View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by IntechOpen

We are IntechOpen, the world's leading publisher of Open Access books

Built by scientists, for scientists

4,800
Open access books available

122,000
International authors and editors

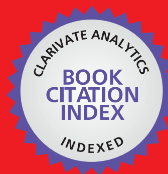
135M
Downloads

154
Countries delivered to

Our authors are among the

TOP 1%
most cited scientists

12.2%
Contributors from top 500 universities



WEB OF SCIENCE™

Selection of our books indexed in the Book Citation Index in Web of Science™ Core Collection (BKCI)

Interested in publishing with us?
Contact book.department@intechopen.com

Numbers displayed above are based on latest data collected.
For more information visit www.intechopen.com

Power-Aware Memory Allocation for Embedded Data-Intensive Signal Processing Applications

Florin Balasa[1], Ilie I. Luican[2], Hongwei Zhu[3] and Doru V. Nasui[4]
[1]*Dept. of Computer Science, Southern Utah University, Cedar City, UT 84721*
[2]*Dept. of Computer Science, University of Illinois at Chicago, Chicago, IL 60607*
[3]*ARM, Inc., San Jose, CA 95134*
[4]*American Int. Radio, Inc., Rolling Meadows, IL 60008*

Abstract
Many signal processing systems, particularly in the multimedia and telecommunication domains, are synthesized to execute data-intensive applications: their cost related aspects – namely power consumption and chip area – are heavily influenced, if not dominated, by the data access and storage aspects. This chapter presents a power-aware memory allocation methodology. Starting from the high-level behavioral specification of a given application, this framework performs the assignment of of the multidimensional signals to the memory layers – the on-chip scratch-pad memory and the off-chip main memory – the goal being the reduction of the dynamic energy consumption in the memory subsystem. Based on the assignment results, the framework subsequently performs the mapping of signals into the memory layers such that the overall amount of data storage be reduced. This software system yields a complete allocation solution: the exact storage amount on each memory layer, the mapping functions that determine the exact locations for any array element (scalar signal) in the specification, and, in addition, an estimation of the dynamic energy consumption in the memory subsystem.

1. Introduction

Many multidimensional signal processing systems, particularly in the areas of multimedia and telecommunications, are synthesized to execute data-intensive applications, the data transfer and storage having a significant impact on both the system performance and the major cost parameters – power and area.
In particular, the memory subsystem is, typically, a major contributor to the overall energy budget of the entire system (8). The *dynamic* energy consumption is caused by memory accesses, whereas the *static* energy consumption is due to leakage currents. Savings of dynamic energy can be potentially obtained by accessing frequently used data from smaller on-chip memories rather than from the large off-chip main memory, the problem being how to optimally assign the data to the memory layers. Note that this problem is basically different from caching for performance (15), (22), where the question is to find how to fill the cache such that the needed data be loaded in advance from the main memory. As on-chip storage, the scratch-pad memories (SPMs) – compiler-controlled static random-access memories, more

energy-efficient than the hardware-managed caches – are widely used in embedded systems, where caches incur a significant penalty in aspects like area cost, energy consumption, hit latency, and real-time guarantees. A detailed study (4) comparing the tradeoffs of caches as compared to SPMs found in their experiments that the latter exhibit 34% smaller area and 40% lower power consumption than a cache of the same capacity. Even more surprisingly, the runtime measured in cycles was 18% better with an SPM using a simple static knapsack-based allocation algorithm. As a general conclusion, the authors of the study found absolutely no advantage in using caches, even in high-end embedded systems in which performance is important. [1] Different from caches, the SPM occupies a distinct part of the virtual address space, with the rest of the address space occupied by the main memory. The consequence is that there is no need to check for the availability of the data in the SPM. Hence, the SPM does not possess a comparator and the miss/hit acknowledging circuitry (4). This contributes to a significant energy (as well as area) reduction. Another consequence is that in cache memory systems, the mapping of data to the cache is done during the code execution, whereas in SPM-based systems this can be done at compilation time, using a suitable algorithm – as this chapter will show.

The energy-efficient assignment of signals to the on- and off-chip memories has been studied since the late nineties. These previous works focused on partitioning the signals from the application code into so-called *copy candidates* (since the on-chip memories were usually caches), and on the optimal selection and assignment of these to different layers into the memory hierarchy (32), (7), (18). For instance, Kandemir and Choudhary analyze and exploit the temporal locality by inserting local copies (21). Their layer assignment builds a separate hierarchy per loop nest and then combines them into a single hierarchy. However, the approach lacks a global view on the lifetimes of array elements in applications having imperfect nested loops. Brockmeyer *et al.* use the steering heuristic of assigning the arrays having the lowest access number over size ratio to the lowest memory layer first, followed by incremental reassignments (7). Hu *et al.* can use *parts* of arrays as copies, but they typically use cuts along the array dimensions (18) (like rows and columns of matrices). Udayakumaran and Barua propose a dynamic allocation model for SPM-based embedded systems (29), but the focus is global and stack data, rather than multidimensional signals. Issenin *et al.* perform a data reuse analysis in a multi-layer memory organization (19), but the mapping of the signals into the hierarchical data storage is not considered. The energy-aware partitioning of an on-chip memory in multiple banks has been studied by several research groups, as well. Techniques of an exploratory nature analyze possible partitions, matching them against the access patterns of the application (25), (11). Other approaches exploit the properties of the dynamic energy cost and the resulting structure of the partitioning space to come up with algorithms able to derive the optimal partition for a given access pattern (6), (1).

Despite many advances in memory design techniques over the past two decades, existing computer-aided design methodologies are still ineffective in many aspects. In several previous works, the reduction of the dynamic energy consumption in hierarchical memory subsystems is addressed using in part enumerative approaches, simulations, profiling, heuristic explorations of the solution space, rather than a formal methodology. Also, several models of mapping the multidimensional signals into the physical memory were proposed in the past (see (12) for a good overview).

[1] Caches have been a big success for desktops though, where the usual approach to adding SRAM is to configure it as a cache.

However, they all failed
(a) to provide efficient implementations,
(b) to prove their effectiveness in hierarchical memory organizations, and
(c) to provide quantitative measures of quality for the mapping solutions.
Moreover, the reduction of power consumption and the mapping of signals in hierarchical memory subsystems were treated in the past as completely separate problems.
This chapter presents a power-aware memory allocation methodology. Starting from the high-level behavioral specification of a given application, where the code is organized in sequences of loop nests and the main data structures are multidimensional arrays, this framework performs the assignment of of the multidimensional signals to the memory layers – the on-chip scratch-pad memory and the off-chip main memory – the goal being the reduction of the dynamic energy consumption in the memory subsystem. Based on the assignment results, the framework subsequently performs the mapping of signals into the memory layers such that the overall amount of data storage be reduced. This software system yields a complete allocation solution: the exact storage amount on each memory layer, the mapping functions that determine the exact locations for any array element (scalar signal) in the specification, metrics of quality for the allocation solution, and also an estimation of the dynamic energy consumption in the memory subsystem using the CACTI power model (31). Extensions of the current framework to support dynamic allocation are currently under development.
The rest of the chapter is organized as follows. Section 2 presents the algorithm that assigns the signals to the memory layers, aiming to minimize the dynamic energy consumption in the hierarchical memory subsystem subject to SPM size constraints. Section 3 describes the global flow of the memory allocation approach, focusing on the mapping aspects. Section 4 discusses on implementation and presents experimental results. Finally, Section 5 summarizes the main conclusions of this research.

2. Power-aware signal assignment to the memory layers

The algorithms describing the functionality of real-time multimedia and telecommunication applications are typically specified in a high-level programming language, where the code is organized in sequences of loop nests having as boundaries linear functions of the outer loop iterators. Conditional instructions are very common as well, and the multidimensional array references have linear indexes (the variables being the loop iterators).[2]
Figure 1 shows an illustrative example whose structure is similar to the kernel of a motion detection algorithm (9) (the actual code containing also a *delay* operator – not relevant in this context). The problem is to automatically identify those parts of arrays from the given application code that are more intensely accessed, in order to steer their assignment to the energy-efficient data storage layer (the on-chip scratch-pad memory) such that the dynamic energy consumption in the hierarchical memory subsystem be reduced.
The number of storage accesses for each array element can certainly be computed by the simulated execution of the code. For instance, the number of accesses was counted for every pair of possible indexes (between 0 and 80) of signal A (see Fig. 1). The array elements near the

[2] Typical piece-wise linear operations can be transformed into affine specifications (17). In addition, pointer accesses can be converted at compilation to array accesses with explicit index functions (16). Moreover, specifications where not all loop structures are *for* loops and not all array indexes are affine functions of the loop iterators can be transformed into affine specifications that captures all the memory references amenable to optimization (20). Extensions to support a larger class of specifications are thus possible, but they are orthogonal to the work presented in this chapter.

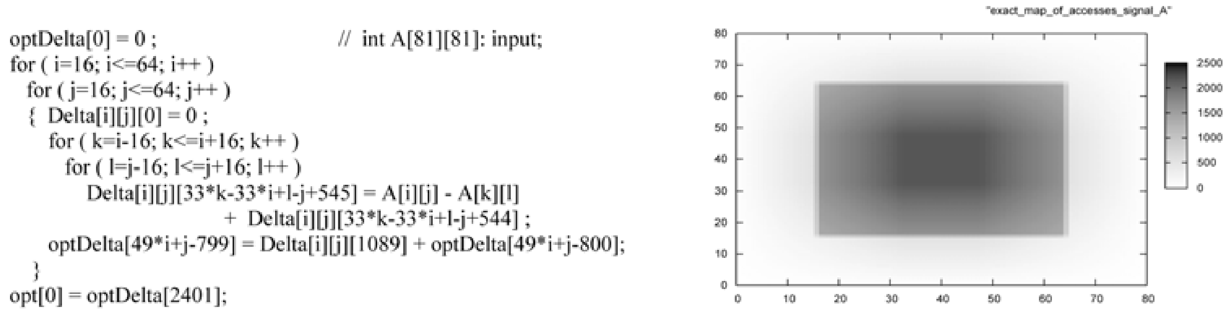


Fig. 1. Code derived from a motion detection (9) kernel ($m = n = 16,\ M = N = 64$) and the exact map of memory *read* accesses (obtained by simulation) for the 2-D signal A.

center of the index space are accessed with high intensity (for instance, $A[40][40]$ is accessed 2,178 times; $A[16][40]$ is accessed 1,650 times), whereas the array elements at the periphery are accessed with a significantly lower intensity (for instance, $A[0][40]$ is accessed 33 times and $A[0][0]$ only once).

The drawbacks of such an approach are twofold. First, the simulated execution may be computationally ineffective when the number of array elements is very significant, or when the application code contains deep loop nests. Second, even if the simulated execution were feasible, such a scalar-oriented technique would not be helpful since the addressing hardware of the data memories would result very complex. An address generation unit (AGU) is typically implemented to compute arithmetic expressions in order to generate sequences of addresses (26); a set of array elements is not a good input for the design of an efficient AGU.

Our proposed computation methodology for power-aware signal assignment to the memory layers is described below, after defining a few basic concepts.

Each *array reference* $M[x_1(i_1,\ldots,i_n)]\ \cdots\ [x_m(i_1,\ldots,i_n)]$ of an m-dimensional signal M, in the scope of a nest of n loops having the iterators $i_1,\ldots,\ i_n$, is characterized by an *iterator space* and an *index* (or *array*) *space*. The iterator space signifies the set of all iterator vectors $\mathbf{i} = (i_1,\ldots,i_n) \in \mathbf{Z}^n$ in the scope of the array reference, and it can be typically represented by a so-called $\mathbf{Z}$-polytope (a polyhedron bounded and closed, restricted to the set $\mathbf{Z}^n$): $\{\ \mathbf{i} \in \mathbf{Z}^n \mid \mathbf{A}\cdot\mathbf{i} \geq \mathbf{b}\ \}$. The index space is the set of all index vectors $\mathbf{x} = (x_1,\ \ldots\ ,x_m) \in \mathbf{Z}^m$ of the array reference. When the indexes of an array reference are linear mappings with integer coefficients of the loop iterators, the index space consists of one or several *linearly bounded lattices* (27): $\{\ \mathbf{x}\ =\ \mathbf{T}\cdot\mathbf{i}+\mathbf{u}\ \in \mathbf{Z}^m \mid\ \mathbf{A}\cdot\mathbf{i} \geq \mathbf{b}\ ,\ \mathbf{i} \in \mathbf{Z}^n\}$. For instance, the array reference $A[i+2*j+3][j+2*k]$ from the loop nest

```
          for (i=0; i<=2; i++)
             for (j=0; j<=3; j++)
                for (k=0; k<=4; k++)
                     if ( 6*i+4*j+3*k ≤ 12 )  ···
A[i+2*j+3][j+2*k] ···
```

has the iterator space $P = \left\{ \left[\begin{array}{c} i \\ j \\ k \end{array}\right] \in \mathbf{Z}^3 \;\middle|\; \left[\begin{array}{ccc} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ -6 & -4 & -3 \end{array}\right] \left[\begin{array}{c} i \\ j \\ k \end{array}\right] \geq \left[\begin{array}{c} 0 \\ 0 \\ 0 \\ -12 \end{array}\right] \right\}$. (The inequalities $i \leq 2$, $j \leq 3$, and $k \leq 4$ are redundant.)

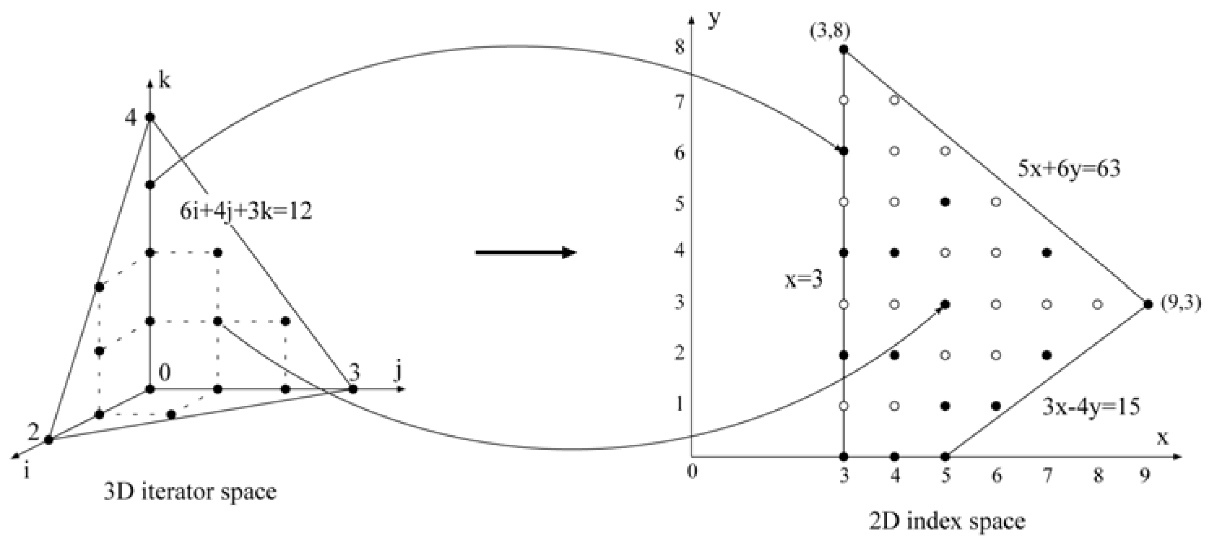


Fig. 2. The mapping of the iterator space into the index space of the array reference $A[i+2*j+3][j+2*k]$.

The A-elements of the array reference have the indices x, y:
$\left\{ \begin{bmatrix} x \\ y \end{bmatrix} = \mathbf{T}\cdot\mathbf{i}+\mathbf{u} = \begin{bmatrix} 1 & 2 & 0 \\ 0 & 1 & 2 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} + \begin{bmatrix} 3 \\ 0 \end{bmatrix} \middle| \begin{bmatrix} i \\ j \\ k \end{bmatrix} \in P \right\}$. The points of the index space lie inside the $\mathbf{Z}$-polytope $\{\, x \geq 3\,,\ y \geq 0\,,\ 3x-4y \leq 15\,,\ 5x+6y \leq 63\,,\ \ x,y \in \mathbf{Z}\}$, whose boundary is the image of the boundary of the iterator space P (see Fig. 2). However, it can be shown that only those points (x,y) satisfying also the inequalities $-6x+8y \geq 19k-30$, $x-2y \geq -4k+3$, and $y \geq 2k \geq 0$, for some positive integer k, belong to the index space; these are the black points in the right quadrilateral from Fig. 2. In this example, each point in the iterator space is mapped to a distinct point of the index space; this is not always the case, though.

Algorithm 1: Power-aware signal assignment to the SPM and off-chip memory layers

Step 1 *Extract the array references from the given algorithmic specification and decompose the array references for every indexed signal into* disjoint *lattices.*

```
for (k=0; k<=10; k++)
  for (l=0; l<=5; l++)        M[k][l]= ... ;
for (j=0; j<=5; j++)
  for (i=0; i<=2j; i++)   ... = M[i][j];
for (i=1; i<=5; i++)
  for (j=0; j<=i-1; j++)  ... = M[2*i][j+1];
```

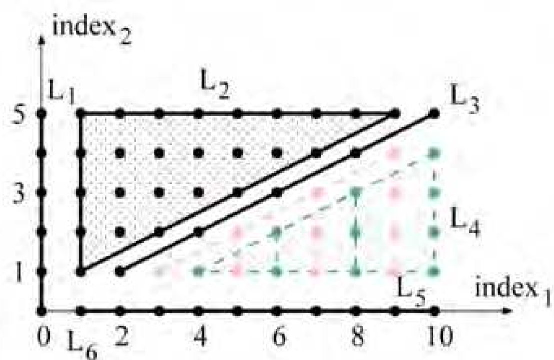


Fig. 3. The decomposition of the array space of signal M in 6 disjoint lattices.

The motivation of the decomposition of the array references relies on the following intuitive idea: the disjoint lattices belonging to many array references are actually those parts of arrays more heavily accessed during the code execution. This decomposition can be analytically performed, using intersections and differences of lattices – operations quite complex (3) involving computations of Hermite Normal Forms and solving Diophantine linear systems (24), computing the vertices of **Z**-polytopes (2) and their supporting polyhedral cones, counting the integral points in **Z**-polyhedra (5; 10), and computing integer projections of polytopes (30). Figure 3 shows the result of such a decomposition for the three array references of signal M. The resulting lattices have the following expressions (in non-matrix format):
$L_1 = \{x = 0,\ y = t \mid 5 \geq t \geq 0\}$
$L_2 = \{x = t_1,\ y = t_2 \mid 5 \geq t_2 \geq 1\ ,\ 2t_2 - 1 \geq t_1 \geq 1\}$
$L_3 = \{x = 2t,\ y = t \mid 5 \geq t \geq 1\}$
$L_4 = \{x = 2t_1 + 2,\ y = t_2 \mid 4 \geq t_1 \geq t_2 \geq 1\}$
$L_5 = \{x = 2t_1 + 1,\ y = t_2 \mid 4 \geq t_1 \geq t_2 \geq 1\}$
$L_6 = \{x = t,\ y = 0 \mid 10 \geq t \geq 1\}$
Step 2 *Compute the number of memory accesses for each disjoint lattice.*
The total number of memory accesses to a given linearly bounded lattice of a signal is computed as follows:
Step 2.1 Select an array reference of the signal and intersect the given lattice with it. If the intersection is not empty, then the intersection is a linearly bounded lattice as well (27).
Step 2.2 Compute the number of points in the (non-empty) intersection: this is the number of memory accesses to the given lattice (as part of the selected array reference).
Step 2.3 Repeat steps 2.1 and 2.2 for all the signal's array references in the code, cumulating the numbers of accesses.
For example, let us consider one of signal A's lattices[3] $\{\ 64 \geq x\ ,\ y \geq 16\}$ obtained in *Step 1*. Intersecting it with the array reference $A[k][l]$ (see the code in Fig. 1), we obtain the lattice $\{i = t_1,\ j = t_2,\ k = t_3,\ l = t_4 \mid 64 \geq t_1,\ t_2,\ t_3,\ t_4 \geq 16,\ t_1 + 16 \geq t_3 \geq t_1 - 16,\ t_2 + 16 \geq t_4 \geq t_2 - 16\}$. The size of this set is 2,614,689 , which is the number of memory accesses to the given lattice as part of the array reference $A[k][l]$. Since the given lattice is also included in the other array reference[4] in the code – $A[i][j]$, a similar computation yields 1,809,025 accesses to the same lattice as part of $A[i][j]$. Hence, the total amount of memory accesses to the given lattice is 2,614,689+1,809,025=4,423,714.
Figure 4 displays a computed map of memory accesses for the signal A, where A's index space is in the horizontal plane xOy and the numbers of memory accesses are on the vertical axis Oz. This computed map is an approximation of the exact map in Fig. 1 since the accesses within each lattice are considered uniform, equal to the average values obtained above. The advantage of this map construction is that the (usually time-expensive) simulation is not needed any more, being replaced by algebraic computations. Note that a finer granularity in the decomposition of the index space of a signal in disjoint lattices entails a computed map of accesses closer to the exact map.
Step 3 *Select the lattices having the highest access numbers, whose total size does not exceed the maximum SPM size (assumed to be a design constraint), and assign them to the SPM layer. The other lattices will be assigned to the main memory.*

[3] When the lattice has **T**=**I** – the identity matrix – and **u**=**0**, the lattice is actually a **Z**-polytope, like in this example.
[4] Note that in our case, due to *Step 1*, any disjoint lattice is either included in the array reference or disjoint from it.

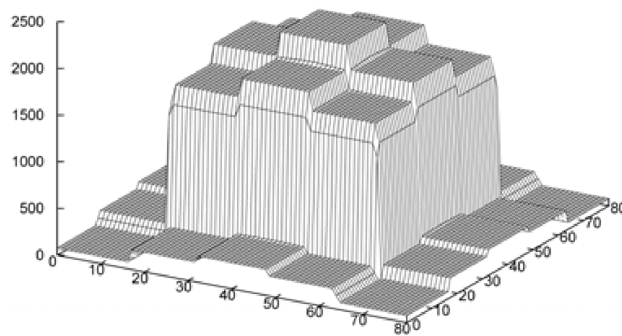


Fig. 4. Computed 3D map of memory *read* accesses for the signal A from the illustrative code in Figure 1.

Storing on-chip all the signals is, obviously, the most desirable scenario in point of view of dynamic energy consumption, which is typically impossible. We assume here that the SPM size is constrained to smaller values than the overall storage requirement. In our tests, we computed the ratio between the dynamic energy reduction and the SPM size after mapping; the value of the SPM size maximizing this ratio was selected, the idea being to obtain the maximum benefit (in energy point of view) for the smallest SPM size.

3. Mapping signals within memory layers

This design phase has the following goals: (a) to map the signals (already assigned to the memory layers) into amounts of data storage as small as possible, both for the SPM and the main memory; (b) to compute these amounts of storage after mapping on both memory layers (allocation solution) and be able to determine the memory location of each array element from the specification (assignment solution); (c) to use mapping functions simple enough in order to ensure an address generation hardware of a reasonable complexity; (d) to ascertain that any scalar signals (array elements) *simultaneously alive* are mapped to distinct storage locations.
Since the mapping models (13) and (28) play an important part in this section, they will be explained and illustrated below.
To reduce the size of a multidimensional array mapped to memory, the model (13) considers all the possible *canonical*[5] linearizations of the array; for any linearization, the largest distance at any time between two live elements is computed. This distance plus 1 is then the storage "window" required for the mapping of the array into the data memory. More formally, $|W_A| \;=\; \min \max \{ \, dist(A_i, A_j) \, \} \;+\; 1$, where $|W_A|$ is the size of the storage window of a signal A, the minimum is taken over all the canonical linearizations, while the maximum is taken over all the pairs of A-elements (A_i,A_j) simultaneously alive.
This mapping model will be illustrated for the loop nest from Fig. 5(a). The graph above the code represents the array (index) space of signal A. The points represent the A-elements $A[index_1][index_2]$ which are produced (and consumed as well) in the loop nest. The points to the left of the dashed line represent the elements produced till the end of the iteration ($i = 14,\ j = 4$), the black points being the elements still alive (i.e., produced and still used as

[5] For instance, a 2-D array can be typically linearized concatenating the rows or concatenating the columns. In addition, the elements in a given dimension can be mapped in the increasing or decreasing order of the respective index.

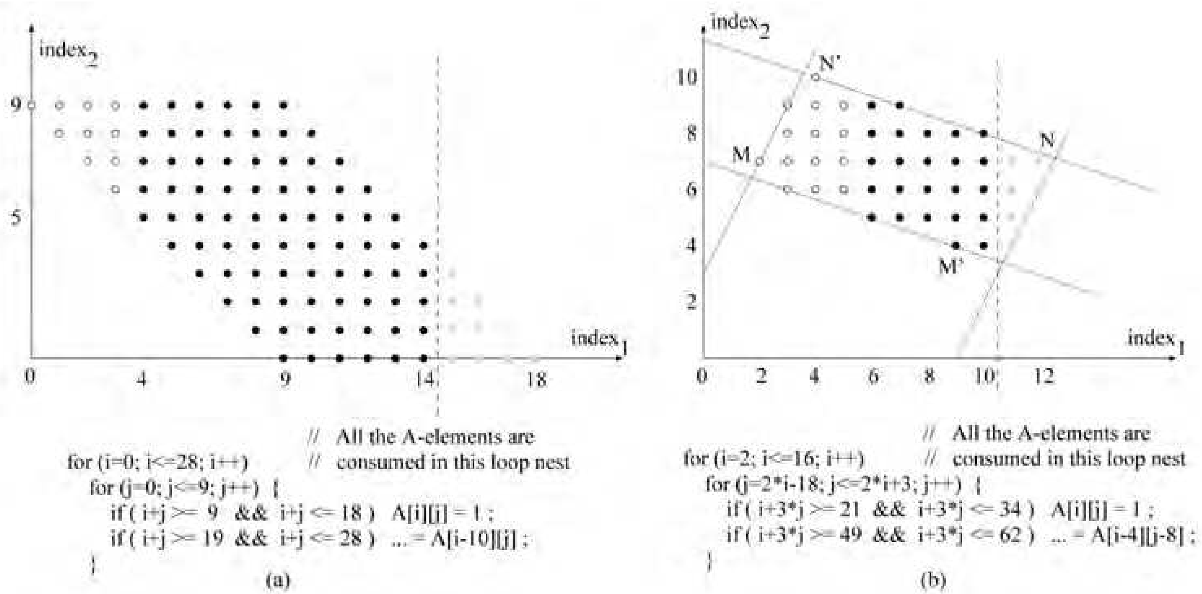


Fig. 5. (a-b) Illustrative examples having a similar code structure. The mapping model by array linearization yields a better allocation solution for the former example, whereas the bounding window model behaves better for the latter one.

operands in the next iterations), while the circles representing A-elements already 'dead' (i.e., not needed as operands any more). The light grey points to the right of the dashed line are A-elements still unborn (to be produced in the next iterations).

If we consider the array linearization by column concatenation in the increasing order of the columns (($A[index_1][index_2]$, $index_1$=0,18), $index_2$=0,9), two elements simultaneously alive, placed the farthest apart from each other, are $A[9][0]$ and $A[9][9]$. The distance between them is 9×19=171. Now, if we consider the array linearization by row concatenation in the increasing order of the rows (($A[index_1][index_2]$, $index_2$=0,9), $index_1$=0,18), the maximum distance between live elements is 99 (e.g., between $A[4][5]$ and $A[14][4]$). For all the canonical linearizations, the maximum distances have the values {99, 109, 171, 181}. The best canonical linearization for the array A is the concatenation row by row, increasingly. A memory window W_A of 99+1=100 successive locations (relative to a certain base address) is sufficient to store the array without mapping conflicts: it is sufficient that any access to $A[index_1][index_2]$ be redirected to $W_A[(10 * index_1 + index_2) \bmod 100]$.

In order to avoid the inconvenience of analyzing different linearization schemes, another possibility is to compute a maximal bounding window $W_A = (w_1, \ldots, w_m)$ large enough to encompass at any time the simultaneously alive A-elements. An access to the element $A[index_1] \ldots [index_m]$ can then be redirected without any conflict to the window location $W_A[index_1 \bmod w_1] \ldots [index_m \bmod w_m]$; in its turn, the window is mapped, relative to a base address, into the physical memory by a typical canonical linearization, like row or column concatenation for 2-D arrays. Each window element w_k is computed as the maximum difference in absolute value between the k-th indexes of any two A-elements (A_i,A_j) simultaneously alive, plus 1. More formally, $w_k = \max\{ |x_k(A_i) - x_k(A_j)| \} + 1$, for $k = 1, \ldots, m$. This ensures that any two array elements simultaneously alive are mapped to distinct memory locations. The amount of data memory required for storing (after mapping) the array A is the volume of the bounding window W_A, that is, $|W_A| = \Pi_{k=1}^{m} w_k$.

In the illustrative example shown in Fig. 5(a), the bounding window of the signal A is $W_A = (11\ ,\ 10)$. It follows that the storage allocation for signal A is 100 locations if the linearization model is used, and $w_1 \times w_2$=110 locations when the bounding window model is applied. However, in the example shown in Fig. 5(b), where the code has a similar structure, the bounding window model yields a better allocation result – 30 storage locations, since the 2-D window of A is $W_A = (5\ ,\ 6)$, whereas the linearization model yields 32 locations (the best canonical linearization being the row concatenation in the increasing order of rows).

Our software system incorporates both mapping models, their implementation being based on the same polyhedral framework operating with lattices, used also in Section 2. This is advantageous both from the point of view of computational efficiency and relative to the amount of allocated data storage – since the mapping window for each signal is the smallest one of the two models. Moreover, this methodology can be applied independently to the memory layers, providing a complete storage allocation/assignment solution for distributed memory organizations.

Before explaining the global flow of the algorithm, let us examine the simple case of a code with only one array reference in it: take, for instance, the two nested loops from Fig. 5(b), but without the second conditional statement that consumes the A-elements. In the bounding window model, W_A can be determined by computing the integer projections on the two axes of the lattice of $A[i][j]$, represented graphically by all the points inside the quadrilateral from Fig. 5(b). It can be directly observed that the integer projections of this polygon have the sizes: $w_1 = 11$ and $w_2 = 7$. In the linearization model, denoting x and y the two indexes, the distance between two A-elements $A_1(x_1, y_1)$ and $A_2(x_2, y_2)$, assuming row concatenation in the increasing order of the rows, is: $dist(A_1, A_2) = (x_2 - x_1)\Delta y + (y_2 - y_1)$, where Δy is the range of the second index (here, equal to 7) in the array space.[6] Then, the A-elements at a maximum distance have the minimum and, respectively, the maximum index vectors relative to the lexicographic order. These array A-elements are represented by the points $M = A[2][7]$ and $N = A[12][7]$ in Fig. 5(b), and $dist(M, N)$ = (12-2)×7 +(0-0)=70. Similarly, in the linearization by column concatenation, the array elements at the maximum distance from each other are still the elements with (lexicographically) minimum and maximum index vectors, *provided an interchange of the indexes is applied first*. These are the points $M' = A[9][4]$ and $N' = A[4][10]$ in Fig. 5(b). More general, the maximum distance between the points of a live lattice in a canonical linearization is the distance between the (lexicographically) minimum and maximum index vectors, providing an index permutation is applied first. The distance between the array elements $A_i(x_1^i, x_2^i, \ldots, x_m^i)$ and $A_j(x_1^j, x_2^j, \ldots, x_m^j)$ is:

$dist(A_i, A_j) = (x_1^j - x_1^i)\Delta x_2 \cdots \Delta x_m + (x_2^j - x_2^i)\Delta x_3 \cdots \Delta x_m + \cdots + (x_{m-1}^j - x_{m-1}^i)\Delta x_m + (x_m^j - x_m^i)$ where the index vector of A_j is lexicographically larger than of A_i (Δx_i is the range of x_i).

Algorithm 2: For each memory layer (SPM and main memory) *compute the mapping windows for every indexed signal having lattices assigned to that layer.*

Step 1 *Compute underestimations of the window sizes on the current memory layer for each indexed signal, taking into account only the live signals at the boundaries between the loop nests.*

Let A be an m-dimensional signal in the algorithmic specification, and let $\mathcal{P}_A$ be the set of disjoint lattices partitioning the index space of A. A high-level pseudo-code of the computation

[6] To ensure that the distance is a nonnegative number, we shall assume that $[x_2\ y_2]^T \succ [x_1\ y_1]^T$ relative to the lexicographic order. The vector $\mathbf{y} = [y_1, \ldots, y_m]^T$ is larger lexicographically than $\mathbf{x} = [x_1, \ldots, x_m]^T$ (written $\mathbf{y} \succ \mathbf{x}$) if ($y_1 > x_1$), or ($y_1 = x_1$ and $y_2 > x_2$), ... , or ($y_1 = x_1, \ldots, y_{m-1} = x_{m-1}$, and $y_m > x_m$).

of A's preliminary windows is given below. Preliminary window sizes for each canonical linearization according to DeGreef's model (13) are computed first, followed by the computation of the window size underestimate according to Tronçon's model (28) in the same framework operating with lattices. The meaning of the variables are explained as comments.

for (*each canonical linearization* $\mathcal{C}$) {
for (*each disjoint lattice* $L \in \mathcal{P}_A$) // compute the (lexicographically) minimum and maximum ...
compute $x^{min}(L)$ and $x^{max}(L)$; // ... index vectors of L relative to $\mathcal{C}$
for (*each boundary n between the loop nests n and* $n+1$) { // the start of the code is boundary 0
let $\mathcal{P}_A(n)$ be the collection of disjoint lattices of A, which are alive at the boundary n ;
// these are disjoint lattices produced *before* the boundary and consumed *after* it
let $X_n^{min} = \min_{L \in \mathcal{P}_A(n)} \{x^{min}(L)\}$ and $X_n^{max} = \max_{L \in \mathcal{P}_A(n)} \{x^{max}(L)\}$;
$|W_{\mathcal{C}}(n)| = dist(X_n^{min}, X_n^{max}) + 1$; // The distance is computed in the canonical linearization $\mathcal{C}$
}
$|W_{\mathcal{C}}| = \max_n \{ |W_{\mathcal{C}}(n)| \}$; // the window size according to (13) for the canonical linearization $\mathcal{C}$
} // (possibly, an underestimate)
for (*each disjoint lattice* $L \in \mathcal{P}_A$)
for (*each dimension k of signal A*)
compute $x_k^{min}(L)$ and $x_k^{max}(L)$; // the extremes of the integer projection of L on the k-th axis
for (*each boundary n between the loop nests n and* $n+1$) { // the start of the code is boundary 0
let $\mathcal{P}_A(n)$ be the collection of disjoint lattices of A, which are alive at the boundary n ;
for (*each dimension k of signal A*) {
let $X_k^{min} = \min_{L \in \mathcal{P}_A(n)} \{x_k^{min}(L)\}$ and $X_k^{max} = \max_{L \in \mathcal{P}_A(n)} \{x_k^{max}(L)\}$;
$w_k(n) = X_k^{max} - X_k^{min} + 1$; // The k-th side of A's bounding window at boundary n
}
}
for (*each dimension k of signal A*) $w_k = max_n\{w_k(n)\}$; // k-th side of A's window over all boundaries
$|W| = \Pi_{k=1}^{m} w_k$; // the window size according to (28) (possibly, an underestimate)

Step 1 finds the exact values of the window sizes for both mapping models only when every loop nest either produces or consumes (but not both!) the signal's elements. Otherwise, when in a certain loop nest elements of the signal are both produced *and* consumed (see the illustrative example from Fig. 5(a)), then the window sizes obtained at the end of *Step 1* may be only underestimates since an increase of the storage requirement can happen *inside* the loop nest. Then, an additional step is required to find the exact values of the window sizes in both mapping models.

Step 2 *Update the mapping windows for each indexed signal in every loop nest producing and consuming elements of the signal.*

The guiding idea is that local or global maxima of the bounding window size $|W|$ are reached immediately before the consumption of an A-element, which may entail a shrinkage of some side of the bounding window encompassing the live elements. Similarly, the local or global maxima of $|W_C|$ are reached immediately before the consumption of an A-element, which may entail a decrease of the maximum distance between live elements. Consequently, for each A-element consumed in a loop nest which also produces A-elements, we construct the disjoint lattices partially produced and those partially consumed until the iteration when the A-element is consumed. Afterwards, we do a similar computation as in *Step 1* which may result in increased values for $|W_C|$ and/or $|W|$.
Finally, the amount of data memory allocated for signal A on the current memory layer is $|W_A| = \min\{\,|W|\,, \min_C\{\,|W_C|\,\}\,\}$, that is, the smallest data storage provided by the bounding window and the linearization mapping models. In principle, the overall amount of data memory after mapping is $\sum_A |W_A|$ – the sum of the mapping window sizes of all the signals having lattices assigned to the current memory layer. In addition, a post-processing step attempts to further enhance the allocation solution: our polyhedral framework allows to efficiently check weather two multidimensional signals have disjoint lifetimes, in which case the signals can share the largest of the two windows. More general, an incompatibility graph (14) is used to optimize the memory sharing among all the signals at the level of whole code.

4. Experimental results

A hierarchical memory allocation tool has been implemented in C++, incorporating the algorithms described in this chapter. For the time being, the tool supports only a two-level memory hierarchy, where an SPM is used between the main memory and the processor core. The dynamic energy is computed based on the number of accesses to each memory layer. In computing the dynamic energy consumptions for the SPM and the main (off-chip) memory, the CACTI v5.3 power model (31) was used.
Table 1 summarizes the results of our experiments, carried out on a PC with an Intel Core 2 Duo 1.8 GHz processor and 512 MB RAM. The benchmarks used are: (1) a motion detection algorithm used in the transmission of real-time video signals on data networks; (2) the kernel of a motion estimation algorithm for moving objects (MPEG-4); (3) Durbin's algorithm for solving Toeplitz systems with N unknowns; (4) a singular value decomposition (SVD) updating algorithm (23) used in spatial division multiplex access (SDMA) modulation in mobile communication receivers, in beamforming, and Kalman filtering; (5) the kernel of a voice coding application – essential component of a mobile radio terminal.
The table displays the total number of memory accesses, the data memory size (in storage locations/bytes), and the dynamic energy consumption assuming only one (off-chip) memory layer; in addition, the SPM size and the savings of dynamic energy applying, respectively, a previous model steered by the total number of accesses for whole arrays (7), another previous model steered by the most accessed array rows/columns (18), and the current model, versus the single-layer memory scenario; the CPU times. The energy consumptions for the motion estimation benchmark were, respectively, 1894, 1832, and 1522 μJ; the saved energies relative to the energy in column 4 are displayed as percentages in columns 6-8. Our experiments show that the savings of dynamic energy consumption are from 40% to over 70% relative to the energy used in the case of a flat memory design. Although previous models produce energy savings as well, our model led to 20%-33% better savings than them.
Different from the previous works on power-aware assignment to the memory layers, our framework provides also the mapping functions that determine the exact locations for any

Application parameters	#Memory accesses	Mem. size	Dyn. energy 1-layer [μJ]	SPM size	Dyn. energy saved (7)	Dyn. energy saved (18)	Dyn. energy saved	CPU [sec]
Motion detection M=N=32, m=n=4	136,242	2,740	486	841	30.2%	44.5%	49.2%	4
Motion estimation M=32, N=16	864,900	3,624	3,088	1,416	38.7%	40.7%	50.7%	23
Durbin algorithm N =500	1,004,993	1,249	3,588	764	55.2%	58.5%	73.2%	28
SVD updating n =100	6,227,124	34,950	22,231	12,672	35.9%	38.4%	46.0%	37
Vocoder	200,000	12,690	714	3,879	30.8%	32.5%	39.5%	8

Table 1. Experimental results.

www.intechopen.com

array element in the specification. This provides the necessary information for the automated design of the address generation unit, which is one of our future development directions. Different from the previous works on signal-to-memory mapping, our framework offers a hierarchical strategy and, also, two metrics of quality for the memory allocation solutions: (a) the sum of the *minimum* array windows (that is, the optimum memory sharing between elements of same arrays), and (b) the minimum storage requirement for the execution of the application code (that is, the optimum memory sharing between all the scalar signals or array elements in the code) (3).

5. Conclusions

This chapter has presented an integrated computer-aided design methodology for power-aware memory allocation, targeting embedded data-intensive signal processing applications. The memory management tasks – the signal assignment to the memory layers and their mapping to the physical memories – are efficiently addressed within a common polyhedral framework.

6. References

[1] F. Angiolini, L. Benini, and A. Caprara, “An efficient profile-based algorithm for scratch-pad memory partitioning,” *IEEE Trans. Computer-Aided Design*, vol. 24, no. 11, pp. 1660-1676, Nov. 2005.

[2] D. Avis, “lrs: A revised implementation of the reverse search vertex enumeration algorithm,” in *Polytopes – Combinatorics and Computation*, G. Kalai and G. Ziegler (eds.), Birkhauser-Verlag, 2000, pp. 177-198.

[3] F. Balasa, H. Zhu, and I.I. Luican, “Computation of storage requirements for multi-dimensional signal processing applications,” *IEEE Trans. VLSI Systems*, vol. 15, no. 4, pp. 447-460, April 2007.

[4] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel, “Scratchpad memory: A design alternative for cache on-chip memory in embedded systems,” in *Proc. 10th Int. Workshop on Hardware/Software Codesign*, Estes Park CO, May 2002.

[5] A.I. Barvinok, “A polynomial time algorithm for counting integral points in polyhedra when the dimension is fixed,” *Mathematics of Operations Research*, vol. 19, no. 4, pp. 769-779, Nov. 1994.

[6] L. Benini, L. Macchiarulo, A. Macii, E. Macii, and M. Poncino, “Layout-driven memory synthesis for embedded systems-on-chip,” *IEEE Trans. VLSI Systems*, vol. 10, no. 2, pp. 96-105, April 2002.

[7] E. Brockmeyer, M. Miranda, H. Corporaal, and F. Catthoor, “Layer assignment techniques for low energy in multi-layered memory organisations,” in *Proc. ACM/IEEE Design, Automation & Test in Europe*, Munich, Germany, Mar. 2003, pp. 1070-1075.

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

[9] E. Chan and S. Panchanathan, “Motion estimation architecture for video compression,” *IEEE Trans. on Consumer Electronics*, vol. 39, pp. 292-297, Aug. 1993.

[10] Ph. Clauss and V. Loechner, “Parametric analysis of polyhedral iteration spaces,” *J. VLSI Signal Processing*, vol. 19, no. 2, pp. 179-194, 1998.

[11] S. Coumeri and D.E. Thomas, "Memory modeling for system synthesis," *IEEE Trans. VLSI Systems*, vol. 8, no. 3, pp. 327-334, June 2000.
[12] A. Darte, R. Schreiber, and G. Villard, "Lattice-based memory allocation," *IEEE Trans. Computers*, vol. 54, pp. 1242-1257, Oct. 2005.
[13] E. De Greef, F. Catthoor, and H. De Man, "Memory size reduction through storage order optimization for embedded parallel multimedia applications," *Parallel Computing*, special issue on "Parallel Processing and Multimedia," Elsevier, vol. 23, no. 12, pp. 1811-1837, Dec. 1997.
[14] G. De Micheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, 1994.
[15] J.Z. Fang and M. Lu, "An iteration partition approach for cache or local memory thrashing on parallel processing," *IEEE Trans. Computers*, vol. 42, no. 5, pp. 529-546, 1993.
[16] B. Franke and M. O'Boyle, "Compiler transformation of pointers to explicit array accesses in DSP applications," in *Proc. Int. Conf. Compiler Construction*, 2001.
[17] C. Ghez, M. Miranda, A. Vandecapelle, F. Catthoor, D. Verkest, "Systematic high-level address code transformations for piece-wise linear indexing: illustration on a medical imaging algorithm," in *Proc. IEEE Workshop on Signal Processing Systems*, pp. 623-632, Lafayette LA, Oct. 2000.
[18] Q. Hu, A. Vandecapelle, M. Palkovic, P.G. Kjeldsberg, E. Brockmeyer, and F. Catthoor, "Hierarchical memory size estimation for loop fusion and loop shifting in data-dominated applications," in *Proc. Asia-S. Pacific Design Automation Conf.*, Yokohama, Japan, Jan. 2006, pp. 606-611.
[19] I. Issenin, E. Brockmeyer, M. Miranda, and N. Dutt, "Data reuse analysis technique for software-controlled memory hierarchies," in *Proc. Design, Automation & Test in Europe*, 2004.
[20] I. Issenin and N. Dutt, "FORAY-GEN: Automatic generation of affine functions for memory optimization," *Proc. Design, Automation & Test in Europe*, 2005.
[21] M. Kandemir and A. Choudhary, "Compiler-directed scratch-pad memory hierarchy design and management," in *Proc. 39th ACM/IEEE Design Automation Conf.*, Las Vegas NV, June 2002, pp. 690-695.
[22] N. Manjiakian and T. Abdelrahman, "Reduction of cache conflicts in loop nests," *Tech. Report CSRI-318*, Univ. Toronto, Canada, 1995.
[23] M. Moonen, P. V. Dooren, and J. Vandewalle, "An SVD updating algorithm for subspace tracking," *SIAM J. Matrix Anal. Appl.*, vol. 13, no. 4, pp. 1015-1038, 1992.
[24] A. Schrijver, *Theory of Linear and Integer Programming*, New York: John Wiley, 1986.
[25] W. Shiue and C. Chakrabarti, "Memory exploration for low-power embedded systems," in *Proc. 35th ACM/IEEE Design Automation Conf.*, June 1998, pp. 140-145.
[26] G. Talavera, M. Jayapala, J. Carrabina, and F. Catthoor, "Address generation optimization for embedded high-performance processors: A survey," *J. Signal Processing Systems*, Springer, vol. 53, no. 3, pp. 271-284, Dec. 2008.
[27] L. Thiele, "Compiler techniques for massive parallel architectures," in *State-of-the-art in Computer Science*, P. Dewilde (ed.), Kluwer Acad. Publ., 1992.
[28] R. Tronçon, M. Bruynooghe, G. Janssens, and F. Catthoor, "Storage size reduction by in-place mapping of arrays," in *Verification, Model Checking and Abstract Interpretation*, A. Coresi (ed.), 2002, pp. 167-181.
[29] S. Udayakumaran and R. Barua, "Compiler-decided dynamic memory allocation for scratch-pad based embedded systems," in *Proc. Int. Conf. on Compilers, Architecture, and Synthesis for Embedded Systems*, pp. 276-286, New York NY, Oct. 2003.

[30] S. Verdoolaege, K. Beyls, M. Bruynooghe, and F. Catthoor, "Experiences with enumeration of integer projections of parametric polytopes," in *Compiler Construction: 14th Int. Conf.*, R. Bodik (ed.), vol. 3443, pp. 91-105, Springer, 2005.
[31] S. Wilton and N. Jouppi, "CACTI: An enhanced access and cycle time model," *IEEE J. Solid-State Circ.*, vol. 31, pp. 677-688, May 1996.
[32] S. Wuytack, J.-P. Diguet, F. Catthoor, and H. De Man, "Formalized methodology for data reuse exploration for low-power hierarchical memory mappings," *IEEE Trans. VLSI Systems*, vol. 6, no. 4, pp. 529-537, Dec. 1998.

IntechOpen

IntechOpen

IntechOpen

IntechOpen

Data Storage

Edited by Florin Balasa

ISBN 978-953-307-063-6
Hard cover, 226 pages
Publisher InTech
Published online 01, April, 2010
Published in print edition April, 2010

The book presents several advances in different research areas related to data storage, from the design of a hierarchical memory subsystem in embedded signal processing systems for data-intensive applications, through data representation in flash memories, data recording and retrieval in conventional optical data storage systems and the more recent holographic systems, to applications in medicine requiring massive image databases.

How to reference

In order to correctly reference this scholarly work, feel free to copy and paste the following:

Florin Balasa, Ilie I. Luican, Hongwei Zhu and Doru V. Nasui (2010). Power-Aware Memory Allocation for Embedded Data-Intensive Signal Processing Applications, Data Storage, Florin Balasa (Ed.), ISBN: 978-953-307-063-6, InTech, Available from: http://www.intechopen.com/books/data-storage/power-aware-memory-allocation-for-embedded-data-intensive-signal-processing-applications

INTECH
open science | open minds

InTech Europe
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821

© 2010 The Author(s). Licensee IntechOpen. This chapter is distributed under the terms of the Creative Commons Attribution-NonCommercial-ShareAlike-3.0 License, which permits use, distribution and reproduction for non-commercial purposes, provided the original is properly cited and derivative works building on this content are distributed under the same license.

IntechOpen

IntechOpen