A Spatial Mapping Algorithm for Heterogeneous Coarse-Grained Reconfigurable Architectures

Minwook Ahn, Jonghee W. Yoon, Yunheung Paek
Software Optimization & Restructuring Laboratory, School of EE/CS, Center for SoC Design Technology
Seoul National University, South Korea
{mwahn, jhyoon}@compiler.snu.ac.kr, ypaek@ee.snu.ac.kr

Yoonjin Kim, Mary Kiemb, Kiyoung Choi
Design Automation Laboratory, School of EE/CS
Seoul National University, South Korea
ykim@poppy.snu.ac.kr, kiemb@marykiemb.net, kchoi@azalea.snu.ac.kr

Abstract

In this work, we investigate the problem of automatically mapping applications onto a coarse-grained reconfigurable architecture and propose an efficient algorithm to solve the problem. We formalize the mapping problem and show that it is NP-complete. To solve the problem within a reasonable amount of time, we divide it into three subproblems: covering, partitioning and layout. Our empirical results demonstrate that our technique produces nearly as good performance as hand-optimized outputs for many kernels.

1. Introduction and previous work

Most popular embedded applications such as MPEG, AC3 and AAC contain repetitive computations. Also we can often discover considerable diversity in their code where many different code patterns are interweaved. To more effectively handle these characteristics, coarse-grained reconfigurable architectures (CRAs) [7] have been studied. The CRA can not only boost performance by exploiting the features of repetitive computations, but also adapt itself to diverse computations by dynamically changing configurations of an array of its internal processing elements (PEs) and their interconnections.

Many conventional CRAs are specialized for SIMD-style computations. They are efficient for data parallelism since they save configuration and cache storage by sharing an instruction for multiple data. But their execution models are limited to the extent that each individual PE cannot execute different instructions independently at the same time. To overcome this limitation, researchers have studied multiple instruction multiple data (MIMD)-style CRAs in which each PE can be configured separately to facilitate processing its own instructions. Since they allow more flexible configurations, they can efficiently cope with a more general form of loop pipelining and loop parallelism [8] through simultaneous execution of multiple iterations of a loop in a pipeline.

Traditionally, PEs in a CRA have been homogeneous; that is, the PEs have regular structures supporting the same computational primitives for parallel execution of multiple instructions with flexible operation scheduling. Such flexibility, however, comes at a price because homogeneous architectures normally require many hardware resources regardless of the characteristics of the application domain.

To alleviate this problem, several CRAs with heterogeneous PEs, which we call heterogeneous CRAs (HCRAs), have been proposed [10]. Earlier study [2] demonstrated empirically that the HCRA may improve performance through reduction of chip size and cycle time with an extra effort of pipelining shared resources.

In this work, we investigate the problem of automatically mapping applications onto a MIMD-style HCRA. Like other CRAs, the performance of a HCRA hinges heavily on a strategy that maps target application onto the PE array so as to exploit the parallelism embedded in an application and computation resources of the hardware. Unfortunately, to the best of our knowledge, little work has been reported in the literature on efficient algorithms to automatically solve the mapping problem for MIMD-style HCRAs.

Probable our work is the most closely related to an earlier work on a CRA, called the resource sharing and pipelining architecture (RSPA) [2], because we also target the RSPA. However, this work is different from ours since they used an ad-hoc approach in which they computed optimal mappings manually for their target machine. Manual mapping is quite time-consuming and error-prone. But, in our work we propose an algorithm to automate the mapping process.

[4], [3], and [5] are another noticeable works related to ours. However, their target machines are different from ours in that our RSPA model supports common resource sharing for heavy computation like multiplication and increase performance using loop pipelining, so their mapping algorithms are not able to be directly applied to our RSPA for maximized performance.

This paper is organized as follows. Section 2 explains the target architecture. Section 3 describes our mapping problem. Section 4 discusses the details of our mapping method. Sections 5 and 6 report experiments and conclude.

2. Target architecture

In this work, we choose the RSPA as our target machine. It has a mesh-based coarse-grained reconfigurable array (RA) of PEs. Each PE is a basic reconfigurable element composed

---

Abstract

In this work, we investigate the problem of automatically mapping applications onto a coarse-grained reconfigurable architecture and propose an efficient algorithm to solve the problem. We formalize the mapping problem and show that it is NP-complete. To solve the problem within a reasonable amount of time, we divide it into three subproblems: covering, partitioning and layout. Our empirical results demonstrate that our technique produces nearly as good performance as hand-optimized outputs for many kernels.

1. Introduction and previous work

Most popular embedded applications such as MPEG, AC3 and AAC contain repetitive computations. Also we can often discover considerable diversity in their code where many different code patterns are interweaved. To more effectively handle these characteristics, coarse-grained reconfigurable architectures (CRAs) [7] have been studied. The CRA can not only boost performance by exploiting the features of repetitive computations, but also adapt itself to diverse computations by dynamically changing configurations of an array of its internal processing elements (PEs) and their interconnections.

Many conventional CRAs are specialized for SIMD-style computations. They are efficient for data parallelism since they save configuration and cache storage by sharing an instruction for multiple data. But their execution models are limited to the extent that each individual PE cannot execute different instructions independently at the same time. To overcome this limitation, researchers have studied multiple instruction multiple data (MIMD)-style CRAs in which each PE can be configured separately to facilitate processing its own instructions. Since they allow more flexible configurations, they can efficiently cope with a more general form of loop pipelining and loop parallelism [8] through simultaneous execution of multiple iterations of a loop in a pipeline.

Traditionally, PEs in a CRA have been homogeneous; that is, the PEs have regular structures supporting the same computational primitives for parallel execution of multiple instructions with flexible operation scheduling. Such flexibility, however, comes at a price because homogeneous architectures normally require many hardware resources regardless of the characteristics of the application domain.

To alleviate this problem, several CRAs with heterogeneous PEs, which we call heterogeneous CRAs (HCRAs), have been proposed [10]. Earlier study [2] demonstrated empirically that the HCRA may improve performance through reduction of chip size and cycle time with an extra effort of pipelining shared resources.

In this work, we investigate the problem of automatically mapping applications onto a MIMD-style HCRA. Like other CRAs, the performance of a HCRA hinges heavily on a strategy that maps target application onto the PE array so as to exploit the parallelism embedded in an application and computation resources of the hardware. Unfortunately, to the best of our knowledge, little work has been reported in the literature on efficient algorithms to automatically solve the mapping problem for MIMD-style HCRAs.

Probable our work is the most closely related to an earlier work on a CRA, called the resource sharing and pipelining architecture (RSPA) [2], because we also target the RSPA. However, this work is different from ours since they used an ad-hoc approach in which they computed optimal mappings manually for their target machine. Manual mapping is quite time-consuming and error-prone. But, in our work we propose an algorithm to automate the mapping process.

[4], [3], and [5] are another noticeable works related to ours. However, their target machines are different from ours in that our RSPA model supports common resource sharing for heavy computation like multiplication and increase performance using loop pipelining, so their mapping algorithms are not able to be directly applied to our RSPA for maximized performance.

This paper is organized as follows. Section 2 explains the target architecture. Section 3 describes our mapping problem. Section 4 discusses the details of our mapping method. Sections 5 and 6 report experiments and conclude.

2. Target architecture

In this work, we choose the RSPA as our target machine. It has a mesh-based coarse-grained reconfigurable array (RA) of PEs. Each PE is a basic reconfigurable element composed

---
of an ALU and a barrel shifter. The configuration of a PE is controlled by its own configuration cache. Each row of the PE array shares read/write-buses. When kernels – time-consuming sections of code mostly nested in important loops in the application – are mapped onto an RA, loop pipelining is used for performance enhancement. Because loop pipelining distributes the same operations over several cycles, we do not need all PEs to have the same functional resource at the same time. This allows the PEs in the same row or column to share area-critical resources. From a runtime profile, it can be known how area-dominant and time-critical heavy resources like multipliers are separated as independent PEs, and shared by other PEs in the same row or column array within the CRA. Figure 1 shows the details of connections for multiplier sharing. The two n-bit operands of a PE are connected to the bus switch. At run time, the mapping control signal from every configuration cache is fed to the switch which decides where to route the operands. After a multiplication, the 2n-bit output is transferred from the multiplier to the original issuing PE via the bus switch.

In general, mapping algorithms can be classified largely into two categories: temporal and spatial mapping. The effectiveness of mapping strategies varies depending on the architectural characteristics of the target machine as well as target applications. Figure 2 compares the two mapping strategies. In temporal mapping, necessary configurations are piled on each configuration cache and the configuration of each PE is dynamically changed with time. In Figure 2, only one PE has four configurations and with time its configuration is changed from A to D. In spatial mapping, each PE has a fixed configuration during whole computation, and the data to be processed are flowed through PEs. If (a) in Figure 2 is loop body and there is no loop carried dependency from D to A, we can issue next iteration every single cycle by loop pipelining.

Temporal mapping may reduce the number of PEs required for mapping in comparison with spatial mapping, so relatively large kernels can be mapped onto the RA. Even in this case the maximum number of instructions in an application is bound by the size of configuration cache and the RA. In spatial mapping, the mapping is limited by the topology and size of the RA. Therefore, relatively small or medium-sized kernels should be mapped onto the RA. But it has fixed configuration, so it has no overhead due to dynamic configuration and saves the storage for configuration cache. Capitalizing on these advantages, the spatial mapping strategy is effective for certain types of loops found in many embedded applications.

Figure 1. 4x4 Reconfigurable Array with 8 shared resources

Although the number of multipliers can be reduced through sharing, it may increase the critical path delay, hence degrading the overall performance. To curtail the delay, the RSPA architects employed the resource pipelining technique. In this work, we use the same architecture configuration as in Figure 1 with multipliers in each side of row.

In our work, we summarize all configurations for a given kernel in another binary tree form, called kernel tree. Kernel tree \( K \) is a binary tree where each node is an atomic operation such as + and *, and an edge \((u,v)\) in \( K \) represents a dependence between \( u \) and \( v \). Figure 6 shows an example of the kernel tree where \( L \) denotes a load operation.

One or more operations in \( K \) can be scheduled together on a PE. This scheduling is directed by a configuration which is a set of signals consisting of selection signals of mux and control signals of ALU, shifter and multiplier in the PE. Figure 3 shows the internal structure of a PE and the possible configurations. Configuration 5, LL-ALU-SHT in Figure 3 directs which load operations read values from the frame buffer and how ALU and shift operations serially process these values. To map kernel operations on the PE array, we determine the configuration of each PE in the array by specifying how operations in \( K \) are grouped and scheduled to PEs. To explain this, consider Figure 4 (a) where we can see that three operations (1 multiplication and 2 loads) directed by Configuration 7 are scheduled to PE1. In our work, we summarize all configurations for a given kernel in another binary tree form, called the configuration tree. In a configuration tree, each node represents a configuration for each PE which can cover and execute more than one operations in the kernel, and an edge \((p,q)\)
represents the dependence between the two PEs covering p and q, respectively. Figure 4 (b) shows an example of a configuration tree built from the kernel tree in Figure 6.

Figure 3 Internal structure of PE and a set of configurations

Figure 4 Kernel tree covered by a set of configuration (a) and configuration tree generated from the kernel tree (a) (b)

Once a configuration tree is constructed, each node PEi is now mapped to a physical PE Mjk located at the j-th row and k-th column of the 2-dimensional PE array. Figure 5 shows how the eight nodes in a configuration tree in Figure 4 (b) is laid onto a 5×4 array. For instance, two nodes PE4 and PE6 are respectively scheduled to M12 and M11 in the PE array; thus forming an execution path (M12,M11). Note in Figure 5 that all PEs included in the layout are either white- or gray-colored. We call the white PE a computation PE because it is involved in actual data computation for a kernel. We call the gray one a channel PE because it is not participated in actual computation but used as a communication channel for data transfer between its neighboring PEs. For example, see the computation PE1→PE5→PE6 in Figure 4 (b) that was originally mapped onto the PE array, forming the execution path (M12,M11,M31). To transfer data from PE1 to PE5, we must pass through M31. So we add it as a channel PE in the execution path; thus, forming (M12,M31,M21,M11) in Figure 5.

When multiple nodes are laid, their interdependences are preserved by allocating them to the PEs that have direct or indirect data paths in the array. There are also additional constraints that should be satisfied to correctly lay nodes in the PE array. We list them in Definitions 1, 2 and 3. These constraints collectively called the RA constraints.

Definition 1. [Bus constraint] Configurations of n PEs in one row or one column share a fixed number of memory buses. Let PEj,k be the k-th PE in the j-th row. Then we denote $c_{j,k}$ to be a configuration for a PEj,k and $m_{j,k}$ the number of atomic memory operations in $c_{j,k}$. Then, we have

$$\sum_{c_{j,k}} m_{j,k} \leq \# \text{ of memory buses in } j - \text{th row}$$

Definition 2. [Resource constraint] Heavy computation resources like multipliers are shared by configurations of n PEs in one row, as explained in section 2. Let $c_{j,k}$ be a configuration for PEj,k, and $s_{j,k}$ the number of atomic shared operations in $c_{j,k}$. Then, we have

$$\sum_{c_{j,k}} s_{j,k} \leq \# \text{ of shared computation resources in } j - \text{th row}$$

Definition 3. [Mesh size constraint] Let’s assume that the PE array of target architecture has a dimension of $m \times n$. The number of PEs to which a configuration is assigned in the same row is bound by $n$. Likewise the number of PEs to which a configuration is assigned in the same column is bound by $m$.

Figure 5 Configuration tree layout on PE array for configuration tree in Figure 5

We define the latency of $M_{j,k}$ in the PE array, denoted by $l_{j,k}$, to be the number of cycles within which $M_{j,k}$ completes a processing directed by a configuration $c_{j,k}$. Then, the latency of an execution path $P$ can be defined

$$\sum_{c_{j,k} \in P} l_{j,k}$$

The total cycle time that takes to complete an execution of a kernel is proportional to the length of longest execution path in a configuration tree layout for the kernel. In Figure 5, we see that the longest execution path for the kernel in Figure 6 (a) would be $(M_{11},M_{31},M_{21},M_{11})$ with latency of five cycles since $l_{11} = l_{21} = l_{31} = 1$ and $l_{41} = 2$. We call this longest path, the critical path of the tree layout. To goal of our spatial mapping problem (SMP) is, given a kernel code, to find its
configuration tree layout on the PE array whose the length of critical execution path ($S_{critical}$) is minimized subject to the RA constraints.

Definition 4. **Given a SMP instance ($S_{critical}$) and constant $k$, the problem decision–SMP problem determines whether or not there is a configuration tree layout on PE array that produce $S_{critical}$ such that $S_{critical} \leq k$.**

Shields [6] proved that the problem of deciding if an arbitrary binary tree can be laid out in a 2-D grid with fixed dimensions is NP-Complete. From this, we have learned that SMP is NP-complete. Therefore, we simplify the original problem into three sub problems, covering, partitioning, and layout, each of which will be tackled individually in a separate phase, as discussed in the following section.

4. Solving SMP in three phases

4.1 Covering

Covering groups the nodes of a kernel tree into a set of configurations, and generates a configuration tree $C$. To minimize $S_{critical}$, we attempt to minimize the number of nodes in $C$ based on the intuition that there would be more likely a chance to have a shorter critical path with a less number of nodes in $C$. From this observation, we found that the covering problem is in fact to analogous to the widely-known instruction selection problem [1]. So, in this work, we have implemented a compiler that builds a kernel tree from the source code and applies dynamic programming to find an optimal solution to the covering problem. Figure 6 and Figure 4 actually display the input and output of our compiler. That is, our compiler takes as input the kernel code in Figure 6 (a), computes the covers for nodes in the kernel tree as shown in Figure 4 (a), and produces as output the configuration tree in Figure 4 (b).

4.2 Partitioning

In this phase, we partition the nodes in a configuration tree into different clusters each of which will be scheduled later to each column of the PE array. In this partitioning process, we enforce the RA constraints. As the result of this process, a partition graph $P=(N_P,E_P)$ is an undirected weighted graph built from configuration tree $C=(N_C,E_C)$. Node $p \in N_P$ is a set of configurations satisfying the RA constraints. $P$ has an edge $(p,q)$ if $(u,v) \in E_C$ for any node $u \in p$ and $v \in q$. The weight on $(p,q)$ is the total number of edges in $C$ between nodes in $p$ and nodes in $q$.

Note that in the partition graph, the weight of an edge $(p,q)$ represents the total amount of data traffic between every node in $p$ and every node in $q$.

The goal of this phase is to find an optimal partition graph $P$ that minimize the size of $N_P$ subject to the RA constraints. Since there are $O(2^n)$ possible partitions from $C$ with $n$ nodes, we use an integer linear programming (ILP) solver to find $P$. Notice that $P$ does not guarantee to minimize $S_{critical}$. But, we believe that $P^*$ will helps to reduce $S_{critical}$ since $|N_P|$ is roughly proportional to $S_{critical}$. Figure 7 shows our formula for the ILP solver glpk 4.8. Figure 7 (a) shows the objective function. Figure 7 (b) shows the must-schedule constraint that all configurations should be located on PE array just once. Figure 7 (g) shows the ordering constraint that all configurations are enforced to be located from the first row. All the other constraints are the RA constraints. Figure 7 (h) is the constraint that enforces the maximum number of configured PE in one row to the number of PEs in one row.

Figure 8 (a) shows how the configuration tree in Figure 4 (a) is partitioned by our ILP formula. The figure shows that we have five partitions as the result, producing the partition graph in Figure 8 (b). We can see that each edge is labeled with weights. For instance, the weight of edge (C,D) is 1 since there is only one dependence edge between the node PE3 in C and PE4 in D, meaning that one word of data should be transferred from C to D.

4.3 Laying-out

In this phase, we schedule every node in the configuration tree onto the 2-D PE array and build interconnections between the configured PEs. In this process, we attempt to minimize $S_{critical}$. This phase is performed in two steps: vertical assignment and horizontal assignment. In the vertical assignment, every single partition $p$ in the partition graph $P$ is assigned to each row $r$ of the PE array. Then, in the horizontal assignment, all configuration nodes in $p$ are scheduled to each PE within the row $r$.

We can view an $m \times n$ PE array $(M_{ij})$ as a vertical linear list of $m$ rows ($r_1,r_2,...,r_m$)\(^T\). Therefore, in order to assign the $m'$ partitions $(p_1,p_2,...,p_{m'})$ in $P$ to each row in the linear list, we...
should consider approximately $\Omega(m^2)$ possible assignments subject to the constraint $m' \leq m$. Luckily, $m$ is in practice no larger than 6 for most applications which is appropriate to spatial mapping method. Thus, we decide to find an optimal assignment using the ILP solver.

Suppose that partition $p_i$ is assigned to row $r_i$, and $p_j$ to $r_j$. Then, we define the distance $d_{ij}$ between $p_i$ and $p_j$ to be $|j-i|$. Let $w_{ij}$ be the weight on edge $(p_i,p_j)$ in $P$. Now, we define the data transfer cost $t_{ij}$ between $p_i$ and $p_j$ to be $d_{ij} \times w_{ij}$. The goal of our ILP solver for vertical assignment is to minimize

$$\sum \sum d_{ij} \times w_{ij}$$

where $w_{ij} = 0$ if $i = j$ or there exists no edge $(p_i,p_j)$ in $P$. To compute this with the ILP solver, we define two variables $x_{ij}$ and $y_{ij}$ for ILP formulation as follows.

$$x_{ij} = \begin{cases} 1 & \text{if } p_i \text{ is on } r_i \\ 0 & \text{otherwise} \end{cases}$$

$$y_{ij} = \begin{cases} w_{ij} & \text{if } \exists (p_i,p_j), w_{ij} \text{ is edge weight of } (p_i,p_j) \\ 0 & \text{otherwise} \end{cases}$$

Using these variables, we formulate the objective function with a constraint that enforces a partition to be assigned into only one row.

$$\sum \sum k_{x_{ij}} - \sum \sum l_{x_{ij}} \times y_{ij} \quad (a)$$

$$\forall i \sum m \times x_{ij} = 1 \quad (b)$$

From this formula, the ILP solver finds an optimal vertical assignment that minimizes the total data transfer cost among the rows in the PE array. For instance, given the partition graph in Figure 8 as input, the solver would output the optimal assignment $\{A=r_1, E=r_2, D=r_3, C=r_4, B=r_5\}$, as shown in Figure 9 (a).

![Figure 9 Enumerated PE tree by partition information](image)

Note from Figure 9 (a) that we create in the graph two channel PEs, CPE1 and CPE2, to make data transfer paths respectively thru the row $r_2$ between PE2 and PE8, and thru the row $r_4$ between PE1 and PE5. Our ILP decision always tends to assign two partitions with heavy data traffic as close as possible, which usually leads to minimizing the number of channel PEs that are to be newly introduced in the PE array. Consequently, this decision would help to reduce $S_{critical}$. In this example, the latency of the critical execution path $\text{PE1} \rightarrow \text{PE5} \rightarrow \text{PE6}$ is increased by one due to the intervening node CPE2. However, this result would be still better than other naïve assignments such as $\{A=r_1, B=r_2, C=r_3, D=r_4, E=r_5\}$.

### 5. Experiment

In our experiment, we target the $6 \times 6$ RSPA with 36 PEs where pair-wise connections and cross connections are inserted between not-neighboring PEs. Also, as in Figure 1, two multipliers (thus, 12 multipliers in total) are attached to each row as shared computation resources. Detailed parameters for the target RSPA are listed in Table 1.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PE structure</td>
<td>Bit width 16</td>
</tr>
<tr>
<td></td>
<td>Register file (# of registers) 1(4×16bits)</td>
</tr>
<tr>
<td></td>
<td>Latency 1 cycle</td>
</tr>
<tr>
<td>Shared Computation (SC) resource</td>
<td>Operation multiplier</td>
</tr>
<tr>
<td></td>
<td>Latency 2 cycles</td>
</tr>
<tr>
<td></td>
<td># of SC resources per row 2</td>
</tr>
<tr>
<td>Memory access resources per row</td>
<td># of read bus 2</td>
</tr>
<tr>
<td></td>
<td># of write bus 1</td>
</tr>
<tr>
<td></td>
<td>Latency 1 cycle</td>
</tr>
</tbody>
</table>

Table 1. Architectural parameters of the target architecture

Table 2 shows common operation types and frequencies of certain operations encountered in each kernel for our
experiment. The performance of our algorithm is compared with hand optimized spatial mappings. Because most kernel codes are loops, we first measured the one-iteration latency. Table 3 indicates that our algorithm produces almost optimal results for most cases.

<table>
<thead>
<tr>
<th>kernels</th>
<th>operation type</th>
<th># mult</th>
<th># load/store</th>
</tr>
</thead>
<tbody>
<tr>
<td>n_complex_update</td>
<td>add/sub/mul</td>
<td>4</td>
<td>6/2</td>
</tr>
<tr>
<td>FFT</td>
<td>add/sub/mul</td>
<td>4</td>
<td>12/4</td>
</tr>
<tr>
<td>state</td>
<td>mul/add</td>
<td>8</td>
<td>9/1</td>
</tr>
<tr>
<td>hydro</td>
<td>mul/add</td>
<td>3</td>
<td>3/1</td>
</tr>
<tr>
<td>inner_product</td>
<td>mul/add</td>
<td>1</td>
<td>4/2</td>
</tr>
<tr>
<td>mpeg2enc(dist2)</td>
<td>mul/add/shft</td>
<td>1</td>
<td>6/1</td>
</tr>
<tr>
<td>motion(bdist)</td>
<td>add/sub/shft</td>
<td>1</td>
<td>10/1</td>
</tr>
</tbody>
</table>

Table 2 Kernel codes in the experiments

<table>
<thead>
<tr>
<th>kernels</th>
<th>hand opt</th>
<th>algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>n_complex_update</td>
<td>6×5</td>
<td>5×4</td>
</tr>
<tr>
<td>FFT</td>
<td>4×5</td>
<td>6×5</td>
</tr>
<tr>
<td>state</td>
<td>5×4</td>
<td>5×5</td>
</tr>
<tr>
<td>hydro</td>
<td>2×4</td>
<td>2×4</td>
</tr>
<tr>
<td>inner_product</td>
<td>1×2</td>
<td>1×2</td>
</tr>
<tr>
<td>mpeg2enc(dist2)</td>
<td>3×3</td>
<td>3×3</td>
</tr>
<tr>
<td>motion(bdist)</td>
<td>6×5</td>
<td>6×5</td>
</tr>
</tbody>
</table>

Table 3 One iteration latency in cycle

Now, we compare the total loop latency taken to run each kernel completely. Figure 11 shows the relative execution time of our outputs as compared to hand optimized outputs. For both hand and algorithm versions, we stripped [12] the loops whenever it is possible. For instance, if a kernel is mapped onto the PE array and a half of the PE array is still free, then we use the free space by duplicating the loop iteration space to run each half of the loop iterations simultaneously on the PE array. As an example, see the two kernels in Table 4: inner_product and hydro. They only take up about a sixth and a third of the 6×6 PE array, respectively. Therefore, we were able to stripmine inner_product six times and hydro three times and achieve speedup for both kernels. Overall, Figure 11 shows that our technique produces near-optimal quality mappings for all our kernel codes. Although the performance of kernel mapping is deeply related to the feature of the kernel codes, we conclude that our algorithm performs fairly well for many embedded applications.

6. Conclusion

We showed that finding an optimal spatial mapping of applications onto a HCRA is an extremely complex problem. To circumvent this complexity, we split the original problem into three subproblems each of which is attacked step-by-step in a separate phase. As can be expected, we discover that our algorithm produces comparable performance when being compared to hand optimizations.

7. References


