Compilation Approach for Coarse-Grained Reconfigurable Architectures

Jong-eun Lee and Kiyoung Choi
Seoul National University

Nikil D. Dutt
University of California, Irvine

Coarse-grained reconfigurable architectures can enhance the performance of critical loops and computation-intensive functions. Such architectures need efficient compilation techniques to map algorithms onto customized architectural configurations. A new compilation approach uses a generic reconfigurable architecture to tackle the memory bottleneck that typically limits the performance of many applications.

MANY COARSE-GRAINED reconfigurable architectures have recently emerged as programmable coprocessors, significantly relieving the burden of the main processor in many computation-intensive loops of multimedia applications.[1-5] Their abundant parallelism, high computational density, and flexibility of changing behavior during runtime make such architectures superior to traditionally used DSPs or ASICs. As technology allows increasingly more devices on a single chip, designers can feasibly implement coarse-grained reconfigurable architectures as programmable platforms in complex systems on chips (SoCs).

Typically, a coarse-grained reconfigurable architecture has identical processing elements (PEs), even though there is a wide variance in the number and functionality of components and the interconnections between them. Each PE contains functional units—for example, an arithmetic logic unit (ALU) and a multiplier. Each PE also has a few storage units—for example, a register file and small local memory. Programmable interconnects connect the PEs. Such interconnects support rich communication between neighboring PEs, typically to form a 2D array.

Despite the many proposed coarse-grained reconfigurable architectures,[3-6] there is very little compiler support, so designers typically must do the application mapping manually.[2] However, compiler tools that can handle the idiosyncrasies of reconfigurable architectures and a parameterizable reconfigurable architectural model are crucial to the exploration and evaluation of these architectures for different applications. This article presents a generic reconfigurable architecture that facilitates exploration of coarse-grained architectures, and describes a compilation approach to tackle the memory bottleneck that typically limits the performance of many applications.

Compilation techniques

There is little work published on compilation and mapping techniques for coarse-grained reconfigurable architectures. There has been extensive research on FPGAs and fine-grained reconfigurable architectures, and there are several commercial tools in this area. However, for various reasons, these techniques are not directly applicable to coarse-grained designs.

Bondalapati proposed a granularity-neutral approach for reconfigurable architectures.[7] The proposed data-context-switch technique parallelizes DSP-nested loops with loop-carried dependency (such as an infinite-impulse-response filter) when there is an outer loop without such dependency. Bondalapati also applied this technique to a high-granularity reconfigurable architecture, Chameleon (http://www.chameleonsystems.com). However, the technique assumes that each PE has access to an ample local memory to store the context data for the outer-loop itera-

0740-7475/03/$17.00 © 2003 IEEE Published by the IEEE Computer Society

tions. This assumption might not hold true for many coarse-grained reconfigurable architectures.

Huang and Malik proposed a design methodology for their own dynamically reconfigurable data path architecture.[8] Although this work targets coarse-grained reconfigurable hardware, in their methodology the reconfigurable data path serves a specific application and only to the loops of the application for which it is designed.

Generic template

To further compilation research and design space exploration for a wide class of reconfigurable architectures, we developed a generic architecture template, the dynamically reconfigurable ALU array (DRAA). The DRAA places identical PEs in a 2D array (reconfigurable plane), with regular interconnections between them and a high-speed memory interface, as Figure 1 shows. The DRAA supports three levels:

- PE microarchitecture,
- line architecture, and
- reconfigurable plane architecture.

Table 1 lists the parameters for an example DRAA architecture.

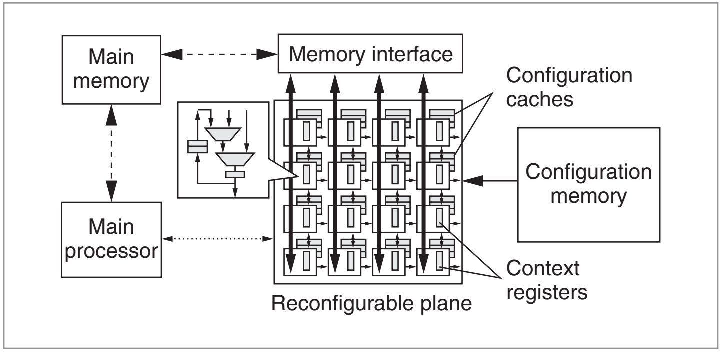


Figure 1. Dynamically reconfigurable ALU array (DRAA), a generic reconfigurable-architecture template.

PE microarchitecture

We can describe the PE microarchitecture using a processor-centric architecture description language such as Expression.[9] For instance, we can describe the data path as a netlist of the components comprising the PE with relevant attributes such as supported opcodes and timing. Alternatively, we can use a set of supported functionalities corresponding to an instruction set as a PE microarchitecture description. We assume that each PE's latency remains constant regardless of its function, configuration, or even data, because variance in a PE's latency can significantly diminish the architecture's regularity.

Table 1. Sample parameters for DRAA architecture.

Category	Parameter	Value
PE microarchitecture	Bit width	16
	No. of register files	1 (4 × 16 bits)
	Operation set	{add, mul, …}
	Latency	1 cycle
Row (line) interconnect	No. of PEs in a line	8
	No. of lines	8
	Set of pairwise interconnections	{(1 2), (2 3), (3 4), …}
	No. of buses	2 (16 bits each)
Memory access resource per line (row)	No. of buses (shared with row global buses)	2 (16 bits each)
	Latency	1 cycle
	Buffer depth per bus	256 × 16 bits
Configuration cache	No. of configurations	8
	No. of configurations in single instruction, multiple data (SIMD) mode use	64
	Fast configuration reloading overhead	0 cycles
	Latency to fill one configuration in configuration cache	8 cycles
Miscellaneous	Direct communication with main processor	None

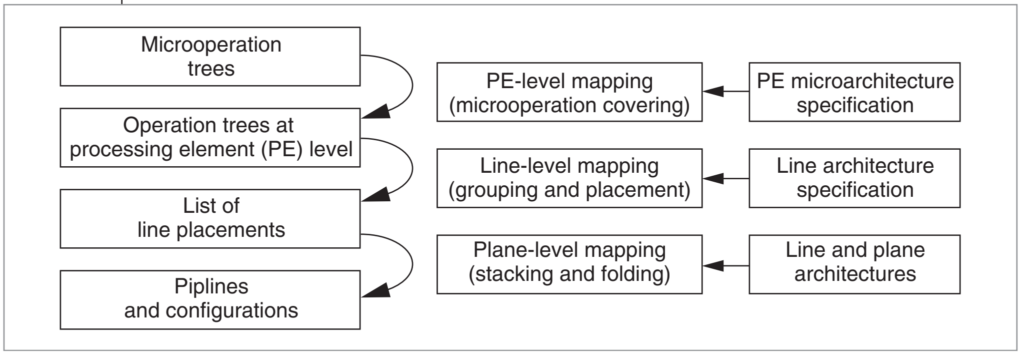

Microoperation trees
Operation trees at processing element (PE) level
List of line placements
Piplines and configurations
PE-level mapping (microoperation covering)
Line-level mapping (grouping and placement)
Plane-level mapping (stacking and folding)
PE microarchitecture specification
Line architecture specification
Line and plane architectures

Figure 2. Three steps in the mapping flow.

Line architecture

We assume that the line architecture includes only row and column interconnections—no diagonal ones. (We refer to a row or a column as a line.) We also assume that all rows and columns have the same interconnections. Therefore, only the interconnections of a row and a column need to be defined. We describe these 1D interconnections as

- a set of dedicated connection pairs,
- the number of global buses, and
- any specialized interconnections.

In addition to the row/column interconnections, the line architecture also includes the memory access resource, which is part of the memory interface. As with the interconnections, we assume that the memory access resources are equally distributed along the rows or columns. For example, Remarc[5] has a 32-bit-wide bus for each column that is usable for transferring four 8-bit or two 16-bit variables. In this case, we specifically refer to a column as a line, and the memory access resource per line is a 32-bit-wide bus.

Reconfigurable plane

Reconfiguration-related parameters are critical for characterizing the plane-level architecture. In many architectures that support a distributed configuration cache for fast runtime configuration switching,[10] parameters such as configuration cache size and dynamic reloading overhead can make a big difference in performance. Compilers also must be aware of the architectural features to avoid a huge penalty when, for example, the generated configurations don't fit in the cache size.

Memory interface

Coarse-grained reconfigurable architectures typically target loops in data-dominated multimedia applications. Thus, they often have specialized memory interfaces, such as frame buffer in MorphoSys[3] and global control unit in Remarc.[5] For application mapping, an important common feature of those architectures is that a group of PEs typically on the same line shares a memory interface. Consequently, a critical performance bottleneck occurs in the shared memory interface resources. Thus, optimizing memory operations of loops can significantly affect the quality of the mapping and the resulting performance. Furthermore, from an architectural point of view, the memory interface parameters are critical for achieving performance and can be tuned to the application's profile.

Mapping flow

We now illustrate a flow for mapping loops onto a typical class of DRAA architectures. To achieve maximal throughput, our approach generates high-performance pipelines for a given loop body so that consecutive iterations of the loop can be executed successively on those pipelines. We generate the pipelines from microoperation trees (expression trees with microoperations as nodes), representing the loop body through the three levels of mapping shown in Figure 2:

- PE (microoperation covering),
- line (operation grouping and placement), and
- plane (stacking of line placements followed by folding with time multiplexing).

PE-level mapping

The PE-level mapping covers microoperation trees with PE-level operations. A PE-level operation is defined as a microoperation pattern that can be implemented with a single configuration of a PE according to the PE microarchitecture. The PE-level mapping process generates PE-level operation trees. The nodes represent PE-level operations (annotated with the number of memory operations it contains), and the directed edges represent data transfers between two nodes. Figure 3 shows an example of PE-level mapping.

Line-level mapping

Following the PE-level mapping is the line-level mapping, which groups the PE-level operation nodes and places them on each line. At this step, two conditions should exist before the nodes are grouped together; the total number of memory operations contained in the nodes should not exceed the capacity of the memory interface bus, and it should be possible to allocate an interconnection resource for each edge (data transfer). Figure 3b shows one possible grouping for the line architecture given in Figure 3c. The line placements satisfying those conditions are passed on to the next step.

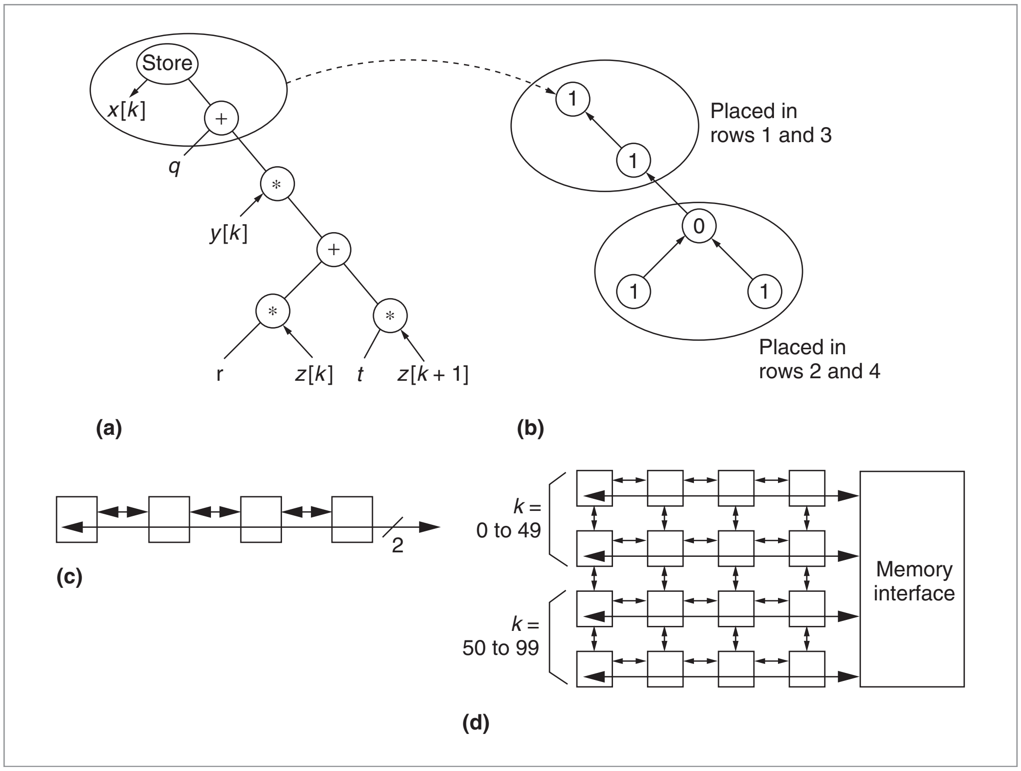


Figure 3. Example of generating pipelines from microoperation trees: microoperation tree (a), PE-level operation tree (b), line architecture (the bus is used as both the memory interface and the line architecture) (c), and plane architecture (d).

Plane-level mapping

The plane-level mapping stitches together the line placements generated in the line-level mapping, on the 2D plane of PEs. In the example of Figure 3b, the line placements come together nicely, resulting in a pipeline within a bounding box of 2 rows × 3 columns. If the physical plane size is 4 rows × 4 columns, the pipeline is replicated in the other two unused rows, as in Figure 3d, roughly doubling the performance. In this example, the generated mapping achieves the maximum throughput in the DRAA architecture while using only 10 of the 16 available PEs, because all the memory I/O resources are used at every cycle. This illustrates the need to pay critical attention to the memory I/O resource.

The mapping flow we present here allows different architectural parameters that enable design space exploration of the DRAA itself. The architectural variabilities supported by the mapping flow include part of the PE microarchitecture (the operations set) and some of the line architecture parameters (the number of PEs, the size of the row/column buses, the size of the memory access buses, and so forth). Such variability yields many interesting design points, with varying performance, in the architecture design space.

Memory operation sharing

Because the loops implemented in DRAAs are often memory-operation bounded, as in the previous example, reducing the effective number of memory operations should boost the application's performance. One such opportunity comes from the data reuse pattern in loops of DSP algorithms. Let's take an example of a finite-impulse-response filter algorithm, defined as

$$y[i] = \sum_{j=0}^{2} w_j \times x[i-j]$$

where w_j is a constant representing the filter coefficient; x and y are the input and output data streams; and $i = 0$, 1, 2, …, n. Figure 4 shows an example of an optimal mapping of this algorithm.

Let's assume that a PE supports a series of MULTIPLY and ADD operations with one configuration if one of the MULTIPLY's operands is a constant. Because the PE-level operation tree has four memory operations, at least two rows are necessary to implement the tree with a single configuration. The optimal mapping in this exam-

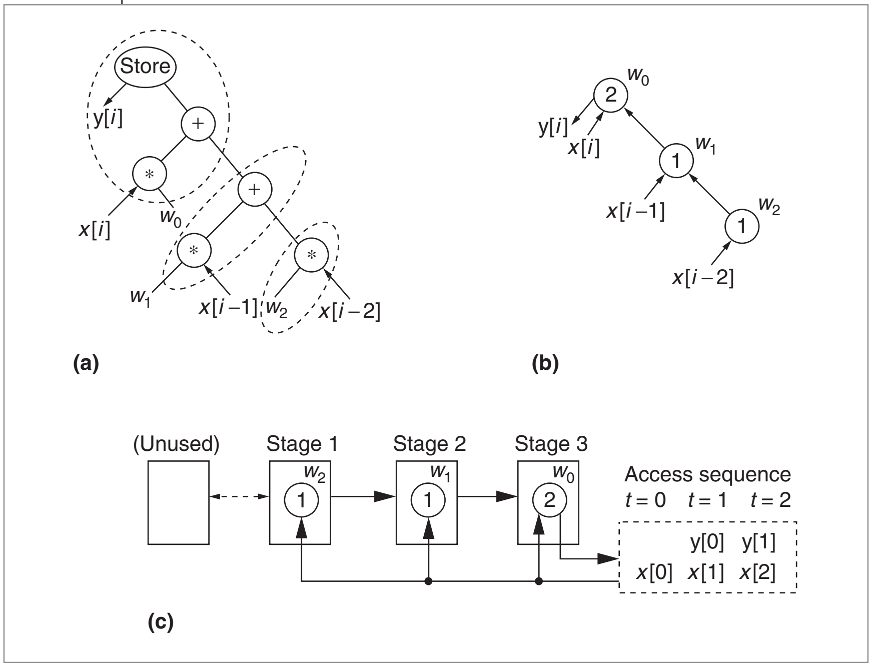


Figure 4. Example of optimal mapping of a finite impulse response: microoperation tree representation of the input algorithm (a), PE-level operation tree after the PE-level mapping (b), and optimal pipeline implementation for the line architecture of Figure 3c (c).

ple, however, is to use only one row (as in Figure 4c), sharing one memory bus for all the read operations of the three nodes. The sharing is possible in this case because the three PEs represent different pipeline stages. For instance, when node w_0 processes iteration $i = 0$, nodes w_1 and w_2 process iterations $i = 1$ and $i = 2$. Because those nodes access consecutive data in array $x[i]$, $x[i-1]$, and $x[i-2]$, they actually access the same data: $x[0]$. Thus, the three memory operations are the same, and the three PEs can share a memory bus or one memory operation using half the resources. Note that the saved resources can be used to execute other iterations in parallel to increase performance.

Conditions for sharing

We distinguish memory operations that read the same address in iterations differing by a constant number because these are the only operations that are the same for every cycle. We call these memory operations, and the nodes containing them, *alignable* (hereafter, nodes mean those in the PE-level operation tree). Alignable nodes can contain other operations, including another memory operation, as the PE microarchitecture permits.

The first condition for memory operation sharing is that alignable nodes must be on the same line. Alignable operations are easily detected from their memory access indices. Consider two memory read accesses, $A[(a \times i) + s]$ and $A[(a \times i) + t]$, having the same base address (A) and the same stride (a) but different offsets (s and t). Also assume that loop iterator i is incremented by integer c at every cycle. Then the two memory read accesses are alignable if difference $s - t$ is divisible by $a \times c$, because they access the same address in iterations differing by $(s - t)/(a \times c)$, which we call the iteration difference of the two memory operations.

To make the alignable operations actually access the same address at every cycle, the nodes containing them must be at the correct stages of the pipeline. Therefore, the second condition for memory operation sharing is that the pipeline stage difference of two alignable nodes must equal the iteration difference of their memory operations. Sharing the memory operations requires no additional hardware, because the DRAA architecture already permits sharing through the bus interface—the memory-operation-sharing technique merely exploits a better placement, considering both the data reuse pattern and the pipeline execution.

Handling sharing conditions

We based our mapping flow on line placement rather than computationally expensive 2D placement. Thus, we must place alignable nodes carefully, respecting the memory-operation-sharing conditions without complicating the plane-level mapping process. We identify a subtree for each set of alignable nodes and place each subtree before the line-level mapping. As Figure 5 shows, the subtree has the first common ancestor of all alignable nodes (for each set) as the root, as well as all the nodes below the root. If this subtree can be successfully placed (generating a group of line placements) a supernode replaces the subtree, and the resulting tree goes to the line-level mapping process. Otherwise, the subtree remains in the tree. Finally, the plane-level mapping process stacks the line placements from both the subtrees and the modified tree.

Placement heuristic

Although a 2D placement is difficult for general subtrees satisfying the conditions for memory operation sharing, we have developed a placement heuristic for regular-structured subtrees found in many DSP and multimedia algorithms.[11] The heuristic restructures the subtree by inserting dummy nodes so that the pipeline stage differences of the alignable nodes equal their iteration differences. An efficient implementation first sorts the alignable nodes according to their array indices. For each pair of neighboring nodes, this implementation enforces the path lengths (from the nodes to the subtree root) to conform with the lengths dictated by the memory-operation-sharing conditions. The subtrees then map directly to the reconfigurable plane, with the alignable nodes on the same line.

Experimental results

For our experiments, we used the example architecture described in Table 1. For the line interconnections and nearest-neighbor connections, we used global buses, which can also serve as memory buses.

Table 2 describes the eight loops used in the experiments. Six loops are from the Livermore loops benchmark suite; the other two are from a wavelet filter and the motion estimation (ME) kernel of an MPEG encoder. We used only the loops exhibiting inter-iteration data reuse patterns. The degree of data reuse varies: Some (hydro, ICCG, and diff) have only two alignable memory operations in a loop, whereas others (banded and state) have three to five sets of alignable operations with two to three operations per set. The ME benchmark has eight sets of 16 alignable operations each.

For each benchmark, we selected an appropriate loop level. Because our current mapping flow can handle only one level of loop at a time, when the benchmark has nested loops, either the inner loops were unrolled or only the innermost loop was used. Then the loops were fed into the mapping flow, with and without the memory-operation-sharing optimization.

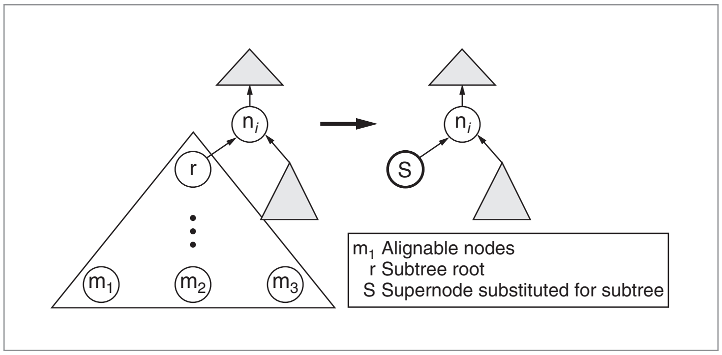


Figure 5. Identifying and replacing a subtree.

Table 3 compares the mapping results, showing

- the number of lines used for one instance of the loop pipeline,
- the number of configurations,
- the pipeline's latency,
- the throughput of the entire reconfigurable plane (the number of pipelines on the plane divided by the number of configurations), and
- the total number of cycles for all iterations, which depends on the loop's repetition count.

The rightmost columns show the reduction in the total number of cycles and the ratio of throughputs due to memory operation sharing.

Table 3 shows that the pipelines generated using memory operation sharing tend to have slightly higher latencies but take fewer lines and often increase the throughput considerably, up to three times in the ME example. Only the banded loop was mapped with fewer lines without memory operation sharing, but this bene-

Table 2. Loops used in the experiments.

Loop	Description	No. of memory operations	No. of repetitions
hydro	Hydrodynamic excerpt from Livermore loops (LL)	4	40
ICCG	Incomplete Cholesky-conjugate gradient from LL	6	40
banded	Banded linear equations (unrolled) from LL	14	3
state	Equations of state from LL	10	12
ADI	Alternating direction, implicit integration (innermost, part) from LL	11	7
diff	First difference from LL	3	98
wavelet	Wavelet filter implementation (innermost)	5	24
ME	Motion estimation kernel (unrolled) from MPEG encoder	128	30

Table 3. Comparison of mapping results.

	Without memory operation sharing					With memory operation sharing						
Loop	No. of lines	No. of configurations	Latency	Throughput	No. of cycles	No. of lines	No. of configurations	Latency	Throughput	No. of cycles	Latency reduction by memory operation sharing (%)	Throughput ratio (with/without memory operation sharing)
hydro	2	1	4	4	13	2	1	5	4	14	-7	1
ICCG	3	1	3	2	22	3	1	3	2	22	0	1
banded	7	2	11	0.5	15	8	1	11	1	13	13	2
state	5	1	9	1	20	4	1	11	2	16	20	2
ADI	6	1	7	1	13	5	1	7	1	13	0	1
diff	2	1	3	4	27	1	1	3	8	15	44	2
wavelet	3	1	4	2	15	2	1	4	4	9	40	2
ME	65	18	66	0.056	588*	23	6	36	0.167	210	63	3

* The actual number of cycles may be larger in this case, because this mapping uses more configurations than the configuration cache can hold.

fit was offset by the need for multiple configurations because the line placements couldn't be connected with a single configuration.

Not all the subtrees for which the heuristic found the placement contributed to improving the throughput. Some subtrees placed by the heuristic ended up using more lines than if we used the mapping flow algorithm, typically when the subtree's height was large and the number of alignable nodes it contained was small. Selectively applying the technique for each subtree could achieve a better result.

Table 3 also shows that memory operation sharing reduces the number of configurations, primarily by creating smaller input trees that the mapping flow can more easily handle.

Our experimental results demonstrate that our technique can generate performance improvement of up to three times for a typical coarse-grained reconfigurable architecture compared with the case when memory operation sharing is not used.

THE PROBLEM of mapping applications onto reconfigurable architectures still leaves many challenges open. For example, in our current work we assumed that loop iterations execute in the pipeline to develop a mapping flow that works reasonably for many applications. However, some loops might be better mapped when iterations execute in parallel. Therefore, the mapping style could be another dimension for optimizing the mapping of different applications. Furthermore, the current mapping flow has several constraints on architectures and application loops that must be relaxed. Our future research will investigate mapping techniques for more general classes of architectures as well as other types of loops. ■

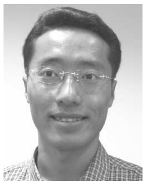
Acknowledgments

This research was conducted while authors Lee and Choi were visiting the University of California, Irvine, and was supported by grants from the National Science Foundation (CCR-0203813), Hitachi, and the Brain Korea-21 program. We also thank members of the UC Irvine Express compiler team for their assistance.

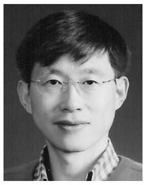
■ References

1. P. Schaumont et al., "A Quick Safari through the Re-Configurable Jungle," *Proc. 38th Design Automation Conf.* (DAC 01), ACM Press, New York, 2001, pp. 172-177.
2. R. Hartenstein, "A Decade of Reconfigurable Computing: A Visionary Retrospective," *Proc. Design, Automation and Test in Europe* (DATE 01), ACM Press, New York, 2001, pp. 642-649.
3. H. Singh et al., "MorphoSys: An Integrated Reconfigurable System for Data-Parallel and Computation-Intensive Applications," *IEEE Trans. Computers*, vol. 49, no. 5, May 2000, pp. 465-481.
4. S. Goldstein et al., "PipeRench: A Coprocessor for Streaming Multimedia Acceleration," *Proc. 26th Int'l Symp. Computer Architecture* (ISCA 99), IEEE CS Press, Los Alamitos, Calif., 1999, pp. 28-39.

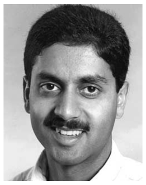
5. T. Miyamori and K. Olukotun, “REMARC: Reconfigurable Multimedia Array Coprocessor (Abstract),” *Proc. ACM/SIGDA Int’l Symp. Field Programmable Gate Arrays*, ACM Press, New York, 1998, p. 261.
6. E. Mirsky and A. DeHon, “MATRIX: A Reconfigurable Computing Architecture with Configurable Instruction Distribution and Deployable Resources,” *Proc. IEEE Symp. Field-Programmable Custom Computing Machines* (FCCM 96), IEEE CS Press, Los Alamitos, Calif., 1996, pp. 157-166.
7. K. Bondalapati, “Parallelizing DSP Nested Loops on Reconfigurable Architectures Using Data Context Switching,” *Proc. Design Automation Conf.* (DAC 01), ACM Press, New York, 2001, pp. 273-276.
8. Z. Huang and S. Malik, “Exploiting Operation Level Parallelism through Dynamically Reconfigurable Data Paths,” *Proc. Design Automation Conf.* (DAC 02), ACM Press, New York, 2002, pp. 337-342.
9. A. Halambi et al., “EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Retargetability,” *Proc. Design, Automation and Test in Europe* (DATE 99), IEEE CS Press, Los Alamitos, Calif., 1999, pp. 485-490.
10. K. Furuta et al., “Spatial-Temporal Mapping of Real Applications on a Dynamically Reconfigurable Logic Engine (DRLE) LSI,” *Proc. IEEE Custom Integrated Circuits Conf.*, IEEE Press, Piscataway, N.J., 2000, pp. 151-154.
11. J. Lee et al., “Mapping Loops on Coarse-Grain Reconfigurable Architectures Using Memory Operation Sharing,” tech. report 02-34, Center for Embedded Computer Systems (CECS), Univ. of California, Irvine, Calif., 2002.



Jong-eun Lee is a PhD candidate in electrical engineering and computer science at Seoul National University, Korea. His research interests include architecture, design, and compilation for configurable and dynamically reconfigurable processors. Lee has a BS and MS in electrical engineering from Seoul National University.



Kiyoung Choi is a professor of electrical engineering and computer science at Seoul National University, Korea. His research interests include embedded systems design, high-level synthesis, and low-power systems design. Choi has a PhD in electrical engineering from Stanford University.



Nikil D. Dutt is a professor of information and computer science and of electrical and computer engineering at the University of California, Irvine. His research interests include embedded computer systems design automation, computer architectures, and compiler optimization. Dutt has a PhD in computer science from the University of Illinois at Urbana-Champaign. He is a senior member of the IEEE.

■ Direct questions and comments about this article to Jong-eun Lee, School of Electrical Engineering and Computer Science, Seoul National University, Seoul 151-742, Korea; jelee@poppy.snu.ac.kr.

For further information on this or any other computing topic, visit our Digital Library at http://computer.org/publications/dlib.