An Integer Linear Programming Based Approach to Simultaneous Memory Space Partitioning and Data Allocation for Chip Multiprocessors*

O. Ozturk, G. Chen, M. Kandemir
Computer Science and Engineering Department
Pennsylvania State University
University Park, PA 16802, USA
{ozturk, gchen, kandemir}@cse.psu.edu

M. Karakoy
Department of Computing
Imperial College
London, SW 2AZ, UK
m.karakoy@ic.ac.uk

Abstract

The trends in advanced integrated circuit technologies require us to look for new ways to utilize large numbers of gates and reduce the effects of high interconnect delays. One promising research direction is chip multiprocessors that integrate multiple processors on the same die. Among the components of a chip multiprocessor, its memory subsystem is maybe the most critical one, since it shapes both power and performance characteristics of the resulting design. Motivated by this observation, this paper addresses the problem of decomposing (partitioning) on-chip memory space across parallel processors and allocating data across memory components in an integrated manner. In the most general case, the resulting memory architecture is a hybrid one, where some memory components are accessed privately, whereas the others are shared by two or more processors. The proposed approach for achieving this has two complementary components: an optimizing compiler and an ILP (integer linear programming) solver. The role of the compiler in this approach is to analyze the application code and detect the interprocessor data sharing patterns, given the loop parallelization information. The job of the ILP solver, on the other hand, is to determine the sizes of the on-chip memory components, how these memory components are shared across multiple processors in the system, and what data each component holds. In other words, we address the problem of integrated memory space partitioning and data allocation for chip multiprocessors.

1 Introduction

A chip multiprocessor integrates multiple processors on the same die. Recent research [18, 11, 13, 14, 19] discusses several advantages of these architectures over complex single-processor based designs. These advantages include capability of exploiting both high level (loop/thread level) and low level (instruction level) parallelism, better performance and power consumption profiles, and easier verification. In particular, it has been reported [17] that, on applications with high-level parallelism (e.g., embedded multimedia codes that make frequent use of large arrays), a chip multiprocessor can perform better than a wide superscalar architecture.

A critical component of a chip multiprocessor is its memory subsystem. This is because both power and performance behavior of a chip multiprocessor is largely shaped by its on-chip memory [6, 16]. While it is possible to employ conventional memory designs such as pure private memory or pure shared memory, such designs are very general and rigid, and may not generate the best behavior for a given embedded application. Our belief is that, for embedded systems that repeatedly execute the same application, it makes sense to design a customized, software-managed on-chip memory architecture. Such a memory architecture should be a hybrid one that contains both private and shared components. Figure 1 depicts the high-level views of pure private, pure shared, and sample hybrid memory architectures. Note that, in the hybrid architecture case, while some processors have private memories, others do not have one. Similarly, the different processor groups can share memory in different fashions. For example, a memory component can be shared by two processors, whereas another component can be shared by three processors.

Designing such a customized hybrid memory architecture is not trivial because of at least three main reasons. First, since the memory architecture to be designed changes from one application to another, a hand-waived approach is not suitable, as it can be extremely time consuming and error-prone to go through the same complex process each time we want to design a memory system for a new application. Therefore, we need an automated strategy that comes up with the most suitable design for a given application. Second, the design of such a memory needs to be guided by a tool that can extract the data sharing exhibited by the application at hand. After all, in order to decide how the different memory components need to be shared by parallel processors, one needs to capture the data sharing patterns across the processors. Third, data allocation in a hybrid memory system is not a trivial problem, and should be carried out along with data partitioning if we want obtain the best results.

*This work is supported in part by NSF Career Awared #0093082 and a fund from GSRC.

Proceedings of the 2006 Emerging VLSI Technologies and Architectures (ISVLSI'06)
0-7695-2533-4/06 $20.00 © 2006 IEEE

IEEE COMPUTER SOCIETY

Authorized licensed use limited to: Univ of Calif Irvine. Downloaded on September 20, 2009 at 15:11 from IEEE Xplore. Restrictions apply.

In this paper, we propose a strategy for designing application-specific on-chip hybrid memories for chip multiprocessors that employs both an optimizing compiler and an ILP (integer linear programming) solver (see Figure 2). The role of the compiler in this approach is to analyze the application code and detect the data sharing patterns across processors, given the loop parallelization information. The job of the ILP solver, on the other hand, is to determine the sizes of the memory components, how these memory components are shared across multiple processors in the system, and what data each memory component is to hold. Note that, the ILP based solution can be used not only for designing hybrid memories, but also as an upper bound against which future heuristic solutions can be compared. Our focus is on array-based application codes that occur frequently in embedded image/video processing. It needs also be noted that, our approach can be targeted at different objectives such as maximizing performance, reducing power/energy consumption, and minimizing the memory space occupied.

The rest of this paper is organized as follows. Section 2 presents the compiler analysis necessary to identify and characterize the data sharings across parallel processors. Section 3 gives the details of our ILP formulation. Section 4 gives an example. Section 5 concludes the paper with a summary.

2 Compiler Analysis for Identifying Shared and Privately-Accessed Data

As mentioned earlier, our ILP solver takes as input the data accessed by each processor and the data shared by processor groups. While there are several ways of obtaining this data (e.g., through simulation or static analysis of the application code), in this work we employ a compiler-based approach. More specifically, the compiler analyzes the application source code and extracts the interprocessor data sharing information. To achieve this, the proposed compiler support employs a polyhedral tool called the Omega Library [10]. Basically, the Omega Library provides several functions that operate on Presburger formulas. Presburger formulas are 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 $\forall$). The Omega Library manipulates integer tuple relations and sets, which are described using Presburger formulas. Specifically, the conditions describing a set or tuple can be described by a Presburger formula. Relations and sets can be combined using functions (operators) such as composition, intersection, union, and difference.

In our work, we express the set of elements accessed by processors and the set of elements shared among processors using Presburger formulas. As an example, consider the nested loop depicted in Figure 3. The iteration space of this loop nest (i.e., the set of iteration points that will be executed by the nest) can be expressed using the following Presburger formula:

$$\mathcal{J} = \{(j_1, j_2) \quad | \quad L_1 \leq j_1 \leq U_1 \quad \wedge \quad L_2 \leq j_2 \leq U_2\}.$$

However, when the loop nest is parallelized[1], each processor typically executes a subset of the iteration points in the nest. In the loop nest shown above, assuming that the outer loop (j_1) is parallelized across P processors, the pth processor ($0 \leq p < P$) is assigned the iterations captured by the following Presburger set:

$$\mathcal{J}(p) = \{(j_1, j_2) \quad | \quad (L_1 + p(U_1 - L_1 + 1)/P \leq j_1 < L_1 + (p+1)(U_1 - L_1 + 1)/P) \quad \wedge \quad (L_2 \leq j_2 \leq U_2)\}.$$

Note that,

$$\mathcal{J} = \bigcup_{p \in \{P\}} \mathcal{J}(p),$$

where $\bigcup$ denotes the set union operator and $\{P\}$ represents the set of processors in the system. Note also that, we assumed, for simplicity, $(U_1 - L_1 + 1)$ is evenly divided by P.

The set of array elements accessed by processor p (based on this parallelization) can be calculated as the union of the set of elements accessed by each reference within the loop nest. In mathematical terms, for our example nest in Figure 3, we have:

$$\mathcal{D}(p, X) = \mathcal{D}(p, X[j_1, j_1 + j_2]) \cup \mathcal{D}(p, X[j_2 - 1, j_1 + 3]) \cup \mathcal{D}(p, X[j_2 + 2, j_2 + j_1]),$$

where

$$\mathcal{D}(p, X[j_1, j_1 + j_2]) = \{(d_1, d_2) \quad | \quad \exists (j_1, j_2) \text{ such that } (d_1 = j_1 \wedge d_2 = j_1 + j_2) \quad \wedge \quad (j_1, j_2) \in \mathcal{J}(p)\}$$

$$\mathcal{D}(p, X[j_2 - 1, j_1 + 3]) = \{(d_1, d_2) \quad | \quad \exists (j_1, j_2) \text{ such that } (d_1 = j_2 - 1 \wedge d_2 = j_1 + 3) \quad \wedge \quad (j_1, j_2) \in \mathcal{J}(p)\}$$

$$\mathcal{D}(p, X[j_2 + 2, j_2 + j_1]) = \{(d_1, d_2) \quad | \quad \exists (j_1, j_2) \text{ such that } (d_1 = j_2 + 2 \wedge d_2 = j_2 + j_1) \quad \wedge \quad (j_1, j_2) \in \mathcal{J}(p)\}.$$

We can also express the array elements shared among a set of processors using the set intersection operator. For example, let $\{P'\}$ be a subset of $\{P\}$, i.e., $\{P'\} \subseteq \{P\}$. For our example, the set of data elements shared by all the processors in $\{P'\}$ can be expressed as:

$$\mathcal{S}(\{P'\}, X) = \bigcap_{p \in \{P'\}} \mathcal{D}(p, X).$$

As will be discussed later in detail, the set of elements shared among processors helps us determine the sizes of the on-chip memory components shared among processors. To determine the size of the on-chip private memories, on the other hand, we need the set of array elements accessed exclusively by processor p. This can be expressed as follows:

$$E(p, X) = \mathcal{D}(p, X) \quad \setminus \bigcup_{[p \in \{P'\}] \wedge [\{p\} \neq \{P'\}]} \mathcal{S}(\{P'\}, X),$$

where $\setminus$ denotes the set subtraction (set difference) operator. In informal terms, what this last expression says that an

[1] In this paper, we do not assume a specific loop parallelization strategy. A loop nest can be parallelized either through user-specified annotations or via automatic compiler analysis.

IEEE COMPUTER SOCIETY
Authorized licensed use limited to: Univ of Calif Irvine. Downloaded on September 20, 2009 at 15:11 from IEEE Xplore. Restrictions apply.

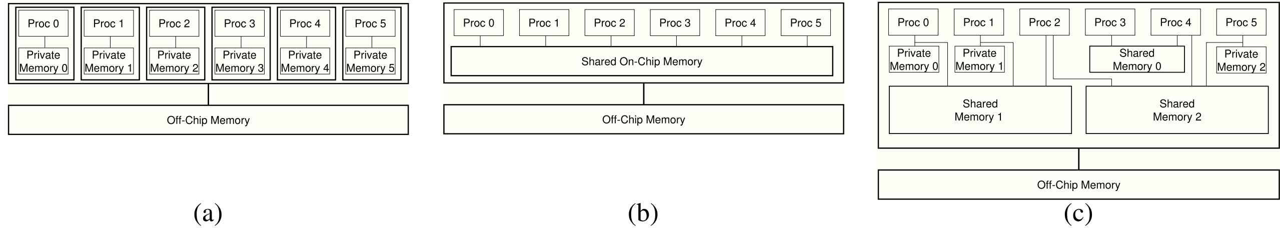

Figure 1. (a) Pure private memory. (b) Pure shared memory. (c) Hybrid memory. Although not shown for clarity, in (a) and (c) there is an on-chip interconnect to allow a processor access a data item which resides in a memory that is not directly accessible by that processor.

array element belongs to the set $E(p, X)$ if and only if this item is not accessed by any processor other than p.

An important point to note here is that, while the analysis above is given in terms of a single nest accessing a single array, it is straightforward to extend it to multiple nest and multiple array cases. For example, if a loop nest accesses q different arrays, namely, X_0, X_1, ..., X_{q-1}, the total set of elements accessed by processor p can be computed as: $\mathcal{D}(p) = \mathcal{D}(p, X_0) \cup \mathcal{D}(p, X_1) \cup ... \cup \mathcal{D}(p, X_{q-1})$. The other sets can be computed in a similar fashion. A similar extension can be written for the multiple nest case as well.

Capturing only the array elements shared or privately accessed by the processors is sufficient to perform memory partitioning, but not sufficient for data allocation across the memory components. This is because it may be necessary in many cases to store a data item in a memory location that belongs to a component that is not shared by a processor that accesses that element. In such a case, we consume energy in both memory access and extra interconnect access if the processor in question wants to access that item. To decide which data needs to be stored remotely and which data locally, we need a mechanism to *rank* the different data items based on their importances (criticalities). For example, from the viewpoint of processor p, not all the data items in $\mathcal{S}(\{p, p'\}, X)$, the set of shared elements between processors p and p', have the same importance; some of them can be more important than the others. In this work, we use the *number of accesses* as the metric using which we can rank the different data items. While a polyhedral analysis, similar to the one conducted above, can be used for capturing the number of accesses to each individual data item, the associated overheads can be too much to tolerate. Instead, we calculate the number of accesses at a set (of data items) granularity. As an example, for the loop nest discussed above, we compute the number of accesses to the elements in sets $E(p, X)$ and $\mathcal{S}(\{P'\}, X)$, for all $p \in \{P\}$ and for all $\{P'\} \subseteq \{P\}$. Note that, we can use polyhedral arithmetic to achieve this. For example, the loop iterations that access the elements in $\mathcal{S}(\{P'\}, X)$ can be captured as:

$$\begin{aligned}\mathcal{J}(\{P'\}, X) \quad = \quad & \{(j_1, j_2) \quad | \quad \exists (d_1, d_2) \quad \text{such that} \\ & (j_1, j_2) \in \mathcal{J}(p) \wedge ((d_1 = j_1 \wedge d_2 = j_1 + j_2) \vee \\ & (d_1 = j_2 - 1 \wedge d_2 = j_1 + 3) \vee (d_1 = j_2 + 2 \wedge \\ & d_2 = j_2 + j_1)) \wedge \\ & (\forall p \in \{P'\} : (d_1, d_2) \in \mathcal{D}(p, X))\}.\end{aligned}$$

One can similarly compute a set $\mathcal{J}(p, X)$, the set of loop iterations that access the elements in $E(p, X)$. An important issue that needs to be clarified at this point is how one can enumerate and count the elements in the Presburger sets defined above, i.e., in sets such as $E(p, X)$, $\mathcal{S}(\{P'\}, X)$, $\mathcal{J}(p, X)$, and $\mathcal{J}(\{P'\}, X)$. This is important since our ILP solver needs the number of elements in these sets, not the sets themselves. We address this problem using the "codegen" utility provided by the Omega Library. Codegen generates code to traverse the points in a given Presburger set in lexicographical order. After generating this code, what we do is to insert a counter variable in the code that keeps track of the number of points in the code, and execute the resulting code at compile time. The final value of the counter variable at the end of this execution gives us the number we want to determine (i.e., the number of elements in the set). A similar approach has been employed for an entirely different problem in [7]. It is to be noted that, the elements in the sets discussed above can be counted using other existing methods as well such as [5] and [1], among the others. In the rest of this paper, we use $|E(p)|$ and $|\mathcal{S}(\{P'\})|$ to denote, respectively, the set of privately accessed elements by processor p and the set elements shared by processors in $\{P'\}$. Similarly, $|\mathcal{J}(p, X)|$ and $|\mathcal{J}(\{P'\}, X)|$ give the number of elements in sets $\mathcal{J}(p, X)$ and $\mathcal{J}(\{P'\}, X)$, respectively.

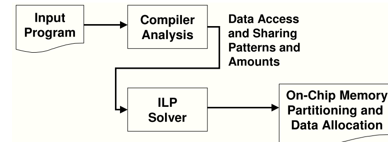

Figure 2. High-level view of our approach.

$$\begin{aligned}& for(j_1 = L_1; j_1 \leq U_1; j_1 + +) \\ & \;\; for(j_2 = L_2; j_2 \leq U_2; j_2 + +) \\ & \;\;\;\; \cdots = X[j_1, j_1 + j_2] + X[j_2 - 1, j_1 + 3] + X[j_2 + 2, j_2 + j_1];\end{aligned}$$

Figure 3. An example loop nest.

3 ILP Formulation

In this paper, 0-1 ILP is used to determine the sizes of the memory components and how they are shared across multiple processors in the system. Table 1 gives the constant terms used in our ILP formulation. Note that, the sizes given in this table and used in the following discussion are

IEEE COMPUTER SOCIETY
Authorized licensed use limited to: Univ of Calif Irvine. Downloaded on September 20, 2009 at 15:11 from IEEE Xplore. Restrictions apply.

Constant	Definition
P	Number of processors
L	Total memory size in terms of slices
D	Maximum possible number of data sets ($D = 2^P - 1$)
E_{Comm}	Communication energy for a non-local data access
$E_{port,s}$	Energy consumed by accessing a memory component with $port$ number of ports and of size s
$F_{p,d}$	Frequency of accesses to data set d by processor p
S_d	Size of data set d
$Local_{m,p}$	Indicates whether memory component m is directly accessible by p
$ports(m)$	Number of ports that memory component m has

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

in terms of units (*slices*), which represents the minimum amount of space that can be allocated to a memory component (i.e., the unit of allocation).

Our objective is to partition the available on-chip memory space into memory components that can be shared among different processors or privately accessed by a processor and perform data allocation across these memory components to minimize the overall energy consumption. We determine the existence and sizes of memory components based on the data access frequency of the processors and the sizes of the data sets using 0-1 variables. For each possible memory component size, we define 0-1 variables. The sizes of memory components are restricted by the total data size. There are possibly $2^P - 1$ memory components on the chip multiprocessor, where P is the number of processors. For example, if there are two processors (p_0 and p_1), possible memory components can be a private memory for p_0, a private memory for p_1, and a shared memory that can be accessed by p_0 and p_1. Therefore, the available on-chip memory space can potentially be partitioned into these three (2^2-1) memory components in an energy minimizing fashion. Note that, it is possible, in the final design (memory partitioning), to have one of these memory components alone, or only two of them, or all of them together. That is, the final on-chip memory space partitioning determined by our approach can contain any subset of these three components. Similarly, there are possibly $D = 2^P - 1$ data sets. The size of a data set, denoted by S_d in Table 1, is determined by the number of elements in the data set. For example, if we have three processors (denoted p_0, p_1, and p_2), we have: $S_1 = |E(p_0)|$, $S_2 = |E(p_1)|$, $S_3 = |E(p_2)|$, $S_4 = |\mathcal{S}(p_0, p_1)|$, $S_5 = |\mathcal{S}(p_0, p_2)|$, $S_6 = |\mathcal{S}(p_1, p_2)|$, and $S_7 = |\mathcal{S}(p_0, p_1, p_2)|$. Similarly, the access frequency of a processor p to a data set d, denoted by $F_{p,d}$ in Table 1, is obtained from $|\mathcal{J}(p, X)|$ and $|\mathcal{J}(\{P'\}, X)|$ defined earlier.

We can use 0-1 variables to specify the size ($size$) of a memory component. Specifically, we have:

- $size_{m,s}$: indicates whether memory component m is of size s.

We use a variable for each one of the possible sizes. If this 0-1 variable is 1, this indicates that the corresponding memory component size is s. If this size is 0, then we conclude that this memory component does not exist.

We use another 0-1 variable for assigning each data set to memory component(s):

- map_{d,d_s,m,m_s} : indicates whether data set d of size d_s is located in memory component m of size m_s.

Note that, we allow a data set to be divided among the different memory components. If we were to restrict a data set to be located only in one memory component, removing the second parameter in the subscript above (d_s), would be sufficient.

If a data set (d_s) does not reside in one of the memory components that is directly accessible by the processor, then accessing that data set would incur an extra on-chip communication energy due to accessing the interconnect. To capture the communication energy, we use $comm_{p,d,d_s,m,m_s}$:

- $comm_{p,d,d_s,m,m_s}$: indicates whether accessing the data set d of size d_s located in memory component m of size m_s by processor p would require communication cost.

Each processor's (p) energy consumption due to accesses to data sets can be identified using a variable A_p which is defined as follows:

- A_p : the energy consumed by processor p due to data accesses.

Also, the energy consumption due to interconnect accesses is captured by the $C_{p,m}$ variable.

- $C_{p,m}$: the energy consumed by processor p due to communication to access data in memory component m.

It should be noted that, access energy A_p and communication energy $C_{p,m}$ are not 0-1 variables. They are simply used to calculate the total energy consumption. After defining the variables in our ILP formulation, now we explain the necessary constraints.

The total memory space (L) should be equal to the sum of the sizes of the individual memory components. This can be captured as:

$$L = \sum_{i=1}^{2^P-1} \sum_{j=0}^{L} size_{i,j} \times j \qquad (1)$$

In this expression, index variable i iterates over the $2^P - 1$ memory components. On the other hand, index variable j iterates over the possible sizes from 0 up to L.

A memory component can have one and only one size. We capture this constraint as follows:

$$\sum_{j=0}^{L} size_{m,j} = 1, \quad \forall m. \qquad (2)$$

A data set (d) can be divided among memory components:

$$\sum_{j=0}^{S_d} \sum_{k=1}^{2^P-1} \sum_{l=0}^{L} map_{d,j,k,l} \times j = S_d, \quad j \le l, \quad \forall d. \qquad (3)$$

In the above formulation, index variable j iterates over the possible data set sizes from 0 up to S_d (the data set size). On

IEEE COMPUTER SOCIETY
Authorized licensed use limited to: Univ of Calif Irvine. Downloaded on September 20, 2009 at 15:11 from IEEE Xplore. Restrictions apply.

the other hand, k iterates over the memory components, and similarly, l is used to identify the size of the corresponding memory component. The sum of these data set portions should be equal to the total data set size (S_d).

A data set (d) can at most have one size allocated within a memory component (m). Although this constraint does not affect the result of the ILP, it prevents having two separate space assignments for the same data set within a memory component. For example, instead of two memory spaces with sizes s_1 and s_2, with this constraint the ILP solution will result in a single memory space of $s_1 + s_2$ within the same memory component.

$$\sum_{j=0}^{S_d} map_{d,j,m,l} \leq 1, \;\; j \leq l, \;\; \forall d, m, m_s. \tag{4}$$

If a memory component is used by a data set, this memory component has to exist, which can be captured by:

$$size_{m,m_s} \geq map_{d,d_s,m,m_s}, \;\; d_s \leq m_s, \;\; \forall d, d_s, m, m_s. \tag{5}$$

The total data size stored in a memory component must be less than or equal to the size of the memory component itself:

$$\sum_{i=1}^{D} \sum_{j=0}^{S_d} map_{i,j,m,m_s} \times j \leq m_s, \;\; \forall m, m_s. \tag{6}$$

In this formulation, i iterates over the data sets, whereas, j iterates over the possible data set sizes.

A communication cost will be incurred, if the data is mapped to a memory component (m) and the memory component is not local to the processor (p) accessing it (i.e., it is not one of the components to which p has a direct access). As explained earlier, we use $comm_{p,d,d_s,m,m_s}$ to denote a 0-1 variable that captures the existence of communication. We have:

$$comm_{p,d,d_s,m,m_s} \geq map_{d,d_s,m,m_s} - Local_{m,p}, \quad d_s \leq m_s, \forall p, d, d_s, m, m_s.$$

In the above expression, $Local_{m,p}$ is a parameter given to the ILP solver based on the memory component in question. For example, if there are two processors (p_0 and p_1), the possible memory components can be m_0 (a private memory for p_0), m_1 (a private memory for p_1), and m_2 (a shared memory between p_0 and p_1). For m_0, this parameter will be given as 0 for p_1. Similarly, for m_1, it will be set to 0 for p_0. On the other hand, for m_2, it will be set to 1 for both processors.

Having specified the necessary constraints in our ILP formulation, we next give our objective function. In our execution model, there are two components of the total memory energy consumption:

- *access:* the energy consumed when a memory component is accessed.
- *communication:* the extra interconnect energy consumed when a remote memory component is accessed.

Each processor's memory access cost, A_p can be formulated as follows:

$$A_p = \sum_{i=1}^{D} \sum_{j=0}^{S_d} \sum_{k=1}^{2^P-1} \sum_{l=0}^{L} map_{i,j,k,l} \times F_{p,i} \times j \times E_{ports(k),l} \;\; j \leq l, \;\; \forall p. \tag{7}$$

Data Set	Size	Proc. 0	Proc. 1	Proc. 2	Proc. 3
D_0	2K	10%	-	-	-
D_1	1K	-	5%	-	-
$D_{0,1}$	4K	5%	15%	-	-
$D_{2,3}$	3K	-	-	5%	10%
$D_{0,1,2,3}$	5K	5%	10%	10%	25%

(a)

$\Rightarrow$

Location
M_0
M_1
$M_{0,1}$
$M_{2,3}$
$M_{1,2,3}$

(b)

Table 2. (a) An example data set with access frequencies. (b) The resulting locations (placement) for the data sets.

Each processor's communication cost due to accessing the interconnect can be formulated using $C_{p,d}$:

$$C_{p,d} = \sum_{j=0}^{S_d} \sum_{k=1}^{2^P-1} \sum_{l=0}^{L} comm_{p,d,j,k,l} \times F_{p,d} \times j \times E_{Comm}, \;\; j \leq l, \;\; \forall p, d. \tag{8}$$

In the last two expressions, indices i, j, k, and l iterate over the data sets, data set sizes, memory components, and memory component sizes, respectively. $F_{p,i}$ denotes the frequency of the accesses to data set i by processor p. $E_{ports(k),l}$ is the unit access energy consumption for a memory component of a size l with $ports(k)$ number of ports. $ports(k)$, the number of ports required for the memory component, is obtained based on the number of processors accessing it (k). The unit communication energy for a remote access is E_{Comm}. Using these two cost expressions (A_p and $C_{p,d}$), we can express the total energy consumption due to memory accesses (E) as follows:

$$E = \sum_{i=1}^{P} A_i + \sum_{i=1}^{P} \sum_{j=1}^{D} C_{i,j}. \tag{9}$$

Based on this formulation, our 0-1 ILP problem can formally be defined as one of "minimizing *E* under constraints (1) through (8)."

Let us explain the operation of our software-managed hybrid memory architecture. There are three scenarios for the outcome of a memory access (request) in our hybrid on-chip memory architecture, depending on where the requested item is located:

- *Local Hit:* When the processor finds the data in one of the memory components it has direct access to.
- *Remote Hit:* In this case, the lookup amongst its assigned component(s) fails, but the data is found in another (on-chip) component that is not directly connected to the processor that issued the memory request.
- *On-Chip Miss:* In this case, the data is not in any of the on-chip components, and requires an off-chip access. The access cost in this case will involve the cost of the off-chip access.

4 Example

An example data access frequency and data set size information for a case with 4 processors is given in Table 2(a). We assume, for the sake of explanation, that the data sets can exactly fit into the available on-chip memory space (15K). The processors that share a data set are indicated in

0-7695-2533-4/06 $20.00 © 2006 IEEE
IEEE COMPUTER SOCIETY
Authorized licensed use limited to: Univ of Calif Irvine. Downloaded on September 20, 2009 at 15:11 from IEEE Xplore. Restrictions apply.

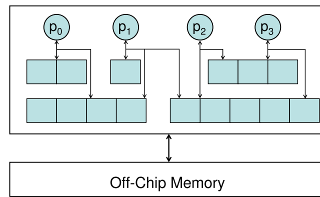

Figure 4. The hybrid memory.

the subscript of that data set. For example, D_0 is a data set privately accessed by processor p_0. Similarly, $D_{2,3}$ denotes a data set shared by processors p_2 and p_3. The second column of Table 2 gives the size of each data set. The next four columns indicate the access frequencies (of processors). In this example, we assume that if a memory component is private to a processor, it has a single port. If the memory component is shared by two or three processors, it has two ports. Finally, if it is shared by more than three processors, it has three ports.

In Figure 4, the resulting hybrid on-chip memory partitioning returned by our approach is shown. Processors p_0 and p_1 have access to both private and shared memory components. In comparison, the other processors have access to shared components only. The location (memory component) of each data set is shown in Table 2(b). As it can be seen, except for the data set $D_{0,1,2,3}$, all of the data sets are in the memory components in accordance with their data set sharing information. In other words, every processor is directly connected to the memory from which it needs data. The only exception is that the memory component that holds data set $D_{0,1,2,3}$ is accessed by only three processors instead of all four. The main reason for this is that, allowing one more processor access to that component would increase the overall energy consumption in this example (we do not give here explicitly the energy values used in the example). Overall, except for accesses by processor p_0 to the memory component that hold data set $D_{0,1,2,3}$, all the memory accesses are local. This example shows how our approach comes up with a hybrid on-chip memory architecture.

5 Concluding Remarks

Chip multiprocessors are suitable for executing data-intensive embedded applications with source-level parallelism. Ensuring that most of data accesses are satisfied from on-chip memories is a critical problem for chip multiprocessors, as cost of an off-chip access is very high. Particularly, multiple cores that need to access the off-chip memory system may contend with each other for the same buses/pins to get there. While it is possible to structure on-chip memory space as shared memory or private memory, each of these has its own drawbacks. An important observation here is that, to reach minimum energy consumption, both memory space partitioning and data allocation need to be optimized in a coordinated manner. In an attempt to achieve lower power consumption than the conventional on-chip memory architectures, this paper proposes an ILP-based strategy that comes up with an application-specific hybrid memory architecture that has both shared and private components. Our strategy also determines the optimal data allocation (placement) for the resulting hybrid memory.

References

[1] R. Bagnara. The Parma Polyhedra Library. *Seminar given in Departement de Mathematiques et Informatique in St Denis de La Reunion,* France, Indian Ocean, May 2002.

[2] L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: a scalable architecture based on single chip multiprocessing. In *Proc. ISCA,* 2000.

[3] CACTI 3.0. http://research.compaq.com/wrl/people/jouppi/CACTI.html

[4] Berkeley Predictive Technology Model, U.C. Berkeley, http://www-devices.eecs.berkeley.edu/ptm/

[5] P. Clauss. Counting solutions to linear and nonlinear constraints through Ehrhart polynomials: applications to analyze and transform scientific programs. In *Proc. ICS,* May 1996.

[6] F. Gharsalli, S. Meftali, F. Rousseau, and A. A. Jerraya. Automatic Generation of Embedded Memory Wrapper for Multiprocessor SoC. In *Proc. DAC,* New Orleans, Louisiana, 1999.

[7] S. Ghosh, M. Martonosi, and S. Malik. Precise miss analysis for program transformations with caches of arbitrary associativity. In *Proc. ASPLOS,* San Jose, CA, October 1998.

[8] U. Ko, P. Balsara, and A. Nanda. Power and Performance Optimization for On-Chip Multi Level Cache Hierarchies in Microprocessors. *IEEE Transactions on VLSI Systems,* pp. 299–308, June 1998.

[9] M. Kandemir, O. Ozturk, and M. Karakoy. Dynamic on-chip memory management for chip multiprocessors, by In *Proc. CASES,* Washington D.C., September 2004.

[10] W. Kelly and W. Pugh. Finding legal reordering transformations using mappings. In *Proc. LCPC Workshop,* pp. 107–124, 1994.

[11] V. Krishnan and J. Torrellas. A Chip Multiprocessor Architecture with Speculative Multi-threading. *IEEE Transactions on Computers, Special Issue on Multi-threaded Architecture,* September 1999.

[12] C. Liu, A. Sivasubramaniam, and M. Kandemir. Organizing the Last Line of Defense Before Hitting the Memory Wall for CMPs. In *Proc. the International Symposium on High-Performance Computer Architecture,* Madrid, Spain, February 2004.

[13] MAJC-5200. http://www.sun.com/microelectronics/MAJC/5200wp.html

[14] MP98: A Mobile Processor. http://www.labs.nec.co.jp/MP98/top-e.htm.

[15] W. Pugh and D. Wonnacott. Constraint-based array dependence analysis. *ACM TOPLAS,* 20(3):635-678, May 1998.

[16] S. Meftali, F. Gharsalli, F. Rousseau, and A. A. Jerraya. An optimal memory allocation for application-specific multiprocessor system-on-chip. In *Proc. ISSS,* Montreal, Canada, 2001.

[17] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The Case for a Single Chip Multiprocessor. In *Proc. ASPLOS,* 1996.

[18] S. Richardson. MPOC: a chip multiprocessor for embedded systems. In *HP Laboratories Technical Report HPL-2002-186,* Palo Alto, CA, July 2002.

[19] S. F. Smith. Performance of a GALS single-chip multiprocessor. In *Proc. PDPTA,* 2004.

[20] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic partitioning of shared cache memory. *Journal of Supercomputing*, 2002.

Proceedings of the 2006 Emerging VLSI Technologies and Architectures (ISVLSI'06)
0-7695-2533-4/06 $20.00 © 2006 **IEEE**
IEEE COMPUTER SOCIETY

Authorized licensed use limited to: Univ of Calif Irvine. Downloaded on September 20, 2009 at 15:11 from IEEE Xplore. Restrictions apply.