partitioning during execution when we move from one execution phase to another.

3.1 High-level view

Our goal is to develop a dynamic on-chip memory architecture for chip multiprocessors. What we mean by 'dynamic' in this context is that it adapts its behaviour to the dynamic changes in memory requirements of individual processors and interprocessor data sharing patterns. The proposed approach is depicted in Fig. 3. The application code is read and analysed by our optimising compiler, and the sets that

doi: 10.1049/iet-cdt.2009.0089

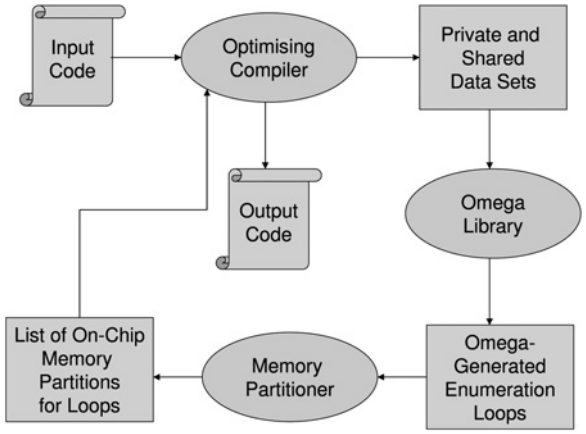

Figure 3 *Overview of the proposed approach to dynamic on-chip memory partitioning*

represent private and shared data items are built and passed to the Omega Library. The Omega Library in turn generates a code that enumerates the elements in these sets and passes them to the memory partitioner. The memory partitioner has two major modules. The first module determines a memory configuration for each program phase (a loop nest in our current implementation), and the second module is responsible from deciding the memory banks to use in the architecture. Once the bank configuration is decided, this information is fed-back to the compiler, which modifies the application source to insert special instructions to change on-chip memory configuration at runtime. Note that the high-level view shown in Fig. 3 is also valid for the static scheme discussed in Section 2, except that an entire program is considered as a single phase.

Fig. 4 illustrates our approach during the execution of a typical scenario with two separate loop nests (program phases in our current implementation). As will be discussed

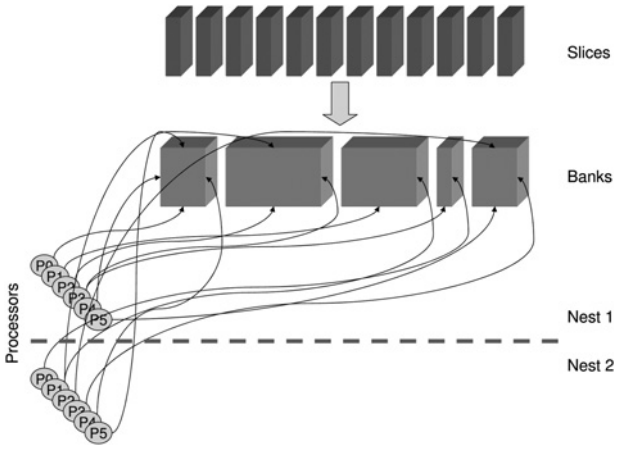

Figure 4 *Example scenario where two loop nests (program phases) use the same set of banks in different ways*

Each bank is constructed from a set of consecutive slices. Each arrow (from processors to banks) indicate that the processor can access that bank

in the next subsection in detail, the dynamic reconfiguration of on-chip memory is actually managed at a bank granularity (instead of a slice granularity), where each bank consists of a set of consecutive slices. Once the bank structure for a given application is determined, each loop nest tries to use these banks in the most energy efficient way. Energy efficiency is achieved by allowing each processor to access the right amount of memory space and making sure that the processors that heavily share data are assigned a common (shared) bank.

It should be observed that proper use of memory banks is critical from a power consumption viewpoint. For example, if a processor is assigned a large bank than necessary, this can increase per access power consumption. On the other hand, if its memory bank is small, this can lead to frequent accesses to off-chip memory (due to not being able to capture all frequently-accessed data in the on-chip memory), which again increases power consumption.

3.2 Slice/bank management

In our slice-based organisation, each processor is assigned a set of slices, and they form its local memory. The slices assigned to other processors in the system collectively form a remote memory for this processor. Finally, the system also has an off-chip memory. The success of an application in achieving energy efficiency in this architecture depends strongly on the percentage of data requests satisfied from local, remote and off-chip memories. As has been discussed in the static case, we normally want all data requests to be satisfied from the local memory (first choice) or remote memory (second choice), i.e. as far as possible, we do not want to go to the off-chip memory. The memory management within a program phase in the dynamic scheme is the same as that in the static scheme discussed in Section 2.3.

Our approach actually manages the on-chip memory at the banks level, not at the slice level. The main reason for this is that slice level management is too fine-granular for dynamic assignment, and this can have serious consequences as far as area requirements and chip interconnect complexity are concerned. In this context, a bank corresponds to a set of slices consecutive in the on-chip memory space. The process of determining the banks to use for a given application is explained in Section 3.4. An important characteristic of our bank-based memory management strategy is that each loop nest can use the banks in a different fashion. For example, while in a loop nest a memory bank can be shared between processors 0 and 1, in the next loop nest it can be shared between processors 1 and 3. Similarly, while processor 2 has a private bank in the first nest (in addition to the bank(s) it shares with the others), it is possible that it does not have a private memory bank in the second loop nest. This dynamic bank allocation (or coarse-grain slice allocation) is necessary to make sure that we adapt the on-chip memory configuration to the dynamically changing requirements of the application at hand.

To manage bank allocation across program phases, we propose to employ a table in the memory controller (called the bank table) as shown in Fig. 5 for a sample chip multiprocessor architecture with six processors and eight banks. In the bank table, the columns denote the memory banks and the rows denote the processors. An 'X' in the table indicates that the corresponding bank is assigned to the specified processor. An important characteristic of this table is that it resides within the memory controller, and its contents can be manipulated using special instructions that are inserted by the compiler (this code modification part will be discussed in Section 3.5). When a memory request comes from a processor, we simply look at the corresponding row and identify the relevant banks (local memory), calculate the memory location that needs to be accessed, and then go to that location to access the data if it is in a location in the local memory. If not, we access it from a remote bank or off-chip memory (depending on its location), and bring it to the local memory of the processor. We would like to emphasise at this point that, using this table-based allocation one can generate very different memory configurations (bank partitions). For example, when there exist multiple 'X's in a given column (as in the case of bank 5 in Fig. 5), this means that some banks are shared by multiple processors. It is also to be noted that, in this architecture, changing bank allocation (i.e. changing on-chip memory configuration) means changing the contents of the bank table. Therefore, changing the bank configuration means rewriting the entries of the bank table. The three important questions addressed here are: (1) How can we determine individual bank structures required by processors in a given program phase? (2) How can we determine banks for the entire application considering the individual bank requirements of the program phases? and (3) How can we assign banks to processors as we move from one program phase to another during execution? These three questions are addressed in Sections 3.3–3.5 respectively.

3.3 Determining bank requirements for a program phase

This is very similar to the static case discussed earlier in Section 2. As an example, let us consider the loop nest shown in Fig. 6. We can represent the iteration space of this loop nest using the following Presburger formulation

$$\mathcal{I} = \{(i_1, i_2, i_3) \,|\, (L_1 \le i_1 \le U_1) \wedge (L_2 \le i_2 \le U_2) \wedge (L_3 \le i_3 \le U_3)\}$$

	Memory Banks							
Processors	0	1	2	3	4	5	6	7
0		X			X			
1					X			X
2	X				X		X	
3					X	X	X	
4	X			X		X		
5			X			X		

Figure 5 *Example bank table*

$$\begin{aligned} &for(i_1 = L_1; i_1 \le U_1; i_1 ++)\\ &\quad for(i_2 = L_2; i_2 \le U_2; i_2 ++)\\ &\quad\quad for(i_3 = L_3; i_3 \le U_3; i_3 ++)\\ &\quad\quad\quad \cdots = U[i_3, i_1 + i_2 + 3] + U[i_2 + i_3, i_3 - i_2] + U[i_2 + 2, i_3 + i_1 - 1]; \end{aligned}$$

Figure 6 *Example loop nest*

When a loop is parallelised across the processors in our chip multiprocessor, each processor typically executes a subset of the iteration points in the iteration space. For example, assuming that the compiler/user chose to parallelise the first loop (i_1-loop) in the nest in Fig. 6, processor s is assigned the following loop iterations, assuming that we have P processors, each processor gets a set of successive loop iterations, and P evenly divides $(U_1 - L_1)$

$$\mathcal{W}(s) = \left\{(k_1, k_2, k_3) \middle| \left(L_1 + \frac{s(U_1 - L1 + 1)}{P} \le k_1 < L1 + \frac{(s+1)(U_1 - L1 + 1)}{P - 1}\right) \wedge (L_2 \le k_2 \le U_2) \wedge (L_3 \le k_3 \le U_3)\right\}$$

For this example, the array elements accessed by processor s can be found using the following Presburger formulation, which makes use of the subscript functions:

$$\mathcal{T}_{\text{tot}}(s) = \mathcal{T}(s, U[i_3, i_1 + i_2 + 3]) \cup \mathcal{T}(s, U[i_2 + i_3, i_3 - i_2]) \cup \mathcal{T}(s, U[i_2 + 2, i_3 + i_1 - 1])\}$$

where

$$\begin{aligned} &\mathcal{T}(s, U[i_3, i_1 + i_2 + 3]) = \{(a_1, a_2) \,|\, \exists(i_1, i_2, i_3) \text{ such that } (i_3 = a_1 \wedge i_1 + i_2 + 3 = a_2) \wedge (i_1, i_2, i_3) \in \mathcal{W}(s)\}\\ &\mathcal{T}(s, U[i_2 + i_3, i_3 - i_2]) = \{(a_1, a_2) \,|\, \exists(i_1, i_2, i_3) \text{ such that } (i_2 + i_3 = a_1 \wedge i_3 - i_2 = a_2) \wedge (i_1, i_2, i_3) \in \mathcal{W}(s)\}\\ &\mathcal{T}(s, U[i_2 + 2, i_3 + i_1 - 1]) = \{(a_1, a_2) \,|\, \exists(i_1, i_2, i_3) \text{ such that } (i_2 + 2 = a_1 \wedge i_3 + i_1 - 1 = a_2) \wedge (i_1, i_2, i_3) \in \mathcal{W}(s)\} \end{aligned}$$

In order to capture data sharing between processors, we need to identify the set of common elements accessed by each pair of processors. Let $\mathcal{N}(s, s')$ represent the set of common array elements accessed by processors s and s'. We can write this set as follows

$$\mathcal{N}(s, s') = \{(a_1, a_2) \,|\, (a_1, a_2) \in \mathcal{T}_{\text{tot}}(s) \wedge (a_1, a_2) \in \mathcal{T}_{\text{tot}}(s')\}$$

Based on this, the private data elements for processor s (i.e. those that are accessed only by processor s) can be

doi: 10.1049/iet-cdt.2009.0089

computed as

$$\mathcal{G}(s) = \mathcal{T}_{\text{tot}}(s) - \bigcup_{\forall s' \text{s.t.} s' \neq s} \mathcal{N}(s, s')$$

where ' − ' denotes set subtraction. If we have multiple arrays in the loop nest of interest, we determine $\mathcal{G}(s)$ and $\mathcal{N}(s, s')$ considering all the arrays. Specifically, let $\mathcal{N}(s, s', U_j)$ be the set of the elements of array U_j shared by processors s and s'. So, we have

$$\mathcal{N}(s, s') = \bigcup_j \mathcal{N}(s, s', U_j)$$

and

$$\mathcal{G}(s) = \mathcal{T}_{\text{tot}}(s) - \bigcup_{\forall s' \text{s.t.} s' \neq s} \mathcal{N}(s, s')$$

Here, the first one of these equalities gives the set of all shared data elements between processors s and s' when considering all the arrays in the loop nest, and the second one gives us the set of data elements private to processor s, assuming that $\mathcal{T}_{\text{tot}}(s)$ is the total number of elements (from all arrays) accessed by processor s.

The next task is to determine the number of elements in $\mathcal{G}(s)$ and $\mathcal{N}(s, s')$ sets. To do this, we need to be able to enumerate the elements in these sets. As in the static case, for this purpose, we use the codegen utility provided by the Omega Library. For our example loop nest in Fig. 6, the code generated for counting the elements in $\mathcal{N}(0, 2)$ is shown in Fig. 7 (in this code, intMod refers to the integer modulus operation), under the assumption that $L_j = 1$ and $U_j = 1000$ for all i_j, where $j = 1, 2, 3$. The space partitioning for private and shared components are handled in a similar fashion to the static case.

3.4 Determining memory banks

The previous section presented a scheme using which one can come up with a memory partitioning for each loop nest in a given application. However, it is not trivial (if not impossible) to dynamically change the on-chip memory configuration (at runtime) at a slice level. Instead, our approach builds a banked (module based) architecture, considering the individual memory partitions obtained for each nest. An important characteristic of such a banked architecture is that it combines the characteristics of the individual partitionings (determined for each program phase as described above) as much as possible, and as mentioned earlier, the different loop nests can use the banks in a different fashion. Our approach can be explained as follows. First, we determine, for each loop nest (program phase), the memory partitionings (bank structure) as discussed in the previous section. An example is illustrated in Fig. 8 for four separate loop nests and a total of 32 kB on-chip memory space. Each column in

```
|N(0,2)| = 0;
for(t1 = 1; t1 <= 2000; t1++)
    if(t1 <= 1000){
     for(t2 = -t1 + 2; t2 <= min(t1 - 2, 500); t2+ = 2)
      |N(0,2)|++;
    }
    if(t1 <= 1000){
     for(t2 = 501; t2 <= min(t1 - 2, 504); t2++)
      if(intMod(t1 + t2, 2) == 0){
       |N(0,2)|++;
      }
      if(intMod(t1 + t2 + 1, 2) == 0){
       |N(0,2)|++;
      }
    if(t1 <= 1002&&t1 >= 1001){
     for(t2 = t1 - 2000; t2 <= -t1 + 1502; t2+ = 2)
      |N(0,2)|++;
    }
    if(t1 >= 1001&&t1 <= 1002){
     for(t2 = -t1 + 1503; t2 <= 504; t2++)
      if(intMod(t2 + t1, 2) == 0){
       |N(0,2)|++;
      }
      if(intMod(t2 + t1 + 1, 2) == 0){
       |N(0,2)|++;
      }
    }
    if(t1 >= 1003){
     for(t2 = t1 - 2000; t2 <= min(-t1 + 2000, 504); t2+ = 2)
      |N(0,2)|++;
    }
    if(t1 >= 3){
     for(t2 = max(t1 - 1, 501); t2 <= 504; t2++)
      |N(0,2)|++;
    }
    if(t1 <= 1000){
     for(t2 = 505; t2 <= 1253; t2++){
      |N(0,2)|++;
     }
    }
    if(t1 <= 1002&&t1 >= 1001){
     for(t2 = 505; t2 <= -t1 + 2000; t2++)
      if(intMod(t1 + t2, 2) == 0){
       |N(0,2)|++;
      }
      if(intMod(t1 + t2 + 1, 2) == 0){
       |N(0,2)|++;
      }
    }
    if(t1 >= 1003){
     for(t2 = 505 + intMod((-t1 - 505), 2); t2 <= -t1 + 2000; t2+ = 2)
      |N(0,2)|++;
    }
    if(t1 >= 1001&&t1 <= 1002){
     for(t2 = -t1 + 2001; t2 <= 1249; t2++)
      |N(0,2)|++;
    }
}
```

Figure 7 *Example (enumeration) code generated with the help of the Omega Library*

Variable $|\mathcal{N}(0, 2)|$ holds the number of elements shared between processors 0 and 2

this figure corresponds to a different phase of the application. An entry marked 'xKB-p' indicates an x KB private memory, and an entry marked 'yKB-s' means a y KB shared memory. Here, we use the term 'virtual banks'

Phase #1	Phase #2	Phase #3	Phase #4
16KB-s	8KB-s	16KB-s	4KB-s
8KB-s	8KB-s	4KB-s	4KB-s
4KB-p	8KB-s	4KB-p	4KB-s
1KB-p	2KB-p	4KB-p	4KB-s
1KB-p	2KB-p	4KB-p	4KB-s
1KB-p	2KB-p		4KB-s
1KB-p	1KB-p		2KB-p
	1KB-p		2KB-p
			2KB-p
			1KB-p
			1KB-p

Figure 8 *Example memory partitionings (virtual banks) for four different phases (total on-chip memory capacity = 32 kB)*

to refer to the bank structure determined for each program phase. Our goal is to come up with a final bank configuration (called the 'physical banks' in the rest of this paper) such that it satisfies the individual requirements of the loop nests of the application as much as possible. Our implementation tries to achieve this by using an integer linear programming (ILP)-based formulation. For simplicity, we assume, for a total 32 kB of on-chip memory space, that the only possible bank sizes available are 1, 2, 4, 8, 16 and 32 kB, all of which can be shared or private. In this subsection, we refer to these banks as type-1, type-2, type-4, type-8, type-16, and type-32 in that order.

Let $l_{i,j}$ denote the number of banks of type-i demanded by loop nest j, and let θ_i be the number of banks of type-i in the final (physical bank) configuration. Note that, we want to determine the θ_i values (for the application under consideration), whereas the $l_{i,j}$ values are given (as in Fig. 8). Our approach employs an objective function that tries to minimise the disparities between the $l_{i,j}$ and θ_i values to the greatest extent possible. In mathematical terms, we want to minimise

$$\sum_{j=1}^{Q} w_j \sum_{i=1,2,4,8,16,32} |\theta_i - l_{i,j}|$$

where Q is the total number of nests in the application, and w_j is the 'weight' attached to nest j (i.e. informally, its relative importance across all the nests in the application). However, we are not allowed to exceed the available memory capacity. Therefore, the following constraint should also be satisfied

$$\sum_{i=1,2,4,8,16,32} i \times \theta_i \leq 32$$

The θ_i values determined from these constraints give us the final memory configuration. However, this does not solve the entire problem since we also need to explain how this final configuration will be used by each loop nest in the application. This issue is addressed in the following section. There are two important issues here that need to be clarified. First, the values of the w_j parameters should be determined based on the relative importance of the program phases with respect to each other. Since our program phases are loop nests, we use the number of iterations in nest j as w_j. Second, the ILP formulation described above does not consider the problem of determining the number of ports for each bank. We postpone this problem to the next section where we determine how each program phase uses each bank. Simply put, if we determine that a bank needs to be shared between two processors in at least one program phase, then we make it double-ported; otherwise, it has a single port.

3.5 Memory reconfiguration across program phases

The physical bank configuration determined from the ILP formulation discussed in the previous section tries to satisfy as many of the loop nests as possible (based on their relative importance as indicated by the w_j values). We now discuss how each loop nest utilises these banks. Our approach is a greedy heuristic that operates as follows. We first build a virtual bank list for each nest (such as the one shown in Fig. 8), not concerning ourselves with the number of ports or with the question of whether a bank will be shared by processors or not. In the next step, we traverse the bank list of each processor, and map its virtual banks to the physical banks determined (in Section 3.4) for the entire application. In doing so, it might be necessary to assign multiple virtual banks to a single large physical bank; or, a single large virtual bank to a set of small physical banks. For example, Fig. 9*a* shows how the bank lists

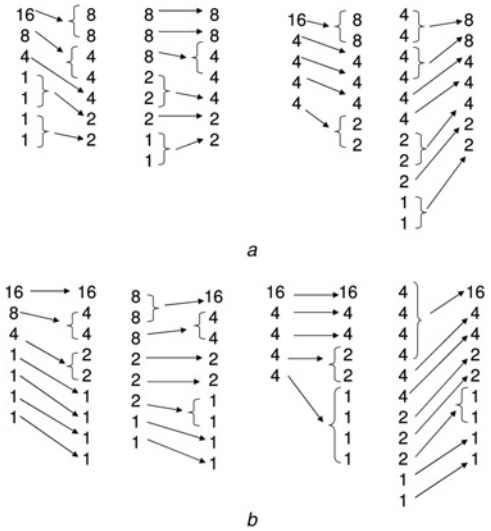

Figure 9 *Two example mappings between virtual banks (see Fig. 8) and physical banks*

The physical banks used in the first mapping are 8, 8, 4, 4, 4, 2 and 2 kB, whereas those used in the second mapping are 16, 4, 4, 2, 2, 1, 1, 1 and 1 kB

doi: 10.1049/iet-cdt.2009.0089

shown in Fig. 8 are handled if the physical banks (determined by the approach explained in the previous section) are 8, 8, 4, 4, 4, 2 and 2 kB. Let us focus on the mapping for the first phase in Fig. 9*a*, and discuss it in more detail. The virtual bank list for this phase is 16, 8, 4, 1, 1, 1 and 1 kB. Since we do not have a 16 kB physical bank, the 16 kB virtual bank requirement of this phase is satisfied by giving it two 8 kB physical banks. Then, we give two 4 kB physical banks to meet its 8 kB virtual bank requirement. The next virtual bank (4 kB) is directly mapped to a 4 kB physical bank. Then, the two 1 kB virtual bank requirements are satisfied by mapping them to a 2 kB physical bank (this pattern occurs twice). It is also important to consider the consequences of this mapping. First, satisfying the 16 kB virtual bank requirement with two 8 kB physical banks increases the port requirements (as any of these two banks can be accessed simultaneously; so, we need to assign two ports per bank). Using two 8 kB banks (instead of one 16 kB bank) can also increase the area demand. Now, let us consider the other case, where 1 kB requirements share a 2 kB bank. This would typically increase the power consumption. The rest of Fig. 9*a* shows the mappings for the remaining phases under this physical bank structure. In comparison, Fig. 9*b* illustrates the virtual-to-physical bank mappings when the physical bank configuration considered is 16, 4, 4, 2, 2, 1, 1, 1 and 1 kB.

3.6 Locality across program phases

An important point that we have not discussed so far is the importance of data locality when the memory configuration is changed across the program phases. As an example, suppose that a processor is assigned a two 4 kB banks in a program phase, and it is assigned a 4 kB bank in the next phase. In this scenario, it might be a good idea, for this processor, to retain (in the second phase) one of the banks it used in the first phase, in an attempt to exploit some inter-phase data reuse. We use ILP to capture these assignments and optimise the locality across program phases. We use $\mathcal{A}_{\mathrm{p,b}_{s,n},\mathrm{ph}}$, returned by the heuristic explained above, for capturing the assignment of memory banks to processors, where p, $\mathrm{b}_{s,n}$, and ph stand for the processor, bank, and phase respectively. Note that, $\mathrm{b}_{s,n}$ is composed of two parameters: size and id. Also, the size parameters (captured by s) are assumed to be in multiples of KBs, whereas ids (n) range from one to total memory size (M_{s}/s), depending on the bank size. More specifically, $\mathcal{A}_{\mathrm{p,b}_{s,n},\mathrm{ph}}$ indicates that whether processor p is using bank $\mathrm{b}_{s,n}$ during phase ph. If ILP returns this value as 1, then we conclude that the bank is being used by processor p. Otherwise, we conclude that it has not been used by the processor. Here we assume that total memory size is given in KBs and the minimum bank size is 1 kB. Recall that the heuristic given in the previous section gives the individual virtual bank information. Our goal here is to exploit this information to improve data locality across the neighbouring program phases

$$\mathcal{K}_{\mathrm{p,b}_{s,n},\mathrm{ph}} \geq \mathcal{A}_{\mathrm{p,b}_{s,n},\mathrm{ph}} + \mathcal{A}_{\mathrm{p,b}_{s,n},\mathrm{ph}-1} - 1, \quad \forall \mathrm{p, b, s, n, ph} \quad (1)$$

In the above formulation, $\mathcal{K}_{\mathrm{p,b}_{s,n},\mathrm{ph}}$ is returned 1 if the bank in question is assigned to the same processor. Based on this expression, our goal would be to choose the next assignment given by $\mathcal{A}_{\mathrm{p,b}_{s,n},\mathrm{ph}}$ in such a way that, it maximises the number of $\mathcal{K}_{\mathrm{p,b}_{s,n},\mathrm{ph}}$ values. We perform this ILP optimisation after each phase $(\mathrm{ph}-1)$ to achieve the maximum locality for the next phase (ph). Therefore, our heuristic performs the following ILP objective once the virtual banks are identified for that phase

$$\max \sum_{p=1}^{P} \sum_{s=1}^{M_{\mathrm{s}}} \sum_{n=1}^{M_{\mathrm{s}}/s} \mathcal{K}_{\mathrm{p,b}_{s,n},\mathrm{ph}}, \quad \forall \mathrm{ph} \quad (2)$$

Note that, in the above expression we do not decide on any virtual bank, rather we map the physical banks to processors in the most locality efficient way.

4 Experimental evaluation

4.1 Setup

We implemented the two schemes proposed in this paper using an optimising compiler and the Omega Library [1]. The compiler part is implemented using the SUIF framework from Stanford University [5]. We simulated a set of applications using a custom multi-bank memory simulator. This simulator takes as input the application executable and memory hierarchy (which can be pure shared, pure private, or hybrid as described here), and keeps track of accesses to different memory banks. Each access is associated with an energy consumption value, whose magnitude depends on the size of the bank and the number of ports it has. The per access energy consumption values are obtained from an input file. The default number of processors used in the experiments is eight, and the default total on-chip memory size is 32 kB. The benchmark codes used in this study are given in Fig. 10, along with the relevant statistics obtained using our simulator with eight processors. The last column of the table in Fig. 10 gives the ratio $\mathcal{F}_{\mathrm{tot}}/\mathcal{E}_{\mathrm{tot}}$, i.e. the ratio

Benchmark	Dataset Size	Description	$\mathcal{F}_{tot}/\mathcal{E}_{tot}$
Atr	812KB	Network address translation	0.799
SP	1,068KB	All-nodes shortest path alg.	1.421
Encr	792KB	Digital signature for security	0.368
Hyper	1,285KB	Machine simulation	1.217
Wood	1,010KB	Colour-based surface inspection	0.606
Usonic	1,044KB	Feature-based estimation alg.	0.231
Jpegview	1,308KB	JPEG image display	0.517
Srec	1,422KB	Speech recognition	1.388

Figure 10 *Benchmark codes used in our experiments*

Each of these benchmarks has both array-based and pointer-based implementations. In this work, we used the array-based implementations

between the total number of shared data elements and the total number of private data elements. We see that, as far as this ratio is concerned, our benchmarks exhibit different behaviours. Specifically, while Usonic is private data intensive, SP exhibits high interprocessor data sharing. The remaining benchmarks fall in between these two extremes. We restrict our exploration mostly to private banks and banks shared between two processors. To support this decision, we give in Fig. 11 the cumulative distribution function (CDF) for data sharing between processors. Each (x, y) point on a curve in this graph says that y% of the total data elements are shared by x or more processors. Note that the y value corresponding to the difference between the $x = 1$ case and the $x = 2$ case gives the percentage of private data. We see from this graph that most of interprocessor sharing occurs between two processors; therefore, it is reasonable to consider shared banks only for processor pairs. Fig. 12 gives the normalised per access energy consumption values for software-managed SRAMs with different capacities and number of ports (for the 0.07 μ process technology). These values have been obtained from CACTI [6] models by making appropriate modifications for software-managed memory (e.g. eliminating tag checks, etc). All values are normalised to that of 2 kB, one port memory. Since the default on-chip capacity is 32 kB, in the pure shared memory case, all processors access a 32 kB memory (in this case, we use three ports since we found this value performs better than the other values we tested). On the other hand, in the pure private memory case, each processor has a single-ported 4 kB private memory (as we have eight processors). As explained earlier, in our banked architecture, if a bank is private, it is single-ported; otherwise (i.e. if it is shared between two processors), it has two ports. Also, we assume a per access memory energy of 6.32 nJ for the off-chip memory (assuming that it is 16 MB). In the rest of the paper, when we mention 'memory energy', we mean the energy consumed in on-chip and off-chip memory accesses, including the energy expended while waiting to resolve port/bus conflicts. We used the publicly available lp_solve [7] tool for determining physical banks. The ILP solution times with this tool ranged from 34.4 s to 4.1 m.

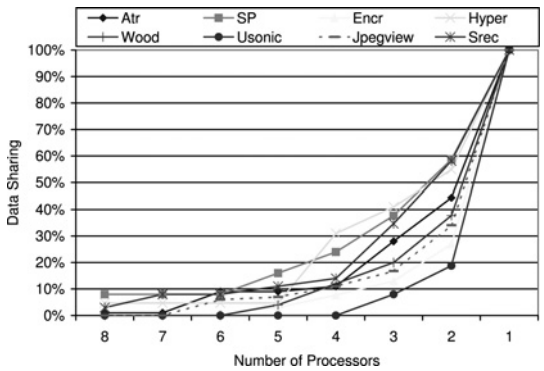

Figure 11 *CDF for interprocessor data sharing*

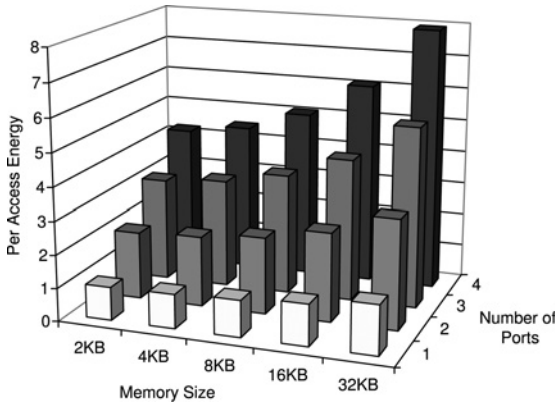

Figure 12 *Per access energy values with different SRAM sizes and number of ports*

All values are normalised with respect to the per access energy consumption of a 2 kB, one port memory

4.2 Results

Unless stated otherwise, the presented results are obtained without running our locality optimisation pass explained in Section 3.6. Later, we also present results quantifying the impact of this additional optimisation. Fig. 13 gives the memory energy consumption values for four different management schemes. The bars marked as 'Private' and 'Shared' correspond to pure private and pure shared memory configurations respectively. The bar marked as 'Application-Specific' shows the result when a fixed, application-specific memory hierarchy is designed for the application. To obtain the results for this version, we used the strategy discussed in Section 2. Finally, 'Application-Specific+' represents the results of the dynamic partitioning approach discussed in this paper. For each application, all bars are given as fractions of the value the first bar represents (i.e. the pure private memory case). Our first observation from this figure is that the proposed dynamic memory partitioning approach generates the best results (among all schemes) for all the applications tested in the experiments. Second, the 'Application-Specific' version

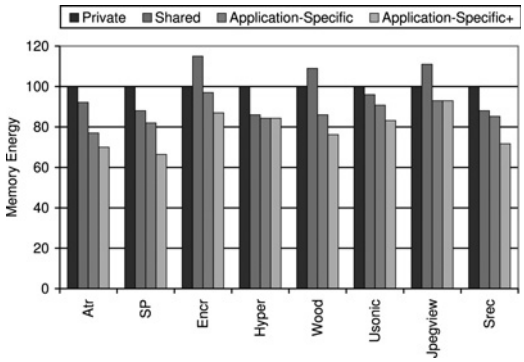

Figure 13 *Memory energy results (eight processors and 32 kB total on-chip memory space)*

Each value is given as a fraction of the value for the pure private memory case

doi: 10.1049/iet-cdt.2009.0089

improves upon both the 'Private' and 'Shared' cases. The third observation is that, while absolute savings change from one application to another, there is a difference between 'Application-Specific' and 'Application-Specific+' in all except two benchmarks, indicating the importance of dynamic reconfiguration at runtime. In Hyper and Jpegview, the two schemes generate the same results, due to the fact that all the program phases in these applications demand similar virtual banks configurations. When averaged over all benchmarks, 'Application-Specific+' reduces the memory energy consumption of 'Application-Specific' by around 10%, which in turn improves upon the pure private and pure shared cases by 13.1% and 11.5% respectively. Although we do not present here in detail, when we allow sharing of a bank by more than two processors (using the approach summarised in Section 2.2.2), we observed an average of about 2% more savings for both 'Application-Specific' and 'Application-Specific+'.

In the rest of our experimental evaluation (except for the performance results), we focus only on the 'Application-Specific+' version. There are two reasons for this decision. First, the sensitivity results with the 'Application-Specific' version are very similar to those with the 'Application-Specific+' version. Second, since 'Application-Specific+' generally performs better than 'Application-Specific', it makes more sense to study it in detail.

In our next set of experiments, we change the number of processors, and study the behaviour of the 'Application-Specific+' version. We see from the results given in Fig. 14 that the effectiveness of our approach increases when the number of processors is increased (keeping the total on-chip memory capacity the same). This result can be explained as follows. When the number of processors is increased, the pure private memory case suffers from the fact that a lot of data are duplicated across the processors, leading to memory space underutilisation. Therefore, the normalised savings increase. It should also be mentioned that (though not presented here in detail) the pure shared memory case also suffers badly when the number of processors is increased, mainly due to the significant increase in port conflicts. These results show that our dynamic on-chip memory partitioning approach exhibits better scalability than the other schemes when the number of processors is increased. Fig. 15 explores what happens when the number of processors is fixed at eight, and the available on-chip memory space is changed from 8 to 256 kB. We see that, in general, our approach generates better savings with small memory capacities. This is because a small capacity renders the problem of memory space management a very important issue. The exceptions observed in this graph are due to the heuristic nature of the proposed approach. Note that, while sophisticated circuit packaging technologies are able to squeeze more and more memories into the same silicon area, the application data set sizes increase at a much higher rate than the increase in (bit/unit area) rate. Consequently, one can expect our scheme to be even more successful in the future.

The next set of experiments study the impact of varying the remote access and off-chip access energies on our savings. In the graph shown in Fig. 16, point 'X' corresponds to the default per access energy consumption for off-chip memory (x-axis) and remote memory (y-axis) for the Srec application. The other values are given multiples or fractions of these default values, while keeping the per access costs for local memory the same as the original. The z-axis gives the memory energy consumption, normalised with respect to that of the pure private memory. We see that the effectiveness of our approach increases when the relative cost of either off-chip or remote access increases. We also observe that the impact of off-chip access cost is more pronounced. The experiments with the remaining applications in our experimental suite generated similar trends; so, we do not present their results.

The next set of results we present are on execution cycles. Fig. 17 gives the execution cycle results for different memory

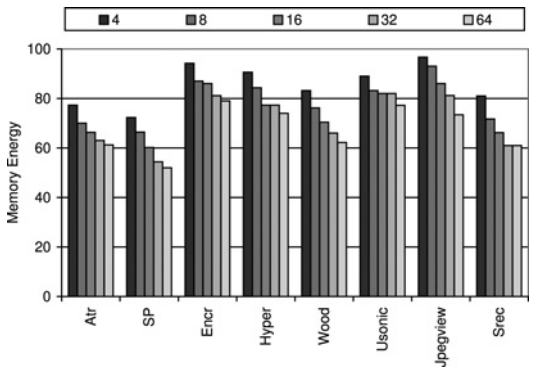

Figure 14 *Memory energy results with different processor sizes (32 kB total on-chip memory space)*

Each value is given as a fraction of the value for the pure private memory case

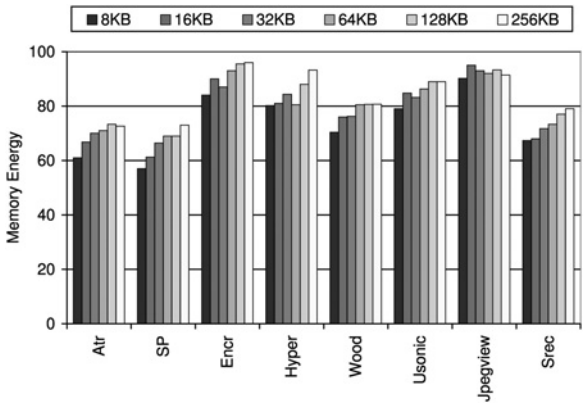

Figure 15 *Memory energy results with different on-chip memory capacities (eight processors)*

Each value is given as a fraction of the value for the pure private memory case

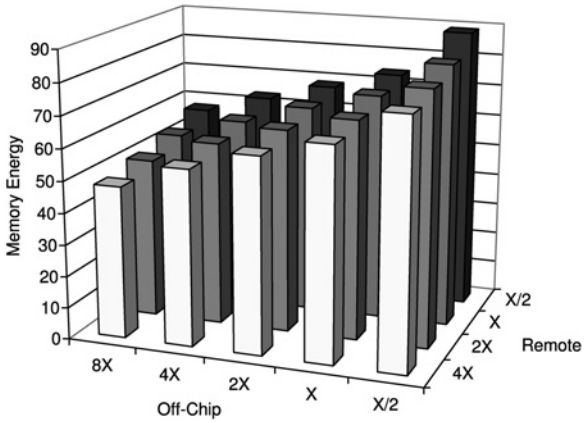

Figure 16 *Memory energy results with different per access energy costs for Srec (eight processors and 32 kB total on-chip memory space)*

Each value is given as a fraction of the value for the pure private memory case

management schemes, normalised with respect to the cycles taken by the pure private memory scheme. We see that while our approaches outperform the remaining schemes from the performance viewpoint as well, the improvements are lower than the energy consumption reductions. The main reason for this is the fact that, as we move from one configuration to another, we may lose some data locality, as has been explained in Section 3.5. We believe that a more sophisticated implementation that addresses this data locality problem during phase transitions can generate better results.

Recall that, so far in our experimental evaluation of the 'Application-Specific+' version, we did not activate the locality optimisation pass, described in Section 3.6. We now present the results with this additional pass and show how much additional improvement it brings. The graph in Fig. 18 gives the normalised energy consumption and execution cycle results with and without this locality

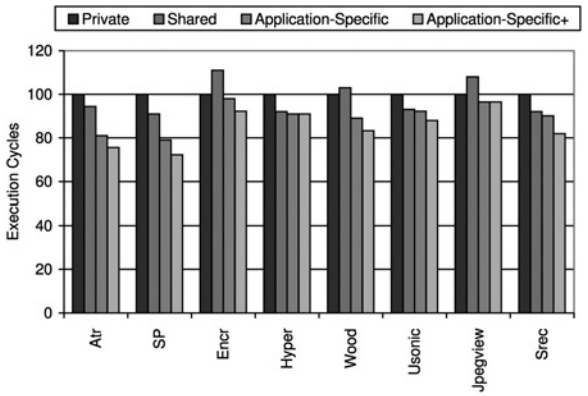

Figure 17 *Execution cycle results (eight processors and 32 kB total on-chip memory space)*

Each value is given as a fraction of the value for the pure private memory case

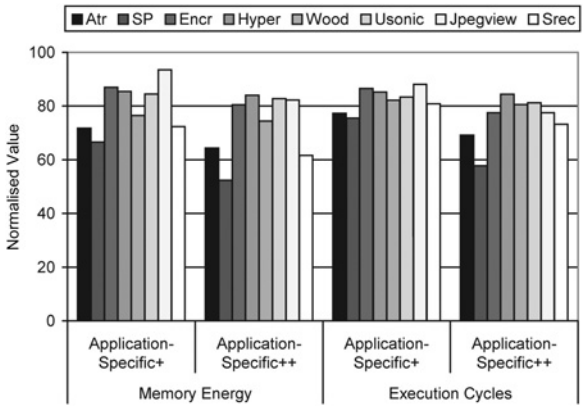

Figure 18 *Normalised energy consumption and execution cycle results with and without locality optimisation pass for the 'Application-Specific+' version under our default simulation parameters*

optimisation pass for the 'Application-Specific+' version under our default simulation parameters. The version with the locality optimisation pass is denoted using 'Application-Specific++'. We see from these results that optimising inter-phase locality brings reasonable improvements in both energy consumption and execution cycles in most of our benchmark codes. The average energy and performance improvements are about 6% over the case without locality optimisation, emphasising the importance of optimising data locality across the program phases.

5 Related work

Most of the prior work on memory synthesis and management focus on single processor-based systems [8–19]. In comparison, Abraham and Mahlke focus on an embedded system consisting of a VLIW processor, instruction cache, data cache and second-level unified cache [20]. Cotterell and Vahid [21] develop an automated simulation environment to find the best loop cache architecture for a given application and technology. In that work, a hierarchical approach of partitioning the system into its constituent components and evaluating each component individually is explored. Meftali *et al.* [22] focus on the memory allocation problem, which they formulate based on an integer linear programming model. Their solution permits one to obtain an optimal distributed shared memory architecture, minimising 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.* [23] 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. Li and Wolf [16] introduced a hardware/

doi: 10.1049/iet-cdt.2009.0089

software co-synthesis algorithm of distributed real-time systems that optimise the memory hierarchy along with the rest of the architecture. Dasygenis *et al.* [24] propose a formalised technique that uses data reuse, lifetime of the arrays of an application and application specific prefetching opportunities. Using these parameters, authors perform a trade-off exploration for different memory layer sizes. Issenin *et al.* [25], on the other hand, introduces a multiprocessor data reuse analysis technique to explore a wide range of customised memory hierarchy organisations with different sizes. In [26], authors implement an incremental technique for hierarchical memory size requirement estimation. The difference between our work and these is that we focus on a general approach based on a polyhedral tool (the Omega Library) driven analysis and on power consumption. In addition, our dynamic approach changes the memory configuration dynamically as we move from one phase of execution to another. In other words, we want each program phase to work with the most suitable memory configuration if it is possible to do so. Several prior studies [27, 28] discusses several advantages of chip multiprocessors over complex single-processor-based designs. Liu *et al.* [29] present an L2 cache partitioning approach that decides the L2 partitions allocated to processors at runtime based on dynamic L2 demands of processors. In comparison, we focus on software-managed on-chip memory partitioning and make use of a compiler-directed mechanism. Consequently, our partitioning strategies and the process of determining partitions are entirely different from [29].

6 Concluding remarks and future work

Deciding a suitable on-chip memory configuration is one of the most important aspects of designing an application-specific memory for chip multiprocessors. Unfortunately, several problems such as identifying and counting the number of data items shared across processors, determining the privately accessed data, and the fact that data sharing patterns can change from one phase of the execution to another make this a very hard problem to solve. This paper proposes two application-specific memory design algorithms. The first scheme aims at finding suitable allocation of on-chip memory space across processors. We attacked this problem using a two-step approach: (1) determining the amount of data that are shared by processors and the amount of data that are private to each processor, and (2) allocating memory space across private and shared data and over all processors. Our experimental analysis demonstrates that our approach generates better results than conventional architectures based on pure shared and pure private on-chip memories. The results also show that the energy benefits are consistent across a wide range of parameters. However, the main problem with this scheme is that the memory partitioning determined is valid throughout the entire execution of the application. The second scheme proposed in this paper is a dynamic on-chip memory management strategy that changes the memory allocation and bank sharing across processors during the course of execution according to a statically-determined schedule. The proposed approach is implemented using an optimising compiler and the Omega Library, and tested using a set of embedded applications. Our results indicate that the proposed technique not just only improves over conventional memories such as pure private and pure shared memories, but it also generates better results than an application-specific memory design methodology that uses the same design for the entire execution of the application. We plan to extend this study by allowing a general on-chip memory hierarchy (with multiple levels) and by developing accompanying code restructurings to achieve better memory energy savings. Work is also underway in porting our entire simulation infrastructure to an FPGA-based platform.

7 Acknowledgments

This research is supported in part by NSF grants CNS #0720645, CCF #0811687, CCF #0702519, CNS #0202007, CNS #0509251, by a grant from Microsoft Corporation, by a grant from IBM, and by a Marie Curie International Reintegration Grant within the 7th European Community Framework Programme.

8 References

[1] KELLY W., PUGH W.: 'Finding legal reordering transformations using mappings'. Proc. 7th Int. Workshop of Languages and Compilers for Parallel Computing (LCPC), 1995, pp. 107–124

[2] BANERJEE U.: 'Loop parallelization' (Kluwer Academic Publishers, 1994)

[3] KOELBEL C.H., LOVEMAN D.B., SCHREIBER R.S.: 'The high performance Fortran handbook' (MIT Press, 1993)

[4] 'The openMP application program interface', version 2.5, http://www.openmp.org/mp-documents/spec25.pdf, May 2005

[5] WILSON R.P., FRENCH R.S., WILSON C.S., *ET AL.*: 'Suif: an infrastructure for research on parallelizing and optimizing compilers', *SIGPLAN Not.*, 1994, **29**, (12), pp. 31–37

[6] REINMAN G., JOUPPI N.P.: 'Cacti 2.0: an integrated cache timing and power model'. Compaq, Technical report, February 2000

[7] BERKELAAR M., EIKLAND K., NOTEBAERT P.: 'lp solve: open source (mixed-integer) linear programming system'. Version 5.0.0.0., 2004

[8] SUH G.E., RUDOLPH L., DEVADAS S.: 'Dynamic partitioning of shared cache memory', *J. Supercomput.*, 2004, **28**, (1), pp. 7–26

[9] RANGANATHAN P., ADVE S.V., JOUPPI N.P.: 'Reconfigurable caches and their application to media processing', *SIGARCH Comput. Archit. News*, 2000, **28**, (2), pp. 214–224

[10] ANGIOLINI F., BENINI L., CAPRARA A.: 'Polynomial-time algorithm for on-chip scratchpad memory partitioning'. Proc. Int. Conf. on Compilers, Architecture and Synthesis for Embedded Systems, 2003, pp. 318–326

[11] UDAYAKUMARAN S., BARUA R.: 'Compiler-decided dynamic memory allocation for scratch-pad based embedded systems'. Proc. Int. Conf. on Compilers, Architecture and Synthesis for Embedded Systems, CASES'03, 2003, pp. 276–286

[12] CAO Y., TOMIYAMA H., OKUMA T., YASUURA H.: 'Data memory design considering effective bitwidth for low-energy embedded systems'. Proc. 15th Int. Symp. on System Synthesis, ISSS'02, 2002, pp. 201–206

[13] PANDA P.R., DUTT N.D., NICOLAU A.: 'Architectural exploration and optimization of local memory in embedded systems'. Proc. 10th Int. Symp. on System Synthesis, ISSS'97, 1997, pp. 90–97

[14] RAMACHANDRAN A., JACOME M.F.: 'Xtream-fit: an energy-delay efficient data memory subsystem for embedded media processing'. Proc. 40th Conf. on Design Automation, DAC'03, 2003, pp. 137–142

[15] SHIUE W.-T., CHAKRABARTI C.: 'Memory exploration for low power, embedded systems'. Proc. 36th ACM/IEEE Conf. on Design Automation, DAC'99, 1999, pp. 140–145

[16] LI Y., WOLF W.: 'Hardware/software co-synthesis with memory hierarchies', *IEEE Trans. Comput. – Aided Des. Integr. Circuit Syst.*, 1999, **18**, pp. 1405–1417

[17] PANDA P.R., CHITTURI L.: 'An energy-conscious algorithm for memory port allocation'. Proc. 2002 IEEE/ACM Int. Conf. on Computer-aided Design, ICCAD'02, 2002, pp. 572–576

[18] CATTHOOR F., DE GREEF E., SUYTACK S.: 'Custom memory management methodology: exploration of memory organisation for embedded multimedia system design' (Kluwer Academic Publishers, Norwell, MA, 1998)

[19] KANDEMIR M., CHOUDHARY A.: 'Compiler-directed scratch pad memory hierarchy design and management'. Proc. 39th Conf. on Design Automation, DAC'02, 2002, pp. 628–633

[20] ABRAHAM S.G., MAHLKE S.A.: 'Automatic and efficient evaluation of memory hierarchies for embedded systems'. Proc. 32nd Annual ACM/IEEE Int. Symp. on Microarchitecture, Haifa, Israel, 1999, pp. 114–125

[21] COTTERELL S., VAHID F.: 'Tuning of loop cache architectures to programs in embedded system design'. Proc. 15th Int. Symp. on System Synthesis, Kyoto, Japan, 2002, pp. 8–13

[22] MEFTALI S., GHARSALLI F., ROUSSEAU F., JERRAYA A.A.: 'An optimal memory allocation for application-specific multiprocessor systemon-chip'. Proc. 14th Int. Symp. on Systems Synthesis, ISSS'01, 2001, pp. 19–24

[23] GHARSALLI F., MEFTALI S., ROUSSEAU F., JERRAYA A.A.: 'Automatic generation of embedded memory wrapper for multiprocessor soc'. Proc. 39th Conf. on Design Automation, DAC'02, 2002, pp. 596–601

[24] DASYGENIS M., BROCKMEYER E., DURINCK B., CATTHOOR F., SOUDRIS D., THANAILAKIS A.: 'A memory hierarchical layer assigning and prefetching technique to overcome the memory performance/energy bottleneck'. Proc. Conf. on Design, Automation and Test in Europe, DATE'05, IEEE Computer Society, 2005, pp. 946–947

[25] ISSENIN I., BROCKMEYER E., DURINCK B., DUTT N.: 'Multiprocessor system-on-chip data reuse analysis for exploring customized memory hierarchies'. Proc. 43rd Ann. Conf. on Design automation, DAC'06, New York, NY, USA, 2006, pp. 49–52

[26] HU Q., VANDECAPPELLE A., PALKOVIC M., KJELDSBERG P.G., BROCKMEYER E., CATTHOOR F.: 'Hierarchical memory size estimation for loop fusion and loop shifting in data-dominated applications'. Proc. 2006 Conf. on Asia South Pacific Design Automation, ASP-DAC'06, 2006, pp. 606–611

[27] KRISHNAN V., TORRELLAS J.: 'A chip-multiprocessor architecture with speculative multithreading', *IEEE Trans. Comput.*, 1999, **48**, (9), pp. 866–880

[28] NAYFEH B.A., HAMMOND L., OLUKOTUN K.: 'Evaluation of design alternatives for a multiprocessor microprocessor'. Proc. 23rd Ann. Int. Symp. on Computer Architecture, ISCA'96, 1996, pp. 67–77

[29] LIU C., SIVASUBRAMANIAM A., KANDEMIR M.: 'Organizing the last line of defense before hitting the memory wall for CMPs'. Proc. 10th International Symposium on High Performance Computer Architecture, 2004, p. 176

doi: 10.1049/iet-cdt.2009.0089