An Algorithm For Mapping Loops Onto Coarse-Grained Reconfigurable Architectures

Jong-eun Lee[†]
jelee@poppy.snu.ac.kr

Kiyoung Choi[†]
kchoi@azalea.snu.ac.kr

Nikil D. Dutt[‡]
dutt@cecs.uci.edu

[†] EECS, Seoul National University, Seoul 151-742 KOREA
[‡] Center for Embedded Computer Systems, University of California, Irvine, CA 92697

ABSTRACT

With the increasing demand for flexible yet highly efficient architecture platforms for media applications, there is a growing interest in the Coarse-grained Reconfigurable Architectures (CRAs). While many CRAs have demonstrated impressive performance improvement, the lack of compilation technology for such architectures causes a bottleneck in the current design process. In this paper, we present a novel mapping algorithm designed to support Reconfigurable ALU Array (RAA) architectures, that represent a significant class of CRAs. More specifically we present a core mapping algorithm that addresses the problem of placing and routing the operations of a loop body onto the ALU array, to be executed in a loop pipelined fashion. Experimental results using our mapping algorithm on a typical RAA show that our algorithm not only has very fast compilation time but can also generate quality mappings exhibiting high memory bandwidth utilization and low global interconnection requirements. Comparison with manual mapping also indicates that our algorithm can generate near-optimal mappings for several loops.

Categories and Subject Descriptors

B.1.4 [**Microprogram Design Aids**]: Languages and compilers; B.7.2 [**Design Aids**]: Placement and routing

General Terms

Algorithms, Design

Keywords

Coarse-grained reconfigurable architecture, mapping algorithm, ALU array, memory bandwidth utilization

1. INTRODUCTION

Coarse-grained Reconfigurable Architectures (CRAs) [1], [2] are drawing more and more attention as we see the soaring difficulty and costs with custom hardware solutions. However, while CRAs have the potential to exploit both the hardware-like efficiency and the software-like flexibility, the lack of adequate compilation technology to efficiently map the applications (typically loops) onto CRAs is causing delay in the widespread use of the CRAs. For the Reconfigurable ALU Array (RAA) type of reconfigurable architectures [2]–[4], the difficulty of mapping comes from the idiosyncrasies of the architecture (e.g., efficient utilization of the abundant ALUs, overcoming the limited memory bandwidth, and dealing with the fixed ALU array size) as well as the characteristics of the application programs (e.g., loops with loop-carried dependency). To overcome the limitations and maximize the performance, some mapping subproblems have been addressed in the literature (e.g., *temporal partitioning* for fixed array dimension and *pipeline vectorization* for throughput improvement [5], *memory operation sharing* for limited memory bandwidth [6], and *data context switching* for loop-carried dependency [7]). However, the core mapping problem of placing and routing the operations of a loop body onto the ALU array in the context of CRAs is one that still requires much effort of research.[1]

The mapping algorithm we present in this paper employs a generic reconfigurable architecture template called DRAA (Dynamically Reconfigurable ALU Array) as the target architecture. The DRAA [6] can represent a broad range of reconfigurable ALU array architectures with a number of parameters, allowing for analysis, evaluation, and exploration of different CRAs tuned towards specific application domains. Such architectures often have a performance bottleneck in the memory subsystem interface especially for media and telecommunication applications; hence, we are developing compilation technologies that focus on memory traffic optimization. In this paper we describe a mapping algorithm, which tries to maximize the utilization of the memory bandwidth (allocated to each line of the array). It employs a two-step approach: first clusters the operations of a given loop to generate line-level placements and then combines the line placements at the plane level. By dividing the 2D mapping problem into the two phases, our algorithm can achieve not only fast compilation time but also very high utilization of the limited memory interface resources. Experimental results using our mapping algorithm on a typical coarse-grained architecture show that our algorithm can generate quality mappings exhibiting high memory bandwidth utilization, low global interconnection re-

[1]While some companies claim they have a C compiler for their ALU array type of reconfigurable architecture (e.g., [5]), the details are hardly available. Also, many so-called "compilers" for CRAs typically use hand-crafted libraries to avoid the challenging task of mapping loops onto the CRAs.

LCTES'03, June 11–13, 2003, San Diego, California, USA.

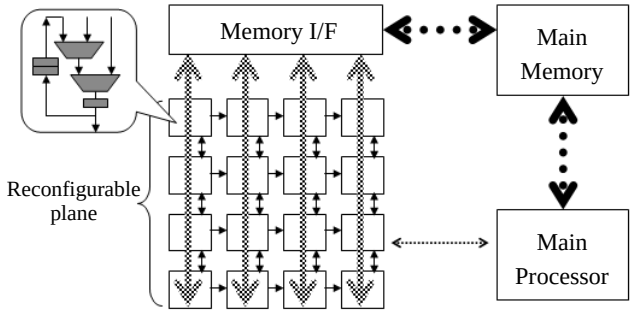

Figure 1: DRAA: Generic reconfigurable architecture template.

quirements, etc. Also, comparison with manual mapping indicates that our algorithm can generate near-optimal mappings for several loops.

The rest of the paper is organized as follows. In Section 2 we introduce the DRAA (the target architecture of the mapping algorithm) and loop pipelining, which are assumed in our compilation flow. We then explain the mapping algorithm in Section 3. We present the experimental results with discussions in Section 4 and conclude the paper in Section 5.

2. BACKGROUND

In this section we first introduce the target architecture, the DRAA, and also identify some architectural constraints that are required by the mapping algorithm. Next, we briefly explain loop pipelining, the most commonly used technique for exploiting performance in mapping loops onto reconfigurable architectures.

2.1 Target architecture

We develop our compilation flow for a generic reconfigurable architecture template called DRAA [6], which can represent a broad range of reconfigurable ALU arrays (e.g., [3], [4], [8]). The DRAA consists of two parts: the computation part and the configuration part. The computation part consists of the PEs (Processing Elements) placed in a 2D array, interconnections between them, and high-speed memory interface as illustrated in Figure 1. The configuration part, which includes the context registers and the configuration cache, provides the computation part with the control signals necessary for opcodes, interconnections, and immediate values.

PEs are word-level configurable functional blocks (CFBs), each of which can be programmed to perform a function or operation pattern selected by the configuration. The interconnections between PEs can include buses (for global communication within the same row or column) and dedicated interconnections on a row (column). While the DRAA does not restrict the interconnection architecture except that all rows (columns) should have the same interconnection scheme, our mapping algorithm requires that the interconnection scheme should be invariant regardless of the position within the row (column).[2] Similarly to the row/column interconnections, the memory interface is also assumed to be identical for all rows (columns). Typically shared buses are used as the memory interface to each PE (see Figure 1) and laid out in row (column) direction. The set of PEs and buses laid out on a row (column) is referred to as a *line* (direction). So the PEs on the same line will share a fixed memory interface.

[2]This is to ensure the line placement found in the clustering phase can be translated along the line later at the combining phase.

2.2 Loop pipelining

Our mapping technique assumes that the loops are mapped and executed on a reconfigurable architecture using loop pipelining [9], [10], in which multiple iterations of a loop are executed simultaneously in a pipeline. So, to map a loop onto a reconfigurable architecture, one only needs to find a mapping for one iteration and the same mapping applies to the other iterations as well. This technique can lead to the maximum throughput (which is one iteration per cycle assuming the delay of a PE is one cycle) when there is no loop-carried dependency. But when there is loop-carried dependency, successive iterations cannot enter the pipeline every cycle, which may result in a very inefficient mapping in terms of the resource usage.

Data context switching [7] addresses this problem in loops with loop-carried dependency, assuming there is an outer loop with no loop-carried dependency (an outer loop with independent iterations). This technique can effectively eliminate the dependency by switching between the independent data sets provided by the independent iterations of the outer loop. Thus, this technique requires relatively large memory space in the reconfigurable architecture to store the multiple data sets, which further highlights the critical importance of mapping techniques optimizing memory bandwidth utilization.

3. MAPPING ALGORITHM

We now explain our mapping algorithm. First we explain our mapping strategy together with the overall compilation flow. Then we explain the two phases of the algorithm: clustering and combining.

3.1 Overview

The overall compilation flow (Figure 2) employs two paths similar to the traditional hardware/software codesign flow: the DRAA is used for the selected loops and the main processor is for the rest of the application program. For the DRAA part, the loops are given to the DRAA mapper, which has at least three basic phases: PE-operation covering, clustering, and combining. The PE-operation covering generates PE-operation trees by covering the expression trees of the input loop description with PE-operations. PE-operation is a representation of an operation pattern supported by a single configuration of a PE. The other two phases constitute the core of our mapping algorithm and correspond to the placement and routing (P&R) step in the conventional hardware synthesis flow.

A unique strategy of our mapping algorithm is to partition the input PE-operation tree into clusters (each of which is to be placed on a line, i.e., a row or a column) with only one output. This clustering scheme has many benefits. First, it reduces the complexity, since only 1D placement needs to be done instead of 2D placement. Second, deciding first the set of PE-operations for each line makes it easy to maximize the utilization of the fixed memory bandwidth, when a common memory interface (such as bus) is shared between the PEs on a line. Third, restricting the clusters to have only one output helps the ensuing phase of combining line placements to easily find the routing resources for inter-cluster data transfers. Since failure to find the routing resources will result in more than one configuration to be used (implying significant performance loss), restricting the clusters to have one output at most is very important.

3.2 Clustering

The clustering phase groups the nodes of a PE-operation tree into clusters (each of which has one output at maximum) such that each cluster can be placed on a line, for the given line interconnect architecture. Note that placing on a line imposes restrictions not only

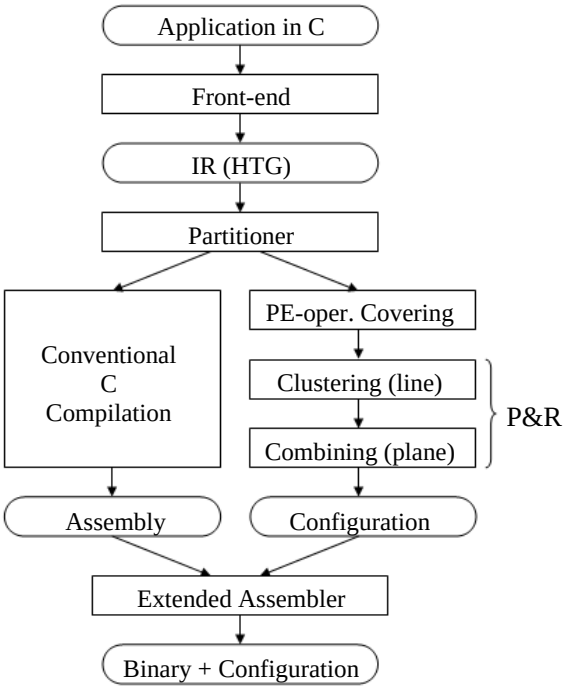

Figure 2: Compilation flow.

```
algorithm line-clustering
 1: list-of-cluster clusters, next-level-clusters
 2: for l = maxlevel downto 0 do
 3:    for all node n at level l do
 4:       new-cluster = make-a-cluster(n, clusters)
 5:       add new-cluster to next-level-clusters
 6:    end for
 7:    clusters = next-level-clusters
 8:    clear next-level-clusters
 9: end for
end line-clustering

algorithm make-a-cluster(n, clusters)
 1: cluster-pair = get-children-clusters(n, clusters)
 2: if is-all-placeable(n, cluster-pair) then
 3:    return merge-clusters(n, cluster-pair)
 4: else if sel = is-any-placeable(n, cluster-pair) then
 5:    return expand-a-cluster(n, cluster-pair, sel)
 6: else
 7:    return create-a-cluster(n)
 8: end if
end make-a-cluster
```

Figure 3: Clustering algorithm.

on the topology of the cluster (the topology should be supported by the line interconnection scheme) but also on the total number of memory operations in the cluster (which should be no greater than the number of memory buses for a line). We will refer to these restrictions as the *line placement condition*.

The basic idea of the clustering algorithm shown in Figure 3 is to grow the clusters in a bottom-up fashion so that they can cover as many nodes as possible before they violate the line placement condition. The algorithm starts from the nodes of the greatest level (line 2). Only after the nodes of one level are all covered by some clusters, the algorithm starts covering the nodes of the next level by either increasing the existing clusters or creating a new one (lines 3–6). Eventually, every node is covered by a cluster.

The way the algorithm finds a cluster to cover a node N is as follows (see the algorithm make-a-cluster in Figure 3). Note that the children nodes of N (if any) are already covered by some existing clusters, which will be called the clusters of the children nodes (line 1). The following three possibilities are tried in the order they are presented.

- First, it merges the clusters of the children nodes and adds N to it, if the resulting cluster satisfies the line place condition (lines 2–3).
- Second, it expands a cluster of the children nodes to add N, if the expanded cluster satisfies the line placement condition. If there is more than one satisfying the condition, the one with the smallest number of memory operations or (in case of tie) smallest number of PEs is selected (lines 4–5).
- Third, a new cluster is created to include only N (lines 6–7).

In the 2nd and 3rd cases above, some of the clusters of the children nodes of N may not include N. These clusters are called *closed clusters*, since they have no chance of including any other node later on. Then there is a data transfer relation between a closed cluster and the cluster covering N, meaning that the output of the closed cluster should be connected to an input of the cluster covering N in the combining phase. So the links to the closed clusters are stored in the cluster covering N for later references.

Throughout the clustering process, the line placement condition is checked every time when a new node is added to a cluster. Having been designed to support a class of reconfigurable architectures (with different interconnection schemes), our algorithm expects the line placement checking part in a separate module; therefore, different interconnection schemes of different architectures can be easily accommodated. For many architectures, the line placement checking part typically includes comparing the topology of the cluster (i.e., connectivity between the PE-operation nodes) with the interconnect architecture of a line. The most general version of this problem of matching two graphs may well have an exponential computational complexity. However, since the number of nodes in a cluster is typically small (not greater than 10), clustering the nodes usually does not require excessive running time.

In Figure 4 we illustrate the clustering process for the PE-operation tree shown in (a), as each level of nodes are covered by clusters. The line architecture is assumed to be given such that each column has two memory buses and there is a dedicated interconnection between every pair of PEs with distance[3] of at most 2. In (b) $\sim$ (f), the nodes are covered by increasing the lower-level clusters or creating new ones until every node is covered by a cluster, generating the clusters (with associated line placements) in (f).

3.3 Combining line placements

In the combining phase, the data transfer between the clusters (represented by the links stored in clusters) is mapped to the routing resources (typically nearest neighbor network and buses) in the architecture. Since each cluster has already been placed on a column and has only one output, putting the clusters together connecting their input/output can be done quite easily. We only need to

[3]Distance means the physical proximity of PEs on a row or column, with one unit being that of neighboring PEs.

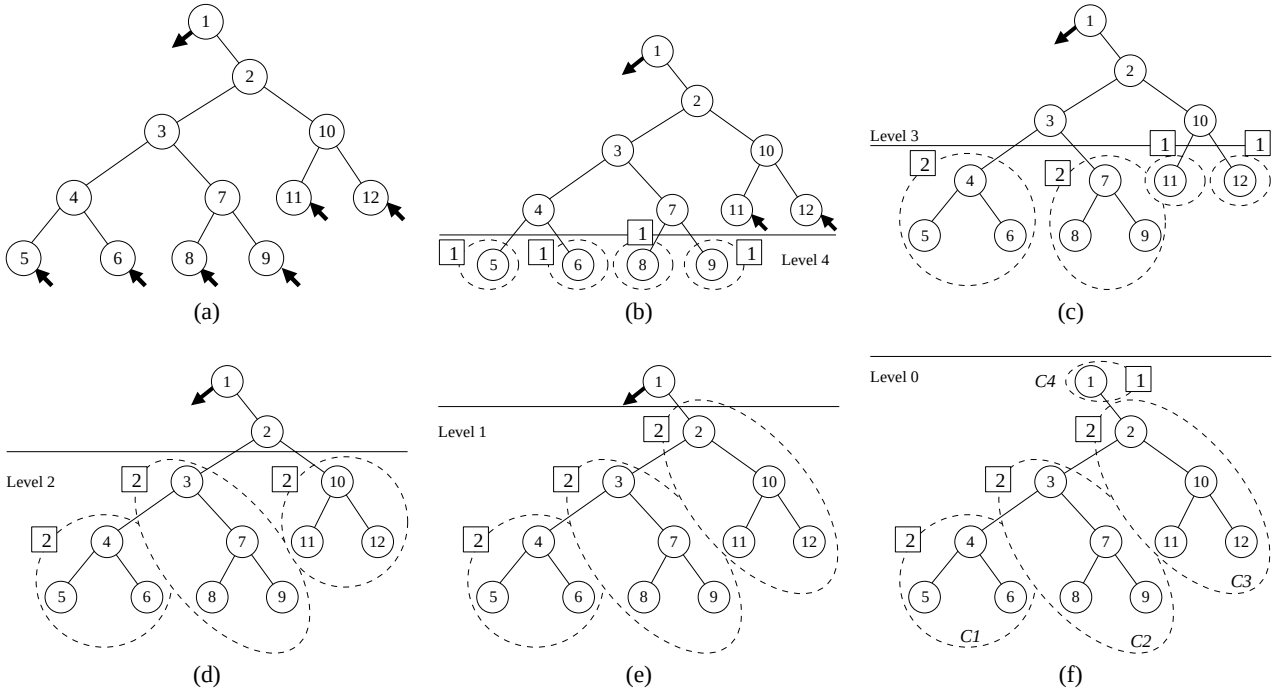

Figure 4: Clustering process. In (a), each node represents a PE-operation and an incoming/outgoing arrow of a node means a memory read/write operation required by the node. In (b) $\sim$ (f), the number in a small box means the number of memory operations in the cluster.

adjust the relative position of the line placements such that the corresponding input and output nodes are placed on the same row to have the horizontal interconnections and buses connect them.

However, the horizontal order of the clusters is not yet determined, and the vertical direction of each placement (whether to flip or not) can be changed to get a better result such as better routability or a smaller dimension in the resulting 2D placement. As the two breeds of the decisions do not seem to be independent, a simple-minded approach to finding the best combination will suffer from the complexity of $O(n! \cdot 2^n)$, where n is the number of line placements.

Our algorithm determines the vertical direction of each line placement by a brute-force algorithm after deciding their horizontal order with a simple heuristic algorithm. The heuristic tries to place the clusters with data transfer relation (link) closer to each other. The link relations can be represented as a tree (with each node representing a cluster), for those derived from expression trees. Thus one simple heuristic is to recursively place the direct children (which may be more than two) of a node at the nearest columns possible, putting a higher precedence on a link at a greater level. Once all those decisions are made, each link is mapped to a routing resource. Though not likely, if it fails to find a routing resource (bus or row-wise interconnection) at a desired location, the algorithm resorts to multiple configurations[4] to complete the 2D placement mapping.

[4]Using multiple configurations is in principle multiplying the resources by the number of configurations. Therefore, although expensive, it can always solve the problems caused by limited resources.

3.4 Complexity Analysis

The time complexity of our algorithm is as follows. First, the line clustering algorithm (Figure 3) invokes the line placement condition checking routine as often as triple the number of PE-operations at maximum. In other words, the complexity of the line clustering algorithm is m times the complexity of the line placement condition checking routine, where m is the number of PE-operations in the input tree. Second, our line combining heuristic visits all the line placements recursively to determine their horizontal order, which has $O(n)$ complexity, where n is the number of the line placements considered. Finally, to determine the vertical direction of each line placement, we employ a brute-force algorithm, which has $O(2^n)$ complexity. Therefore, if the the line placement condition checking routine has the complexity of $O(L!)$, where L is the (maximum) number of PE-operations in a cluster, the overall complexity of our mapping algorithm is $O(m \cdot L! + n + 2^n) = O(m \cdot L! + 2^n)$. In practice, checking the line placement condition is often early terminated due to the number of memory operations violation. Moreover, L (the number of PE-operations in a cluster) and n (the number of clusters) can be limited to the number of PEs in a line and the number of lines, respectively, which are determined from the RAA architecture in question. MorphoSys (M2 chip) [3] or REMARC [8] architectures, for instance, have eight rows and eight columns; in such cases, L and n are limited to 8. Thus, the time complexity of our mapping algorithm is not high in practice and indeed very low as can be seen in our experimental results.

4. EXPERIMENTS

For our experiments we used an example architecture described in [6]. Each column has two memory buses that are shared with global buses, and each row has two global buses. There is an in-

Table 1: Summary of mappings. C: #columns used, M: #memory operations in the loop, %: M/(2*C), R: #rows used, G: #global buses used in the mapping, %: G/(2*R), S: size of the bounding box of the mapping (=C*R), P: #PE-operations of the loop, %: P/S. Note that each column/row has two buses on it.

	Memory bus (vertical)			Horizontal global bus			PE (ALU Array)			
Loop	C	M	%	R	G	%	S	P	%	#Conf
hydro	2	4	100	3	0	0	6	5	83	1
ICCG	3	6	100	2	0	0	6	4	67	1
banded	7	14	100	6	3	25	42	26	62	1
state	5	10	100	4	1	13	20	16	80	1
ADI	6	11	92	4	2	25	24	16	67	1
wavelet	3	5	83	2	0	0	6	4	67	1
MPEG kernel	33	64	97	12	3	13	396	127	32	1

terconnection between every pair of neighboring PEs, both vertically and horizontally. A PE supports all basic arithmetic operations as well as a series of MULTIPLY and ADD operations with one configuration if one of the MULTIPLY's operands is constant. For application loops, we selected seven loops from the Livermore Loops benchmark suite, a wavelet filter algorithm (**wavelet**), and an MPEG2 encoding algorithm (**MPEG kernel**).

Table 1 summarizes the mapping results. First of all, the last column (#Conf) shows the number of configurations assuming the ALU array has a dimension size large enough. So, on an ALU array of 7 columns * 6 rows, all the loops except **MPEG kernel** will be mapped with a single configuration, so that the loops can be executed with the maximum possible throughput (1 iteration/cycle). In other words, there was no unnecessary configuration change due to the routing failure in the combining phase.

To the best of our knowledge, there is no previously reported work that can be referenced for comparison. To assess the quality of the mappings we consider the utilization of the resources in three categories: memory interface (vertical memory bus), global interconnection (horizontal bus), and computation (PE). The data in the table implies the following. First, the generated mappings utilize the memory buses very efficiently; in most cases, only the minimum number of columns is used for the given number of memory operations. Second, they use the global bus very sparingly. Since the horizontal global buses are the last means to complete the intercluster routing before resorting to multiple configurations, it is important to keep this utilization low in this case. Third, the utilization of the PEs is typically over two thirds with exceptions of **banded** and **MPEG kernel**. This shows our algorithm generates relatively compact mappings even in terms of the PE usage.

Comparison between the size of the input (P) and the utilization (%) suggests that our algorithm is scalable for memory bus and global bus utilizations while the PE utilization becomes poorer with a larger input size. However, considering that a typical DRAA has 8 * 8 PEs with 16 memory buses distributed over the PEs, **MPEG kernel** is a very large loop (127 PE-operations and 64 memory operations) and should be divided into several loops for efficient mapping anyway. For the other loops, our mapping algorithm generally maintained PE utilization of over 60%. The compilation time with our mapping algorithm was very short—far less than 1 sec for each loop.[5]

For comparison, we also generated manually optimized mappings for smaller loops. Table 2 summarizes the results, which we think are optimal at least in terms of the metrics that are shown in the table. Comparing with the manual mapping, our algorithm generated mappings with the same resource requirements in terms of the #rows (R) and the #columns (C) as well as the #configurations (#Conf), for all the loops for which manual mappings were generated. Also, in terms of the horizontal global bus requirement (G), our algorithm generated mappings using the same number of global buses as in the manual mappings, except for **ADI**. In summary, the experimental results show that our mapping algorithm can generate near-optimal[6] mappings for several loops found in DSP algorithms.

Table 2: Manual mapping results. C: #columns used, R: #rows used, G: #global buses used in the mapping, S: size of the bounding box of the mapping (=C*R).

Loop	C	R	G	S	#Conf
hydro	2	3	0	6	1
ICCG	3	2	0	6	1
state	5	4	1	20	1
ADI	6	4	0	24	1
wavelet	3	2	0	6	1

5. CONCLUSION

We have presented a novel automatic mapping algorithm for loops with general applicability to the DRAA architecture, a generic architecture template for a broad range of CRAs. To achieve fast compilation as well as very high memory interface utilization, our mapping algorithm has a two-phase organization. The first phase clusters operations to maximize the memory interface utilization while ensuring the line placement of the clusters. The line placements are then combined in the second phase to generate a 2D placement. To minimize the risk of performance loss due to the routing problem in the combining phase we have found it advantageous to limit the topology of clusters in the clustering phase, which renders an efficient and effective algorithm as presented in this paper. Our experimental results show the mapping algorithm can generate high quality mappings (such as high memory bandwidth utilization and low global interconnection requirement) and often near-optimal ones when compared with hand-generated mappings.

The compilation for CRAs involves much more complexity than for traditional microprocessors due to the numerous architectural

[5]For MPEG kernel the result shown is with the vertical direction decision stage skipped, for which our brute-force algorithm requires too much computation time due to the exceptionally large number of columns (33) used.

[6]i.e., close to our best-effort manual mapping results in terms of the metrics considered

features and their inter-dependency. We will examine other aspects of the compilation problem to bring the core mapping algorithm presented here to a more complete compilation flow with management of data memory and configurations among others.

6. ACKNOWLEDGEMENTS

This research was conducted while the first two authors were visiting UC Irvine, and supported in part by grants from NSF (CCR-0203813 and CCR-0205712) and Hitachi Ltd. We also thank members of the UCI EXPRESS compiler team for their assistance.

7. REFERENCES

[1] A. Bindra. Reconfigurable architectures chart a new course for DSPs. *Electronic Design*, pages 46–52, August 5 2002.

[2] R. Hartenstein. A decade of reconfigurable computing: A visionary retrospective. In *Proc. DATE*, pages 642–649, 2001.

[3] H. Singh et al. MorphoSys: An integrated reconfigurable system for data-parallel and computation-intensive applications. *IEEE Trans. Computers*, 49(5):465–481, May 2000.

[4] *Elixent's D-Fabrix*. http://www.elixent.com, last accessed April 24, 2003.

[5] J. Cardoso and M. Weinhardt. Fast and guaranteed C compilation onto the PACT-XPP^{TM} reconfigurable computing platform. In *Proc. FCCM*, pages 291–292, 2002.

[6] J. Lee, K. Choi, and N. Dutt. Compilation approach for coarse-grained reconfigurable architectures. *IEEE D&T*, 20:26–33, January/February 2003.

[7] K. Bondalapati. Parallelizing DSP nested loops on reconfigurable architectures using data context switching. In *Proc. DAC*, pages 273–276, 2001.

[8] T. Miyamori and K. Olukotun. REMARC: Reconfigurable multimedia array coprocessor. In *Proc. ACM/SIGDA FPGA*, page 261, 1998.

[9] M. Weinhardt and W. Luk. Pipeline vectorization. *IEEE Trans. CAD*, 20:234–248, February 2001.

[10] K. Bondalapati and V. Prasanna. Loop pipelining and optimization for run-time reconfiguration. In *Proc. RAW*, 2000.