Automatic Compilation to a Coarse-Grained Reconfigurable System-opn-Chip

GIRISH VENKATARAMANI and WALID NAJJAR
University of California, Riverside
FADI KURDAHI and NADER BAGHERZADEH
University of California, Irvine
WIM BOHM and JEFF HAMMES
Colorado State University

The rapid growth of device densities on silicon has made it feasible to deploy reconfigurable hardware as a highly parallel computing platform. However, one of the obstacles to the wider acceptance of this technology is its programmability. The application needs to be programmed in hardware description languages or an assembly equivalent, whereas most application programmers are used to the algorithmic programming paradigm. SA-C has been proposed as an expression-oriented language designed to implicitly express data parallel operations. The Morphosys project proposes an SoC architecture consisting of reconfigurable hardware that supports a data-parallel, SIMD computational model. This paper describes a compiler framework to analyze SA-C programs, perform optimizations, and automatically map the application onto the Morphosys architecture. The mapping process is static and it involves operation scheduling, processor allocation and binding, and register allocation in the context of the Morphosys architecture. The compiler also handles issues concerning data streaming and caching in order to minimize data transfer overhead. We have compiled some important image-processing kernels, and the generated schedules reflect an average speedup in execution times of up to 6× compared to the execution on 800 MHz Pentium III machines.

Categories and Subject Descriptors: D.1.2 [**Multiple Data Stream Architectures**]—*Array and vector processors*; C.1.4 [**Parallel Architectures**]; C.3 [**Computer systems Organization**]: Special-Purpose and Application-based Systems—*Real-time and embedded systems*; D.3.2 [**Language Classifications**]—*Data-flow languages*; D.3.4 [**Processors**]—*Code generation, compilers, optimizations*

General Terms: Design, Algorithms, Experimentation

Additional Key Words and Phrases: Reconfigurable computing, compilers, SIMD

This research was supported by DARPA under US Air Force Research Laboratory Contract F33615-98-1319 and by NSF under ITR Award 0083080.
Authors' addresses: G. Venkataramani and W. Najjar, Department of Computer Science and Engineering, University of California Riverside, Riverside, CA; 92521-0304, USA; email: najjar@cs.ucr.edu; F. Kurdahi and N. Bagherzadeh, Department of EECS, University of California Irvine, Irvine, CA; W. Bohm and J. Hammes, Department of Computer Science, Colorado State University, Fort Collins, CO.
Present addresses: J. Hammes, SRC Corp. Colorado Springs, CO; G. Venkataramani, Carnegie-Mellon University, Department of Computer Science.

1. INTRODUCTION

In the past decade, the performance and circuit density of silicon-based VLSI systems have experienced significant growth. Some of the CPU-intensive applications that were previously feasible only on supercomputers have now entered the realm of workstations and PCs. Similarly, the viability of reconfigurable hardware has also increased greatly in the past decade. Such systems rely on the dynamic mapping of a program segment directly onto the hardware in order to take advantage of inherent data parallelism in the program. We focus on accelerating applications in the image processing domain. These applications can greatly benefit from the advantages of reconfigurable computing systems since they are characterized by inherent data parallelism and regular data structures with regular access patterns.

A common reconfigurable platform deployed widely today is the field programmable gate array (FPGA). FPGAs have the potential for a very large degree of parallelism as compared to traditional processors. Consequently, the performance achieved by FPGA-based reconfigurable architectures can potentially be several orders of magnitude greater than that of processor-based alternatives for some applications. However, fine-grained (or LUT-based) reconfigurable platforms such as FPGAs have a number of inherent disadvantages:

- *Difficulty of Programmability*: In most cases, reconfigurable computing systems still require the manual translation of a program into a circuit using a hardware description language (HDL). This process is a significant hindrance to the wider acceptance of this technology by application program developers, since most of them are used to expressing the application at the algorithmic level in some high-level programming language.
- *Logic Granularity*: FPGAs are designed for logic replacement. Consequently, for applications where the data path is coarse grained (8 bits or more), the performance on FPGAs is inefficient.
- *Compilation and Reconfiguration Time*: Applications meant to execute on FPGAs are typically written in a hardware description language such as VHDL or Verilog. Mapping such code onto FPGAs has to go through a number of compilation passes that include logic synthesis, technology mapping, and placing and routing on the target FPGA. This process takes from a few hours to days for some applications.

As a result, many coarse-grained (or ALU-based) reconfigurable systems [Goldstein et al. 2000; Singh et al. 2000; Waingold et al. 1997] have been proposed as an alternative between FPGA-based systems and fixed logic CPUs. The reconfigurable computing element in such systems is typically a custom computing hardware that is (usually) deployed as a co-processor.

The research work in this paper addresses the problem of compiling a program written in a high-level language, SA-C, to a coarse-grained reconfigurable architecture, Morphosys. The Morphosys architecture consists of a general-purpose processor core and an array of ALU-based reconfigurable processing elements. The remainder of this section presents some background for this work, including a brief overview of reconfigurable computing architectures, the

SA-C language, the Morphosys architecture, and some previous relevant research. Section 2 presents a brief description of the approach to compilation adopted in this work. Section 3 describes the entire mapping process in detail. Section 4 presents the performance evaluation of the compiler-generated code. Finally, the last section presents the conclusions and possible future directions.

1.1 Reconfigurable Computing Systems

The main idea behind reconfigurable computing is to avoid the *von Neumann bottleneck* (the limited bandwidth between processor and memory) by mapping computation directly into hardware. Such a system also has the ability to dynamically change the hardware logic that it implements. Hence, an application can be temporally partitioned for execution on the hardware. After one partition completes its execution the hardware is reconfigured to execute the next partition. Thus, system designers can execute more hardware than they have gates to fit. Reconfigurable computing systems represent an intermediate approach between application specific integrated circuits (ASICs) and general-purpose processors.

The most common way to deploy reconfigurable computing systems is to combine a reconfigurable hardware-processing unit with a software programmable processor. Reconfigurable processors have been widely associated with field programmable gate array (FPGA)-based system designs. An FPGA consists of a matrix of programmable logic cells with a grid of interconnect lines running between them. In addition, there are I/O pins on the perimeter that provide an interface between the FPGA and the interconnect lines and the chip's external pins.

However, reconfigurable hardware is not limited to FPGAs. Several projects have investigated and successfully built systems where the reconfiguration is coarse grained and is performed within a processor or amongst processors. In such cases the reconfigurable unit is a specialized hardware architecture that supports dynamic logic reconfiguration.

1.2 The SA-C Language

SA-C [Hammes and Böhm 2001; Hammes et al. 1999] is a single assignment language with functional features. It was originally designed for automatic compilation to fine-grained reconfigurable platforms such as FPGAs. The language borrows much of C's syntax and expression-level semantics, so that it is accessible to the common C programmer. It has been designed with two primary goals in mind—one, easy expression of image processing applications with a high degree of abstraction; and two, efficient compilation to hardware—the language constructs are such that they expose the inherent parallelism in the program. The main features of SA-C can be summarized as follows:

- It is an expression-oriented language. Hence, every construct in the language will have to return a value. Alternatively, every statement in the language is an assignment statement.

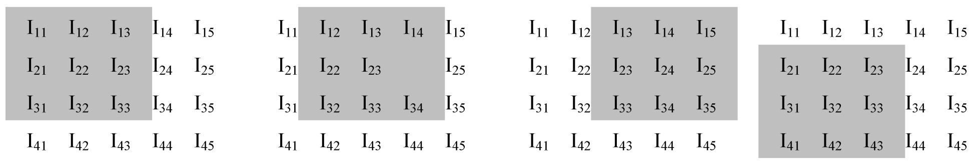
- Its data types support variable bit-width precision for integer and fixed-point numbers.
- The arrays in SA-C are true multidimensional arrays. Hence, any plane, slice, row, column, window, or element of the array can be accessed directly.
- The language is based on single assignment. This means that variables cannot be reassigned. This feature makes it possible for the compiler to perform extensive optimizations when mapping algorithms to hardware.
- There are no pointers in SA-C. However, since the language supports such flexible array accessing patterns, other features of the language have satisfied the usefulness of pointers. A number of standard image processing applications, libraries, and benchmarks have been written in SA-C in spite of this restriction.
- SA-C does not support recursion. This feature ensures that the algorithms can be easily mapped to hardware. Moreover, all tail recursions can be converted to iterative constructs.
- Multiple-value returns and assignments. In addition to being expression oriented, the language allows multiple values to be returned from an expression/statement.
- *Image Processing Reduction Operators*: the language supports a number of useful image processing reduction operators (such as histogram, median, and so on) that can be applied to loops and arrays.
- Loops in SA-C are the most commonly used constructs. They are very particular in that they tend to inherently specify how the loop may traverse a particular array and what kind of parallel operations it performs. The next section describes SA-C loops in a little more detail.

A compiler for SA-C has been developed that maps applications to multi-FPGA-based systems. The SA-C compiler supports a wide range of optimizations aimed at producing an efficient hardware execution model. This is achieved by re-using previous computations, eliminating unnecessary computations, reducing the storage area required on the FPGA, reducing the number of reconfigurations, exploiting the locality of data and therefore reducing the required data bandwidth from the host to the FPGA, and finally improving the clock rate of the circuit. These include traditional optimizations such as constant folding, operator strength reduction, dead code elimination, invariant code motion, and common subexpression elimination. Other optimizations have been either developed or adapted from vectorizing and parallelizing compilers as well as synthesis tools. These include bit-width narrowing, loop unrolling, stripmining, and loop fusion.

Perhaps the most important language construct in SA-C is loops. Every loop in SA-C has three components to it: the loop generator, the loop body, and the loop collector. A loop generator specifies what values are generated in each iteration, and how many iterations the loop will perform. The loop collector generates a return value for the loop expression, by combining, in various ways, values that are produced within the loop. This makes SA-C loops particularly

```
for window w[3,3] in Image
```

(a)


I_{11} I_{12} I_{13} I_{14} I_{15}
I_{21} I_{22} I_{23} I_{24} I_{25}
I_{31} I_{32} I_{33} I_{34} I_{35}
I_{41} I_{42} I_{43} I_{44} I_{45}

(b)

Fig. 1. (a) Syntax of a SA-C loop generating 3 × 3 windows from the source array, *Image*. (b) The shaded portion of *Image* represents the value of w in various iterations.

conducive to data-streaming kernels, such as compression/decompression, which tend to have the same profile.

There are two main types of loop generators: *array-element* and w*indow* generators. An *element generator* produces a single scalar value from the source array per iteration. A *window generator* produces a subarray (of specified size) of the same dimensions as the source image per iteration. Figure 1 shows how a window generator in SA-C works. The example shows windows of size 3 × 3 being generated from a source array (*Image*) of size 4 × 5. Shaded portions of Figure 1(b) represent the different windows generated in different iterations (the figure does not show all the iteration windows).

Every loop in SA-C must return one or more values that are described by the loop's collectors. There are two kinds of collectors: *ConstructArrayCollector* and *ReductionCollector*. A *ConstructArrayCollector* returns an array whose elements correspond to results from each iteration of the loop execution. A *ReductionCollector* applies some sort of an arithmetic reduction operation (such as sum, product and so on) on the range of values produced from each iteration, and returns the reduced value.

1.3 Morphosys Architecture

Morphosys [Lee et al. 2000; Singh et al. 2000] is a model for reconfigurable computing systems that is targeted at applications with inherent data parallelism, high regularity, and high throughput information. Most applications that fit this profile fall under the domain of image processing. The Morphosys architecture consists of five main components: the tiny RISC processor core, the reconfigurable cell array (RC array), the context memory, the frame buffer, and the DMA (direct memory access) controller. In this section, we briefly describe these components. For more details, please refer to Singh et al. [2000].

1.3.1 *Tiny RISC.* Tiny RISC is a MIPS-like processor with a 4-stage pipeline. It has 16 × 32-bit registers and three functional units: a 32-bit ALU, a 32-bit shift unit, and a memory unit. An on-chip data cache memory reduces accesses to external memory. The tiny RISC processor handles general-purpose

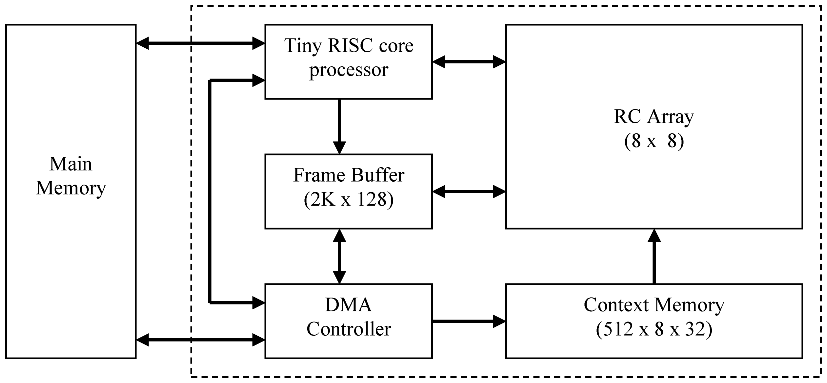


Fig. 2. The Morphosys architecture.

operations and controls the execution of the RC array through special instructions added to its ISA [Chaves 1998]. Through DMA instructions, it also initiates all data transfers to and from the frame buffer, and the loading of configuration programs into the context memory. RC array instructions specify one of the internally stored configuration programs and how it is broadcast to the RC array. The tiny RISC processor is not intended to be used as a stand-alone, general-purpose processor. Although tiny RISC performs sequential tasks of the application, performance is mainly determined by data-parallel processing in the RC array.

1.3.2 *RC Array.* The RC array consists of an 8 × 8 matrix of processing elements called the reconfigurable cells. Each RC cell consists of an ALU-multiplier, a shift unit, input multiplexers, and the context register. In addition to standard arithmetic and logical operations, the ALU-multiplier can perform a multiply-accumulate operation in a single cycle. The input multiplexers select from one of several inputs for the ALU-multiplier:

1. One of the four nearest neighbors in the RC array,
2. Other RCs in the same row/column within the same RC array quadrant,
3. The operand data bus, or
4. The internal register file.

The context register provides control signals for the RC components through the context word. The bits of the context word directly control the input multiplexers, the ALU/multiplier, and the shift unit. The context word determines the destination of a result, which can be a register in the register file and/or the express lane buses. The context word also has a field for an immediate operand value.

1.3.3 *Context Memory.* The context memory stores the configuration program (the contexts) for the RC array. It is logically organized into two partitions, called context block 0 and context block 1. Each context block is logically subdivided into eight further partitions, called context Sets. context words are broadcast to the RC array on a row/column basis. Context words from context block 0 are broadcast along the rows, while context words from context block 1 are broadcast along the columns. Within context block 0 (1), context set n is associated with row (column) n, $0 <= n <= 7$, of the RC array. Context words from a context set are sent to all RCs in the corresponding row (column). All RCs in a row (column) receive the same context word and therefore perform the same operation. It is also possible to selectively enable a single context set (and therefore, row/column) to be active in any given clock cycle. This demonstrates the SIMD/MIMD hybrid model of the RC array. It supports SIMD-style execution within each row/column, while different rows (columns) can execute different operations.

1.3.4 *Frame Buffer.* This is a streaming buffer and is part of a high-speed memory interface. It has two sets and two banks. It enables streamlined data transfers between the RC array and main memory, by overlapping computation with data load and store, alternately using the two sets.

The frame buffer is an internal data memory logically organized into two sets, called set 0 and set 1. Each set is further subdivided into two banks, bank A and bank B. A 128-bit operand bus carries data operands from the frame buffer to the RC array. This bus is connected to the RC array columns, allowing eight 16-bit operands to be loaded into the eight cells of an RC array row/column (i.e., one operand for each cell) in a single cycle. Therefore, the whole RC array can be loaded in eight cycles.

The operand bus has a single configuration mode, called interleaved mode. In this mode the operand bus carries data from the frame buffer banks in the order A0, B0, A1, B1, ..., A7, B7, where An and Bn denote the nth byte from bank A and bank B, respectively. Each cell in an RC array column receives 2 bytes of data, one from bank A and the other from bank B. Results from the RC array are written back to the frame buffer through a special "*result bus*."

1.3.5 *DMA Controller.* The DMA controller performs data transfers between the frame buffer and the main memory. It is also responsible for loading contexts into the context memory. The tiny RISC core processor uses DMA instructions to specify the necessary data/context transfer parameters for the DMA controller.

1.4 Related Research

Computing systems with reconfigurable architectures can be classified by the kind of reconfigurable computing fabric that they use. LUT-based reconfigurable fabrics, such as field programmable gate arrays (FPGAs), have been widely used in many research efforts. There has also been significant work in the area of non-FPGA-based reconfigurable computing architectures in which the reconfigurable computing element is a custom-computing hardware.

1.4.1 *Compiling to FPGA-Based Systems.* Many research efforts have focused on trying to automatically map algorithms that are written in a high-level language to FPGAs. Some of this work has focused on defining a new language that can be more easily mapped to FPGAs, while still maintaining the level of abstraction that is important to algorithm writers. In most of the work in this field, the main focus of the compiler is to partition the high-level algorithms first temporally, and then spatially, in order to fit them on the FPGAs for execution. A brief review of this work follows.

In Peterson et al. [1996], the approach is to generalize the reconfigurable elements of the architecture. The base platform is a multi-FPGA system (Annapolis MicroSystems WildForce platform). The resources available in this base architecture can be parameterized through an architectural description file. The system consists of a host processor and the reconfigurable system as specified above. The source language is graph description language (GDL) that is closer to the data flow graph representation of the program and is intended to be an intermediate form as opposed to a human interface. Given the source (GDL) program and the resource constraints as input, the compiler temporally partitions the program to satisfy resource constraints. Spatial partitioning will then map the partitions produced by temporal partitioning onto the multiple FPGAs. Simulated annealing algorithms are used to select the most desirable partitions.

Streams-C [Frigo et al. 2002] is a restricted version of C. The framework proposes a compilation system, based on the SUIF compiler infrastructure, for automated mapping of algorithms onto FPGAs. There is particular emphasis on extensions that facilitate the expression of communication between parallel processes.

In Hall et al. [1999], the approach is to leverage parallelizing compiler technology based on the Stanford SUIF compiler. The architecture is made of a general-purpose processor (GPP) core and several configurable computing units (CCUs). The source program can be either C or MATLAB. The compiler identifies those portions of the program that can be executed on the CCUs and partitions the program accordingly based on resource and timing requirements. The code that can be executed on the CCUs is usually identified as parallelizing loops and vector-style SIMD computations. Each partition is then scheduled to execute on the CCUs and control for the partition is usually a finite state machine (FSM) that executes on the GPP.

There have been many research efforts aimed at mapping algorithms to FPGAs. These efforts focus on a specific problem in the mapping process and propose heuristics and algorithms that attempt to optimally solve these problems.

The NIMBLE [Li et al. 1999] compiler is a framework for compiling C code to VHDL targeted at FPGAs. The work addresses the problem of temporal partitioning of applications intended to run on FPGAs as a hardware–software partitioning problem. The NIMBLE compiler is designed to preprocess the application to extract candidate loops (kernels) that can be scheduled to execute on the FPGAs. In addition to preprocessing, profiling the kernels is necessary to determine the optimizations that are best suited for the targeted hardware. The work proposes a heuristic algorithm to select from the candidate kernels those that will execute on the FPGAs and those that will execute on the general-purpose host CPU, such that the execution time for the whole application is minimized.

In Bondalapati and Prasanna [2000], the work addresses the problem of mapping loop constructs to a generic reconfigurable architecture. In particular, the approach aims at minimizing reconfiguration overhead by optimally scheduling the reconfigurations. The loop is represented as a kernel with a set of operations each of which is associated with a configuration cost. The work is aimed at coming up with an optimal solution that searches the solution space in polynomial time using dynamic programming.

On a related front, Kaul et al. [1999] proposes a model to perform near-optimal temporal partitioning of a given application that is intended to execute on a multi-FPGA system such as the AMS WildForce board. The application is specified as a task graph, which is really a dataflow graph whose nodes represent tasks/operations with given execution latencies. The target reconfigurable hardware is parameterized in terms of resource constraints and reconfiguration costs. Integer linear programming model is used to find near-optimal (in terms of execution time) temporal partitions of the application that can be mapped to the target hardware.

1.4.2 *Compiling to Non-FPGA-Based Systems.* Most non-FPGA systems are based on a special configurable computing hardware component that is generally attached as a co-processor to a general-purpose core processor. Compiling to this kind of system is a not as generic a problem as compiling to FPGA-based systems because FPGAs have almost become standard hardware elements. In non-FPGA-based systems, the reconfigurable computing element would pose special problems that are specific to the concerned hardware. Hence, the compilation approach needs to be closely related to the kind of reconfigurable architecture.

The Garp [Wawrzynek and Calahan 1998] architecture consists of a general-purpose architecture and a reconfigurable array of computing elements (configurable logic blocks or CLBs), and is designed to function as a general-purpose architecture. This approach draws heavily from compiling techniques for VLIW processors. The compiler aims at exploiting fine-grained parallelism in applications by scheduling frequently executed instruction sequences (the trace-scheduling technique from VLIW compilers) for execution on the array. The source program is converted to an equivalent data flow graph, which is then partitioned into modules and hyperblocks, which are a group of basic blocks that expose ILP. This data flow graph is further optimized and is implemented as a fully spatial network of modules in the array; hence, every operation gets its own hardware. Further compiler analysis is performed that can then pipeline loops and add pipeline registers where necessary.

CHIMAERA [Ye et al. 2000] is a RISC processor with a reconfigurable functional unit (RFU). The compiler recognizes frequently executed sequences of instructions that can be performed on the RFU, and creates new operations (RFUOPs) based on them. To do this, three important compiler optimizations are performed: control localization (to remove branches), SIMD within a register (to maximize parallelism by identifying loop bodies and optimizing the data access within the loop), and finally frequently executed basic blocks are transformed into an RFUOP.

PipeRench [Goldstein et al. 2000] is an interconnection network of configurable logic and storage elements. The PipeRench compiler introduces the idea of pipelined reconfiguration in which the application's virtual pipe stages are first analyzed and then optimally mapped to the architecture's physical pipe stages to maximize execution throughput. The source language, dataflow intermediate language (DIL), is characterized by the single-assignment paradigm and configurable bit-widths. The compiler flattens the application's dataflow graph and then uses a greedy place-and-route algorithm (that runs in polynomial time) to map the application onto the reconfigurable fabric.

The RAW microarchitecture [Waingold et al. 1997] is a set of interconnected tiles, each of which contains its own program and data memories, ALUs, registers, configurable logic, and a programmable switch that can support both static and dynamic routing. The tiles are connected with programmable, tightly integrated interconnects. The proposed compiler is meant to partition program execution into multiple, coarse grained parallel regions. Each parallel region may execute on a collection of tiles. The size and the number of these regions are determined by compiler analyses that take into account the resource restrictions. Then, static schedules are generated for each such execution thread. These schedules are designed to exploit fine-grained parallelism and minimize communication latencies. The compiler is implemented using the Stanford SUIF compiler infrastructure.

The RaPiD architecture [Ebeling et al. 1996] is a field-programmable architecture that allows pipelined computational structures to be created from a linear array of ALUs, registers, and memories. These are interconnected and controlled using a combination of static and dynamic control. RaPiD-C is proposed as a programming language to specify the application that is to be executed on the RaPiD architecture. The language, however, requires the programmer to explicitly specify the parallelism, data movement and partitioning. Hence, partitioning is inherent in the language itself—outer loops specify time and inner loops specify space. It turns out that an application written in RaPiD-C is very close to a structural, hardware description of the algorithm. Hence, compiling a RaPiD-C program essentially involves mapping this RaPiD-C description onto a complete structural description consisting entirely of components in the target architecture.

1.4.3 *Other Work.* Kennedy and Allen [1987] have focused on automatic translation of Fortran programs to Fortran 8× programs that are meant to be executed on vector computers such as Cray-1. Fortran 8× allows the programmer to explicitly specify vector and array operations. Although their work is similar to our work with respect to exploiting implicit SIMD parallelism, the architecture of vector computers is very different from the Morphosys architecture. In particular, the reconfigurable element of Morphosys is an array (RC array) of processors. Each processor of the array has its own register file, which can be accessed by other processors via an interconnection network. Hence, issues in instruction scheduling and register allocation are more complex.

1.4.4 *Our Work.* Our compiler is different from these in that it focuses on automatically mapping an application written in a high-level language to a coarse grained, ALU-based reconfigurable architecture that supports a hybrid SIMD/MIMD computational model. It is designed to accelerate streaming multimedia applications on this target architecture. In such applications, data streams through some kernel, which performs certain transformations, and the results are streamed out. The kernel transformations are typically applied uniformly on each element of the input data window. Given the SIMD/MIMD model that Morphosys offers, our compiler attempts to schedule the kernel such that it can benefit from this architecture. The schedule is complete in the sense that it specifies the order of operations (or context execution), and it dynamically configures the datapath while being constrained by resource availability. In addition, the compiler overlaps data streaming with kernel computation, in order to achieve more parallelism. It is a fully automatic, end-to-end compiler that has been built completely from scratch.

We have targeted the Morphosys architecture because we feel that the constructs of the SA-C language correlate with the execution model of Morphosys. Presently, the Morphosys architecture can only be hand programmed using Morphosys assembly codes. Writing the assembly for large applications is tedious, and this inhibits the wider acceptance of the Morphosys architecture. Our compiler attempts to bridge this gap by making the Morphosys architecture available to the application programmer.

2. COMPILER FRAMEWORK OVERVIEW

The main focus of this work is to build a compiler framework to translate SA-C programs into an execution schedule that can be mapped to the Morphosys reconfigurable architecture. A typical image-processing application consists of a number of kernels, or loops. A kernel is, typically, a set of computationally intensive operations performed in a loop. We focus on synthesizing each such kernel for execution on the reconfigurable element. The result of the synthesis is an execution schedule for the internal operations within, and the configuration memory requirements for the kernel. This will provide the execution characteristics (execution latency and configuration memory requirements) for each kernel in the program.

The mapping process itself is similar in nature to the architectural synthesis of algorithms. Each loop is analyzed as a set of operations that require a certain number of resources for execution. Algorithms that perform operation scheduling, processor binding, and register allocation in the context of the Morphosys computational model are applied to produce a complete execution schedule. Data transfer and caching can make a significant difference in the overall execution of the program. The compiler uses a simple strategy to prefetch data, so as to overlap most data fetching and storing with computation.

The compiler is evaluated by comparing the execution times of some common image-processing kernels on Morphosys to execution on an 800 MHz Pentium III. An average speedup of $6\times$ is observed among the benchmarks used. The compiler presented here aims at maximizing the benefits of the

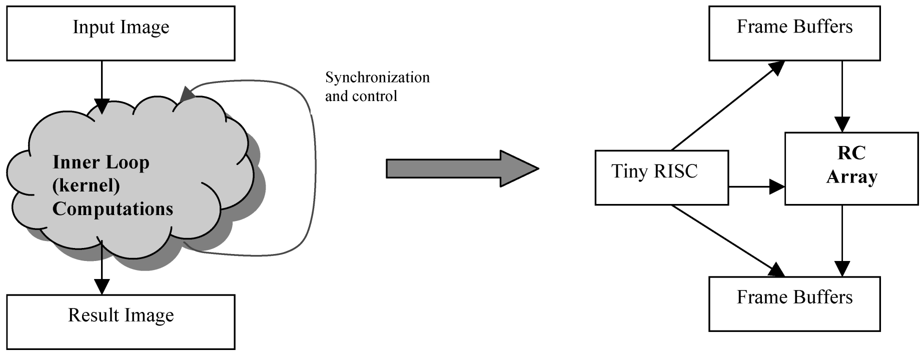


Fig. 3. Mapping kernels to Morphosys.

computation model presented by the Morphosys architecture, under the restrictions and resource constraints presented by the architecture and the language.

This work concentrates on producing an instruction schedule that exploits the SIMD computational model of Morphosys, and identifies and exploits parallelism at a fine- and coarse grained level. The focus of the compiler is to build a framework to map a *single kernel* onto the reconfigurable hardware for efficient execution. This objective is orthogonal to those addressed in Li et al. [1999], where the focus is on optimal *inter* kernel scheduling.

The compiler takes a SA-C source program as input and generates two files as output—the tiny RISC assembly code and the configuration contexts, which represent the list of operations that will be performed by the RC array during the execution of the program. These contexts are stored in the context memory of the Morphosys architecture. The tiny RISC instruction set architecture (ISA) contains instructions to direct the rows/columns of the RC array to perform the operation represented by a particular context. The tiny RISC assembly code that is generated will contain such instructions that will control the execution flow of the RC array based on the contexts that are generated. Hence, the tiny RISC assembly code represents the control code that drives the execution of the whole program, while the configuration contexts are just a list of the operations that need to be performed by the RC array to execute the given program. This section briefly describes the entire compilation process.

2.1 Flow of Compilation

Code partitioning determines which segments of the program will execute on the RC array and which will execute on the tiny RISC processor. The focus of this work is to completely map a given kernel (Figure 3) for execution on the RC array. All sequential code (outside loops) and code for synchronization and control are mapped for execution on the tiny RISC.

Figure 4 shows the flow of compilation. The right-side branch of compilation after code partitioning represents the compilation of code that is not within loops. This phase of code generation is, essentially, similar to that of traditional compilers. The left-hand branch represents the heart of this compiler. The process of generating the detailed execution schedule is referred to as "loop

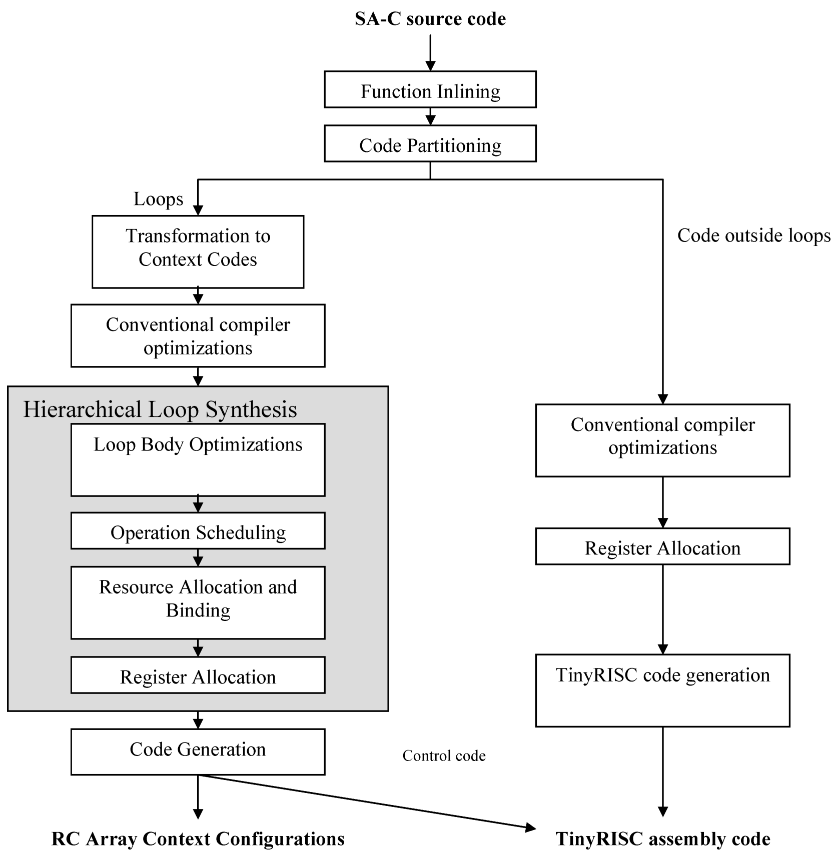


Fig. 4. Flow of compilation.

synthesis" throughout this document and is described in detail in the next section. The compiler first performs a number of tasks that prepare the program graph for loop synthesis.

Function Inlining. Since SA-C does not support pointers or recursion, every function in SA-C can be inlined. Inlining a function ensures that the context within which a function is called is exposed. This makes a difference during code partitioning, as a particular function can be called either from within a loop or from outside a loop, and this will determine whether the particular function will be mapped to the RC array or to the tiny RISC processor.

Transformation to Context Codes. The loops in the SA-C program are mapped onto the RC array for execution. Hence, as a requirement, every simple node within a loop must have a one-to-one correspondence to an RC array context code. Most of the operations within a loop will usually correspond directly to an RC array context code. However, at times, the operation is implicit and may be associated with a group of graph nodes. During the optimization phase, the compiler essentially performs a pattern-matching pass to find such candidate groups of nodes that can represent a single RC array context code, and transforms such nodes.

On the other hand, there may be certain nodes that do not directly correspond to any of the RC array context codes. For examples, there are no context codes that correspond to MAX, MIN, or SQRT. These operations can, however, be represented as a sequence of context codes that have the same effect. In such cases, the operation execution latencies of these nodes are updated to reflect the time required to execute this sequence of contexts. Ordinarily, for all other operations that directly correspond to an RC array context code, the execution latency of the operation is 1 clock cycle.

An interesting operation to implement is SQRT. The compiler uses the Friden algorithm [Crenshaw 2000] to implement square root functionality. It assumes that all numbers in the applications that execute on Morphosys are 8-bit numbers. Given this, the Friden algorithm computes in constant time. This algorithm has been converted to a sequence of RC array context codes to compute the square root of any 8-bit number. The execution latency of the algorithm is 50 clock cycles: this may seem large, but it can easily be amortized over multiple computations (remember that the RC array functions as per the SIMD computation model).

Conventional Compiler Optimizations. Apart from the optimizations mentioned above, the compiler also performs certain conventional optimizations. Note that these optimizations are again directed at the code within loops.

- Conversion of multiplications and divisions to shifts
- Common subexpression elimination
- Constant folding and constant propagation
- Dead code elimination

2.2 Hierarchical Data Flow Graph

The hierarchical data flow graphs (HDFG) are used as intermediate representations in the compiler. These graph representations are similar in structure to the data dependence control flow (DDCF) graphs [Hammes et al. 2001] with certain differences that reflect the nature of the Morphosys architecture. It is a convenient representation for performing compiler optimizations and for analysis in the mapping process.

An HDFG is an acyclic, directed, data flow graph, where some nodes can have subgraphs within them. This hierarchical property preserves the program semantics. Figure 5(a) shows an example of a SA-C program that computes the following function:

$$R[x][y] = \sum_{a=x}^{x+2} \sum_{b=y}^{y+2} A[a][b].$$

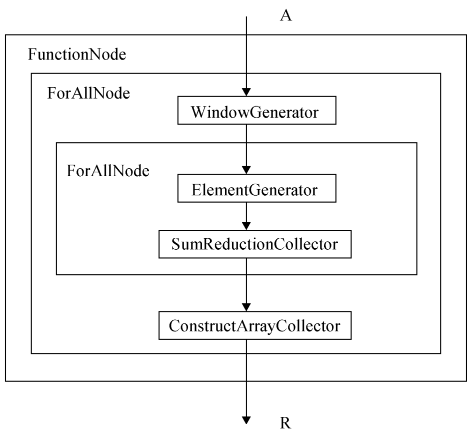
Figure 5(b) shows the equivalent C program, and Figure 5(c) shows the equivalent HDFG representation. The SA-C program has two loops. The outer loop contains a window generator that produces 3×3 windows from the source

```
Int8[8,8] f(int8[8,8] A) {
  Int8[8,8] R =
    For window w[3,3] in A {
      Int8 x = For e in w
      return (sum(e));
    } return (array(x));
} return R;
```

(a)

```
 For (I=0; I<M; I++) {
  For (J=0; J<N; J++) {
   For (X=I; X<(I+3); X++) {
    For (Y=J; Y<(J+3); Y++) {
     R[I][J] += A[X][Y];
    }
   }
  }
 }
```

(b)



(c)

Fig. 5. SA-C loop example that performs the following function: R[x][y] = $\sum_{a=xb}^{x+2} \sum_{b=y}^{y+2} A[a][b]$. (a) The SA-C source code, (b) Equivalent C code and (c) equivalent HDFG representation.

image, *A*. The inner loop contains an element generator, which produces scalar values from the generated window. Its loop collector is a *ReductionCollector* that performs a summation. Essentially, the inner loop computes the sum total of each window generated. The outer loop creates an array whose elements are the summation values produced from the inner loop. Hence, the outer loop contains a *ConstructArrayCollector*.

3. HIERARCHICAL LOOP SYNTHESIS

The objective of this phase is to analyze each loop, perform optimizations, and generate a complete, efficient execution schedule that specifies the temporal ordering of each operation, where on the RC array each of the operations will execute, and which results are written to which registers within the RC array. The RC array provides extensive support for the SIMD computation model. Hence, the goal of the compiler is to exploit the benefits of the RC array by scheduling the loop code for execution on the Morphosys architecture, while adhering to its computational model.

In general, the RC array can be visualized as an array of programmable computing elements. Hence, in the trivial case, a data flow graph can be mapped to the RC array using traditional architectural synthesis techniques. However, the number of computing elements is limited and the computation model is not as flexible as a general programmable logic array. Also, it is important to be aware of the structure and the restrictions of the source language in which the application is expressed—this will really govern the kind of programs the compiler has to deal with. Hence, a framework is designed, based on the characteristics of the language as well as of the RC array, to automatically map the loops for execution on the RC array.

All code in the SA-C program is statically scheduled for execution by the compiler. The compiler adopts a hierarchical approach to solve the problem of mapping SA-C loops. Loops are synthesized based on their relative position in the *loop hierarchy*, with the innermost loop defined to be at the bottom of the loop hierarchy. The compiler's approach is to synthesize the inner most loop, and then progressively move up the loop hierarchy until the outermost loop is synthesized. The compiler framework defines different execution models based on the loop's generator. This section examines different loop generators and describes the strategy used in synthesizing their loops.

In general, the problem of mapping loops to the RC array is treated as a generic architectural synthesis problem, which is concerned with mapping algorithms to hardware. Hence, given the SA-C program, the compiler determines which operations will be executed on which resources and in which execution cycle. In this context, a *resource* is defined to be one row (or column) of the RC array. This is due to the SIMD nature of the RC array—in a particular clock cycle, all the cells in a particular row (or column) shall perform the same operation. Hence, there is a total of eight resources (eight rows) that is available in any given clock cycle. The objective of the compiler scheduling is to maximize the resource usage in every clock cycle. Hence, loops are always unrolled when executing on the RC array so that multiple iterations are executing in any given clock cycles.

Each node (or operation) within the loop is marked with its execution latency times and resource requirements. For the loop body of an innermost loop, these numbers are pre-defined. Once an inner loop is synthesized, these numbers for the loop itself can be recursively inferred.

3.1 Loops with Element Generators

Loops with element generators are generally the innermost loop of any loop hierarchy. Its loop body is a function of a particular element of the source array. Currently, it is assumed that there are no data dependencies between iterations.[1] The loop is unrolled in both the horizontal and vertical direction so as to process 64 loop iterations in a single RC array iteration. Execution of every loop iteration is performed on a single RC array cell. Hence, the resource-binding problem is trivial and is obviated.

The operation-scheduling problem reduces to scheduling a data flow graph onto a single, sequential processor. There is only one constraint that needs to be considered in scheduling these loops. Certain operations within the loop may be data fetch operations, whose source data inputs reside in the frame buffer. As per the Morphosys architecture, only one row (or column) can perform a data fetch operation in a given cycle. Alternately, only eight elements can be fetched from the frame buffer in any given cycle. To accommodate this constraint, such operations are identified, and their operation latency numbers are tweaked to eight times their actual latencies. Finally, the compiler uses the ASAP ("as

[1]When there are data dependencies between iterations, one could rather use other SA-C loops (such as window-generating loops), or loop operations (such as summation). Hence, this is not a serious limitation.

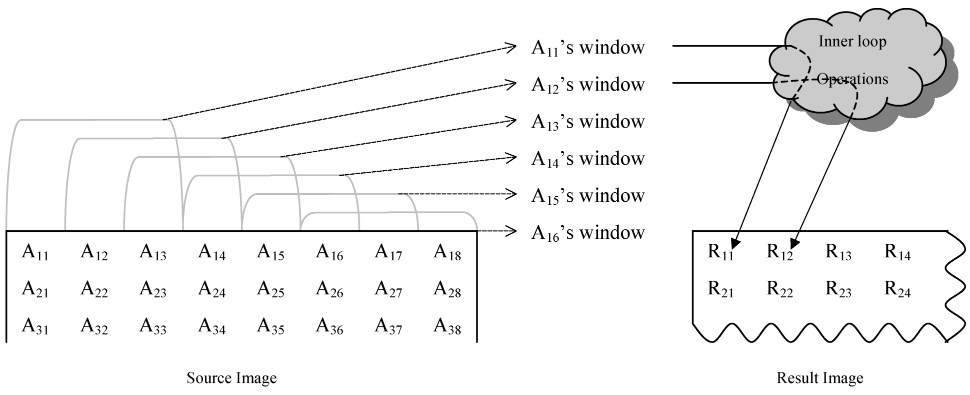


Fig. 6. Snapshot of windowing loop.

soon as possible") scheduling algorithm to schedule operations. This algorithm schedules operations as soon as their source data inputs are available. The details are described in Venkataramani [2001].

A greedy register allocation strategy is used that keeps track of the free registers, and allocates registers to intermediate results of operations, as and when required. Register spills are handled by writing the values to the frame buffer.

3.2 Loops with Window Generators

Perhaps the most important part of the SA-C language is its *window generators*, which are useful in expressing a number of common image processing applications in an elegant way. This kind of a loop allows a window to "slide" over the source array producing subarrays of the same rank (dimensions) as the source array.

Figure 6 shows a snapshot of a windowing loop from the example in Figure 5. The loop generates a 3×3 window in each iteration of the loop. Hence, every 3×3 window in the loop is the input data for a separate iteration. The inner loop body transforms this window into a single pixel (corresponding to the sum total of elements in the iteration's window) in the resultant image.

In spite of the SIMD computational model of the RC array, all the iteration windows present in the RC array cannot be computed concurrently. This is because some of the elements are part of multiple iteration windows. For example, element A_{13} is a member of three iteration windows—A_{11}, A_{12}, and A_{13}. However, execution of windows A_{11} and A_{14} can be performed concurrently. Similarly, windows A_{12} and A_{15} can also be executed concurrently.

Hence, nonoverlapping windows can be computed in parallel. Specifically, all windows that are separated by whole multiples of the window size are computed concurrently. Hence, iteration windows corresponding to elements A_{11}, A_{14}, A_{41}, and A_{44} are computed concurrently. Then, iteration windows corresponding to elements A_{12}, A_{15}, A_{42}, and A_{45} are computed concurrently, and so on. There are a total of 36 iteration windows in the RC array, and sets of four iterations can be executed concurrently. After all the 36 iterations are completed, the next chunk of 64 elements is brought into the RC array.

```
Int8[:,:] R =
  For window wa[3,3] in A
      dot window wb[3,3] in B {
    Int8 asum =
      For a in wa dot b in wb
      return (sum(a * b));
  } return (array(asum));
```

(a)

```
For (I=0; I<M; I++) {
 For (J=0; J<N; J++) {
  R[I][J] = 0;
  For (X=I; J<(I+3); X++) {
   For (Y=J; Y<(J+3); Y++) {
    R[I][J] += A[X][Y] * B[X][Y];
   }
  }
 }
}
```

(b)

Fig. 7. Windowing loop example: (a) SA-C code and (b) C code.

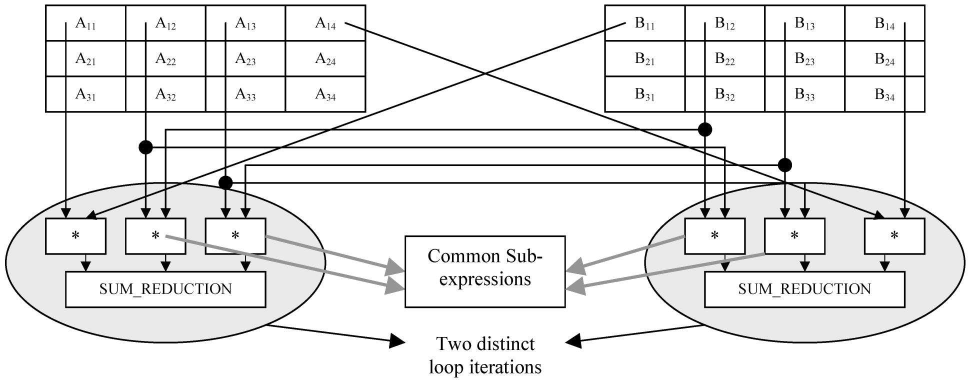


Fig. 8. Common subexpressions.

This framework can be generalized for any loop generating windows of size $M \times N$. The RC array processes $(8 - M + 1)$ iterations in the horizontal dimension and $(8 - N + 1)$ iterations in the vertical dimension, for a total of $[(8 - N + 1) \times (8 - M + 1)]$ iterations between successive data fetches. The following sections describe how this strip-mined version of the loop is synthesized. The compiler assumes that all window generators produce windows that are smaller than or equal to an 8×8 window in size. Since most standard image-processing applications work within this constraint, this is a reasonable assumption to make.

3.2.1 *Windowing Loop Optimizations.* Figure 7 shows a simple program that computes the resultant array, R, for any two given arrays, A and B. The program can be summarized by the following function:

$$R[x][y] = \sum_{a=x}^{x+2} \sum_{b=y}^{y+2} (A[a][b] * B[a][b])$$

The windows generated in two separate iterations of this loop have some common subexpressions. Figure 8 shows the pictorial view of two iterations of this loop. The computations "A_{12} * B_{12}" and "A_{13} * B_{13}" are performed in both iterations and are common subexpressions.

In general, whenever a particular element of the source array appears in multiple windows generated, there could potentially be common subexpressions. In order to eliminate these common subexpressions, the windowing loop must be unrolled so as to expose all dependencies across iterations.

The number of iterations of the windowing loop that need to be unrolled is equivalent to the number of overlapping iterations. For a loop generating an $M \times N$ window with steps of *sh* and *sv* in the horizontal and vertical dimensions, respectively, the number of overlapping iterations, NI, is given by

$$\mathrm{NI} = \mathrm{ceil}(N/sh) * \mathrm{ceil}(M/sv)$$

where ceil(n) returns the largest integer lesser than or equal to n.

However, for window sizes greater than 4 in either direction, it is not possible to fetch all the data corresponding to NI windows into the RC array. Consider a window size of 5×5. The first window in each row corresponds to column 1, and the last window will begin on column 5 and end in column 9. Hence, this requires a total of 9×9 elements, whereas the total size of the RC array is 8×8. For such windows, there will be a total of $(8 - N + 1)$ iterations that need to be analyzed. However, if the source array (of size $IX * IY$, say) is smaller than the RC array itself, then the number of windows is equivalent to $(IY - N + 1)$. Hence, the number of iterations, NI, is modified as follows:

$$\begin{aligned} X &= \mathrm{MIN}(IX, 8) \\ Y &= \mathrm{MIN}(IY, 8) \\ H &= \mathrm{ceil}[\{\mathrm{MIN}(N, Y - N + 1)\}/sh] \\ V &= \mathrm{ceil}[\{\mathrm{MIN}(M, X - M + 1)\}/sv] \\ \mathrm{NI} &= H * V. \end{aligned}$$

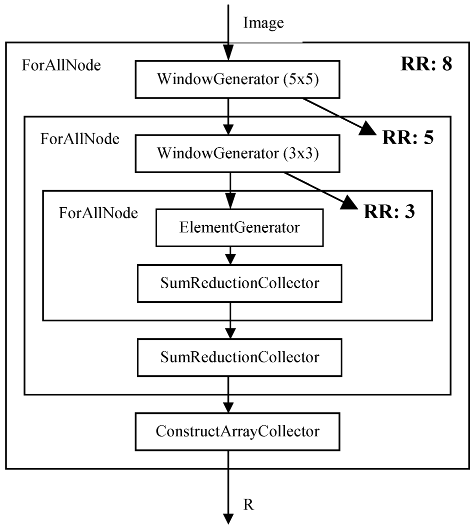
The compiler analyzes these NI iteration windows and eliminates all redundant subexpressions. This gives rise to dead code, which is eliminated as well. At the end of this optimization pass, there will be NI distinct data flow graphs corresponding to each iteration. However, there may be some *crossedges* between these data flow graphs that represent the re-use of computation. These edges are synthesized into registers during the register allocation phase.

This optimization pass only unrolls an element-generating loop that may be embedded within it. If another windowing loop is embedded within this windowing loop, then the inner loop is not unrolled, and is treated as a compound node and is not considered as a candidate node during this optimization pass. This is because analyzing a windowing loop in this manner produces iterations of the order $O(n^2)$. A windowing loop that encloses another windowing loop will have to analyze $O(n^2)$ iterations, where each iteration is an iteration of the windowing loop and contains another $O(n^2)$ iterations. Hence, opening inner window-generating loops in this manner would result in exponential growth.

3.2.2 *Loop Synthesis.* Once the optimizations specified in the previous section are performed, each windowing loop will be associated with NI different loop iterations. These iterations may be different because of redundant

```
Int8[:,:] R =
  For window win[5, 5] in Image {
    Int8 res =
      For window w[3, 3] in win {
        Int8 x =
          For elem in w
          Return (sum(elem));
      } return (sum(x));
  } return (array(res));
```

(a)



(b)

Fig. 9. Resource allocation: for loop hierarchies is based on the vertical dimension of the window generated by the parent loop (is eight for outer-most loops). Example of (a) SA-C program and (b) its equivalent HDFG representation.

computations that may be eliminated. To synthesize the inner loop of the windowing loop, each of these iterations must be separately synthesized. Hence, the synthesis techniques discussed in the following sections will be applied to *each* of the NI iterations. These iterations are never executed concurrently. Hence, the final schedule would just be a linear ordering of each iteration's schedule.

Before using the synthesis techniques mentioned in this section, it is necessary to assign operation latencies and resource requirements to each node (operation) within the loop. All simple nodes will already have been assigned predetermined latencies and resource requirements.

The top-most loop in the loop hierarchy is always assigned a resource requirement of 8—this will ensure that the loop will attempt to fully use all the resources available in the RC array. A windowing loop can have an element-generating loop and/or another windowing loop embedded within it. The resource requirement for an inner loop is defined to be "the vertical dimension of the window generated by its parent loop." This will ensure that the inner loops will attempt to fully utilize the data exposed by the outer loop. Figure 9 shows an example program (a), and its HDFG representation (b). Each loop in the HDFG is annotated with its resource-requirement (RR) assignment.

3.2.2.1 *Operation Scheduling.* The operation-scheduling problem for a windowing loop is defined as finding a schedule that executes in minimum time under two constraints: the availability of resources and the RC array execution mode. There are two modes of execution on the RC array—row mode and column mode. In any given clock cycle, only one of these can be active. However, a node could be scheduled to execute over both modes through multiple clock cycles. Concurrent operations must all execute in the same modes throughout each operation's lifetime.

The operation-scheduling algorithm itself is known to be NP-complete, and usually heuristic algorithms are used. One popular heuristic is the *list scheduling* algorithm. The compiler uses an extension of this algorithm that attempts to schedule nodes on the critical path as early as possible. A node is scheduled only when there are sufficient resources available and when the node's execution mode does not conflict with the execution mode of the schedule generated thus far.

The schedules thus generated (for each of the NI iterations) are then linearly ordered to complete the execution of all the iterations that are present in the RC array. Each set of data fetched into the RC array is subjected to these execution schedules. For a windowing loop generating $M \times N$ windows, the total execution time, T, of the loop over an image of size, $[h, w]$, is given by

$$\begin{aligned} \text{Let } S &= \text{Size of the source image in any dimension} \\ \text{Let } Dt &= \text{Distance between first element of two successive data fetches} \\ &= \text{Number of windows in that dimension, } NW * \text{Window step} \\ &\quad\ \text{in that dimension, } st \end{aligned}$$

Then, number of data fetches in that dimension = S/Dt.

Number of windows in any dimension,

$$NW = \text{ceil}[(X - W + 1)/st]$$

where $X = \text{MIN}(Wp, 8)$, W = Window size in that dimension, Wp = source image size in that dimension (= 8 if outermost loop).

Hence, total data fetches,

$$D = Dh * Dv$$

where Dh = data fetches in horizontal dimension and Dv = data fetches in vertical dimension.

Execution time of window loop

$$T = D, * \sum_{i=1}^{\text{NI}} k_i$$

where k_i = latency of the ith iteration's schedule.

3.2.2.2 *Resource Allocation and Binding.* The RC array is divided into four quadrants each of size 4×4. A given RC cell can directly access only the cells in the same row and column as itself, and in the same quadrant as itself. In Figure 10, for example, cell R_{22} can directly access (in the same clock cycle) cells R_{12}, R_{32}, and R_{42} in the vertical dimension, and cells R_{21}, R_{23}, and R_{24} in the horizontal dimension. Accessing any other cell would incur a communication penalty.

The objective of resource allocation is to minimize these communication latencies. To solve this problem, a special graph is created, where the nodes are operations and edges between nodes indicate "affinity to sharing a resource"

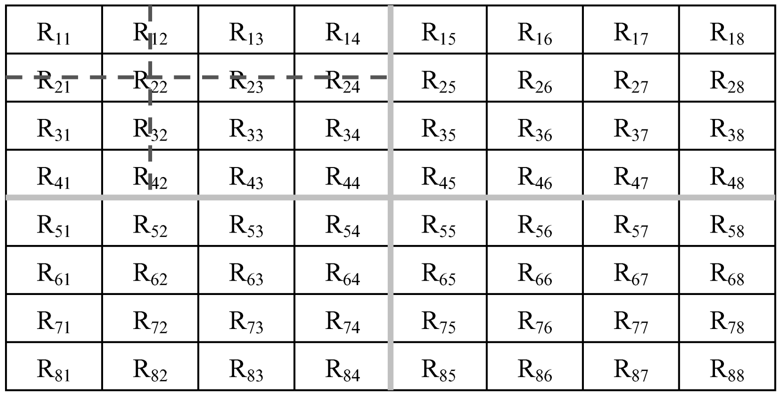

R11 R12 R13 R14 R15 R16 R17 R18
R21 R22 R23 R24 R25 R26 R27 R28
R31 R32 R33 R34 R35 R36 R37 R38
R41 R42 R43 R44 R45 R46 R47 R48
R51 R52 R53 R54 R55 R56 R57 R58
R61 R62 R63 R64 R65 R66 R67 R68
R71 R72 R73 R74 R75 R76 R77 R78
R81 R82 R83 R84 R85 R86 R87 R88

Fig. 10. RC array connectivity.

between the two nodes. These edges, called *shareable* edges, are added as follows:

- Operation-pairs scheduled to execute concurrently do not share any edge.
- All other operation-pairs (that are not scheduled for concurrent execution) share an edge.
- Two nodes that have a direct data-dependence (i.e., an edge in the data flow graph) are assigned a higher weight (say k) than all other nodes (default weight is 1). This is because the result of one operation is the input operand of the other. There will be no communication penalty if the two operations share the same resource. Hence, a weight on an edge gives more importance to it.

Another type of edge, called *closeness* edges, is also added to the graph. These edges reflect the condition in which two nodes are assigned to different resources; however, these resources must be as close to each other as possible. Consider an operation, *op*, which needs two operands that are produced as results of operations, *op1* and *op2*. Then, *op1* and *op2* must be scheduled as close to each other as possible in order to avoid the communication penalty. These edges are added as follows:

- If the operands of a node are produced by two different operations, then these two (source) operations will share a *closeness* edge between them
- The weight on this *closeness* edge is accumulated if more *closeness* edges are generated between the same two nodes.

The graph thus generated is subject to CLIQUE_PARTITIONING.[2] There are two different objectives that need to be satisfied during resource allocation: resource sharing (based on the *shareable* edges) and assignment of resources close to each other (*closeness* edges). To satisfy these seemingly orthogonal objectives,

[2]CLIQUE_PARTITIONING is a popular graph-partitioning algorithm. A *clique* is defined as a fully connected sub-graph.

the compiler performs two levels of clique partitioning:

- Perform CLIQUE_PARTITIONING based on the *shareable* edges. A cluster of nodes thus formed will indicate the nodes that should share a resource.
- Create a new graph by collapsing each clique into a single, unique node.
- Perform CLIQUE_PARTITIONING on this new graph based on the *closeness* edges. *Superclusters* now formed represent a group of nodes, which need to be assigned resources as close together as possible.

One of the components of the CLIQUE_PARTITIONING problem is to find the maximal clique in the graph (MAX_CLIQUE). This problem is known to be NP-complete. The compiler uses a heuristic to solve it—the clique containing the node with the maximum number of edges is assumed to be the best candidate for the maximal clique.

In the end, the graph is a set of "supercliques," where each node in the superclique represents a clique from the first level of clique partitioning. When every clique in the superclique has been assigned a resource, all the operations within that clique will share this resource. The compiler uses a heuristic in assigning resources to the cliques within a superclique. It tries to keep the clique with the largest "closeness requirements" (equal to the sum total of all weights on its *closeness* edges) as close as possible to every other clique within its superclique.

3.2.2.3 *Register Allocation.* Register Allocation strategy for windowing loops uses the same strategies as used by element-generating loops. However, after performing common subexpression elimination, values (represented by *crossedges*) may be forwarded to other iterations. Register allocation is performed in two phases. First, common computation results that are forwarded between iterations are allocated to registers. These registers will be required throughout the entire loop execution between data fetches. Then, registers are allocated to each (of the NI) iterations. However, these registers are alive only during the particular iteration's execution.

3.3 Data Prefetching and Caching

One of the principal hindrances to achieving higher performance is the memory–CPU bandwidth. This factor has been taken into consideration in the design of the Morphosys architecture. The frame buffer in the architecture is designed to perform the function of a "streaming cache." It consists of two sets, each of which in turn is made up two banks. The DMA controller sets up direct data transfers between the main memory and the frame buffer.

The design of the frame buffer is such that its two sets can be independently accessed at the same time. Hence, while data are being loaded into one set, the RC array can fetch data from the other set. Thus, computation can be overlapped with data transfers, thereby reducing the memory latency.

To implement this kind of strategy, the compiler analyzes the concerned kernel's rate of data consumption (implying data loading), its rate of data production (implying data storing), and its execution latency, and inserts data prefetch instructions in appropriate places in order to reduce the memory latency.

Consider a kernel with Ni streaming inputs, and No streaming outputs. Assume that the compiler-generated schedule for the kernel has an overall execution latency of T cycles. Further, assume that this kernel consumes a data slab of size $M \times N$ and produces output slabs of sizes $P_1, P_2, \ldots, P_o$ for each of the No outputs. Memory latency for a typical execution cycle[3] is made up of the time to fetch data for the next execution cycle and the time to write back the results of the previous execution cycle. Hence, the total execution time for an execution cycle is given by

$$\text{Actual execution time, } T_{\text{actual}} = \text{MAX}(T, T_{\text{mem}}(Ni * (M * N) + \Sigma P))$$

where $T_{\text{mem}}(X)$ is the memory latency for fetching X data elements.

The compiler computes these latencies and inserts the data-fetch instructions in the execution cycle prior to the one that needs them, and write-back instructions in the execution cycle following the one that produces them.

The delay due to memory latency exists only if the T_{mem} component of the MAX function is bigger than T. An interesting observation is that the memory latency does not depend directly on the size of the frame buffer. The frame buffer need only be big enough to house all input data required in and output data produced by an execution cycle (which is easily satisfied for most benchmarks). This implies that for the execution of a single kernel, the speed of data transfer, and not the size of the cache buffers, is the principal bottleneck. However, Maestre et al. [1999] discusses an exploration algorithm that analyzes many such kernels, and based on their execution latencies, and data consumption and production rates, determines a schedule for kernel execution as well as for data transfer. In this context, a particular data structure (element) may be accessed multiple times by multiple kernels. Hence, the algorithm examines the trade-offs between executing multiple kernels using the same set of data, and running each kernel to completion, one at a time. In the former case, once a data set is loaded, it satisfies all its consumers (kernels). However, each kernel's configuration code may have to be loaded multiple times. In the latter case, each kernel's configuration code needs to be loaded only once, but data sets may need to be re-loaded. In this context, the size of the pre-fetch cache will make a significant difference.

4. PERFORMANCE MEASUREMENTS

This section discusses the performance measurements of applications compiled for the Morphosys architecture. In addition, it also evaluates the efficiency and benefits of the loop optimizations that are performed by the compiler. It first describes the framework and methodology used to evaluate the performance of the compiler; then, the representative applications that have been used for the purpose of measurement are described; and finally, the results are presented.

[3]A cycle in this context does not refer to the clock cycle. It refers to an "execution cycle" during which the kernel performs all its computations on the given input data slab. It can be thought of as an equivalent of one (or more) loop iterations.

The approach used to test the efficiency of the compiler is to examine the performance of the instruction schedules generated by the compiler for certain sample applications written in SA-C. These applications are also executed separately, using the same test data, under Windows 2000 on an 800 MHz Pentium III platform. For this purpose, the applications are written in native C code and are compiled using the VC++ 6.0 compiler with the highest level of optimization turned on. Then, the resulting binary is executed repeatedly (for a million iterations) on the Pentium platform. We ensure that this is the only application active during the kernel's execution. We time the execution run, and divide it by the number of iterations to obtain the execution time for the given kernel. While it is true that the operating system could influence the total execution time, we believe that the large number of execution runs can offset this error. The results of the two execution times are compared and contrasted. The section also presents comparison between optimized and unoptimized schedules to examine the usefulness of the compiler loop optimizations.

4.1 The Applications

The test applications used for the purpose of performance evaluation represent important application kernels from the image-processing domain. These applications are chosen to describe a variety of behaviors, differing both in the type of loops they contain and in the kind of inner loop computations that need to be performed.

Wavelet Compression. Wavelets are commonly used for multiscale analysis in computer vision, as well as for image compression. Honeywell has defined a set of benchmarks for reconfigurable computing systems, including a wavelet-based image compression algorithm. The wavelet program has been translated into SA-C and generalized to operate on any size image. The algorithm works on 5×5 windows of the source image.

Prewitt Edge Detection. This is an edge detection program that calculates the square root of the sum of the squares of responses to horizontal and vertical Prewitt edge masks. Since this same task can be performed using the Intel Image Processing Library (IPL), the results can be compared with that of a hand-optimized Pentium program. The Prewitt edge detection masks are one of the oldest and best understood methods of detecting edges in images. Basically, there are two masks, one for detecting image derivatives in X and one for detecting image derivatives in Y. To find edges, a user convolves an image with both masks, producing two derivative images (dx and dy). The strength of the edge at any given image location is then the square root of the sum of the squares of these two derivatives. (The orientation of the edge is the arc tangent of dy/dx.). This particular implementation of the algorithm works on 3×3 windows using 3×3 horizontal and vertical masks.

Motion Estimation for MPEG. Motion estimation helps to identify redundancy between frames in an MPEG video stream. The most popular technique for motion estimation is the block-matching algorithm [Hsieh and Lin 1992]. This algorithm is one of the kernels in the MPEG-4 compression algorithms.

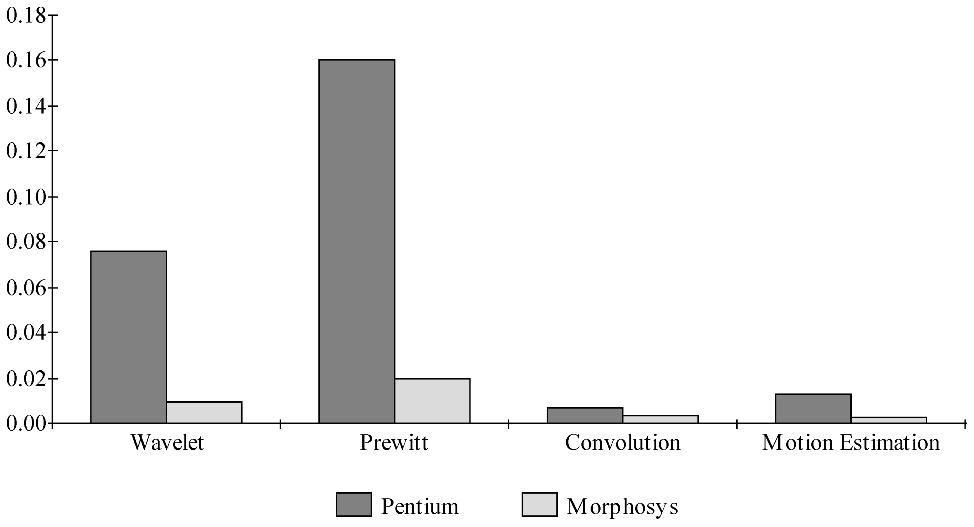


Fig. 11. Performance comparison.

2D Convolution. This a common application used in digital signal processing. It computes the linear convolution of every 3×3 window in the source image with a 3×3 kernel size. This algorithm is a part of the Intel Image Processing Library (IPL) as well as the Vector, Signal and Image Processing Library (VSIPL). (The VSIPL forum is a volunteer organization made up of industry, government, users, and academia representatives who are working to define an industry standard API for vector, signal, and image processing primitives for embedded real-time signal processing systems).

4.2 Results

All applications were executed using 8-bit 512×512 source image(s). The Morphosys processor runs at a clock frequency of 200 MHz. Figure 11 compares the compiled codes running on Morphosys with execution on the Pentium platforms. The execution time is specified in seconds.

In order to understand why Morphosys out-performs Pentium every time, let us consider the profile of these benchmarks. All the benchmarks discussed here are streaming applications—they read regular, predictable chunks of data from memory, perform certain operations on this data, and write-back another regular chunk to a predictable memory location. This represents one iteration of the kernel loop. In order to maximize performance, we must ensure two things: timely availability of data to the kernel and maximal usage of computational resources (RC array) to reduce the kernel execution latency. We will address both these factors in turn.

Timely availability of data is ensured through data prefetching. The Morphosys architecture is tailored to accelerate the throughput of streaming multimedia applications. The architecture supports direct memory access (DMA), which ensures that data can be streamed through the RC array at a steady rate. Once the TinyRisc instruction to fetch (write-back) a data stream

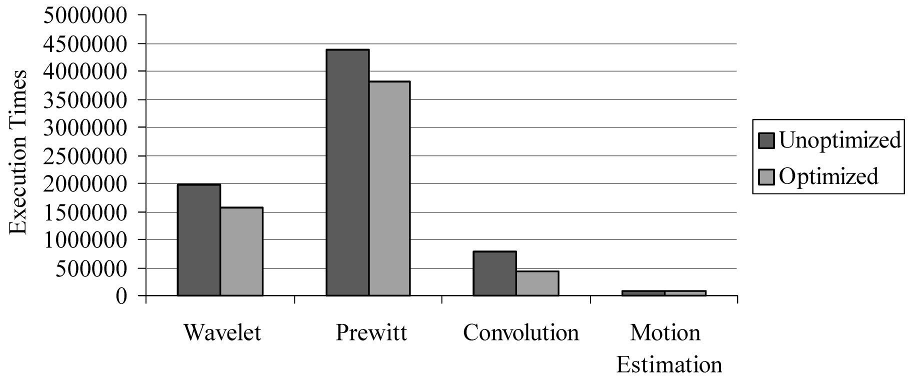


Fig. 12. Effect of compiler optimizations.

is issued, the latency of the data transfer is completely deterministic. Moreover, with the absence of pointers in the SA-C language, the details of all memory accesses are known at compile-time. Hence, there are no variable latency operations, and the compiler uses this knowledge to perform a combination of latency estimation, data prefetching and instruction scheduling, to ensure that data is always available just before it is required. In contrast, the Pentium processor is not aware of the profile of these benchmarks, and has to access memory through a cache hierarchy, resulting in nondeterministic, variable access latency.

The hybrid SIMD/MIMD execution model of Morphosys can, essentially, be perceived as an 8-wide processor with an 8-stage pipeline. The compiler schedules the operations within each kernel such that processor bandwidth is best utilized. The objective of the compiler is to keep every RC cell in the RC array in busy every cycle. As long this can be achieved, the computational bandwidth of the RC array will far exceed that of a Pentium processor. In contrast, the Pentium III processor is restricted to a 4-wide issue pipeline.

Convolution and *motion estimation* are tiny kernels, and their execution latencies are smaller than the data transfer latencies. Hence, the compiler is unable to completely hide the data transfer latency by prefetching, and the execution stalls most of the time waiting for data. Consequently, the performance of these kernels on Morphosys is comparable to that on Pentium. However, *Prewitt* and *Wavelet* are relatively large kernels.[4] Hence, the RC array rarely stalls, and the execution latency of benchmarks corresponds closely to the available data parallelism.

Figure 12 shows the effect of performing loop optimizations (the execution time is specified as number of clock cycles). The only application that does not benefit is the *motion estimation* kernel. This is because the application works on an element-generating loop, and the iterations of element-generating loops are all independent of each other. Hence, there is no opportunity to perform any of the inner loop optimizations.

[4]Prewitt, infact, has a square root operation. This results in a big execution latency for the kernel.

5. CONCLUSIONS AND FUTURE DIRECTIONS

5.1 Conclusions

This paper presents a methodology and framework that enables the efficient compilation of applications written in a high-level language to a reconfigurable computing architecture. In particular, the compiler aims at extracting the data parallelism at both coarse- and fine-grained levels in a given application, and producing an instruction schedule that explicitly reflects a SIMD computational model. It describes how an image-processing application written in SA-C is partitioned and how it can be executed on the Morphosys architecture. In doing so, it describes how data parallel semantics of the program are identified, analyzed, and mapped for execution on the RC array of the Morphosys architecture.

It describes the synthesis approach of the mapping process, which performs operation scheduling, resource binding, and register allocation, in order to produce an execution schedule. In the process, a number of algorithms that need to be used in the mapping process are proposed. Also, different compiler optimizations are proposed that could potentially improve the execution time of applications on the target platform. It also discusses the data transfer and caching issues that could greatly alleviate memory latencies.

The performance results presented in the previous section show that the compiler-generated schedule can achieve an average speedup of up to 6× for the tested benchmarks. Also, it shows that the loop optimizations performed by the compiler could potentially produce significant improvements in the execution times.

5.2 Future Work

The work presented here is a first step in the direction of automatic compilation for the Morphosys platform. It presents an approach to efficiently analyzing the computation-intensive kernels in image-processing applications, and mapping them onto the RC array of the platform. There are, however, a number of other issues that could be addressed as the next steps in improving this process:

- *InterKernel Analysis*: The current model is designed to analyze a single computational kernel in a given image-processing application, and then map it to the Morphosys architecture for near-optimal execution. A standard image-processing application such as automatic target recognition (ATR), for example, may have a number of different kernels, which may interact with one another. Issues such as maximizing data re-use and minimizing reconfiguration time are critical to an optimal execution schedule. Some research efforts [Kaul et al. 1999; Maestre et al. 1999] have already looked into this, but special analysis is required in the context of the Morphosys architecture. Hence, a future step would be to incorporate these issues into the current compiler model.
- *Exploration*: The current compilation model to map loops is to fit as many loop iterations in the RC array as possible. This is done by unrolling and strip-mining the loop iterations as much as necessary. However, it may sometimes

be beneficial to restrict this unrolling, so that other operations within the loop body could be performed concurrently. The compiler has to decide the optimal unrolling/stripmining factor that might produce the best schedule in terms of execution times. For this purpose, the compiler may have to produce a number of different schedules and compare their execution times.

- *Pipelining*: In the current model, the compiler adopts a bottom-up approach to synthesize the different nodes within a loop. In the process, compound nodes are created that are annotated with execution times and resource requirements. When scheduling these nodes, the compiler devotes to them all the resources needed for the entire duration of its execution latency. However, a node may not need all of these resources for the entire duration of the operation. As a next step, the compiler could analyze such situations, and pipeline them to optimally use all the resources.
- *Extending the Compilable Domain*: Currently, only a restricted subset of the whole SA-C language can be mapped to the Morphosys architecture. A number of reduction operators, loop generators, and language semantics have not been analyzed. The main reason for this is that currently only the most essential features that need to be mapped onto the RC array have been analyzed. Moreover, some of the language features cannot be directly mapped to the Morphosys architecture as yet. For example, the SA-C language supports variable bit-width precision, and extracting a column slice or a plane from an array. Currently, these features cannot be implemented on the Morphosys architecture.

REFERENCES

Bondalapati, K. and Prasanna, V. K. 2000. Loop pipelining and optimization for run time reconfiguration. In *Reconfigurable Architectures Workshop*.

Crenshaw, J. W. 2000. *MATH Toolkit for Real-Time Programming*. CMP Books.

Ebeling, C., Cronquist, D. C., and Franklin, P. 1996. RaPiD—reconfigurable pipelined datapath. In *Proceedings Field Programmable Logic*.

Filho, E. M. C. 1998. The TinyRISC instruction set architecture. www.eng.uci.edu/morphosys/docs/isa.pdf.

Frigo, J., Gokhale, M., and Lavenier, D. 2002. Evaluation of the Strems-C to FPGA compiler: An application perspective. In *9th International Symposium on Field Programmable Gate Arrays*. Monterey, CA.

Goldstein, S., Schmit, H., Budiu, M., Cadambi, S., Moe, M., and Taylor, R. R. 2000. PipeRench: A reconfigurable architecture and compiler. *IEEE Computer 33*, 70–77.

Hall, M., Diniz, P., Bondalapati, K., Ziegler, H., Duncan, P., Jain, R., and Granacki, J. 1999. DEFACTO: A design environment for adaptive computing technology. In *6th Reconfigurable Architectures Workshop* (RAW'99).

Hammes, J. P. and Böhm, A. P. W. 2001. The SA-C language. www.cs.colostate.edu/cameron Colorado State University.

Hammes, J. P., Rinker, R. E., McClure, D. M., Böhm, A. P. W., and Najjar, W. A. 2001. The SA-C compiler dataflow description. www.cs.colostate.edu/cameron Colorado State University.

Hammes, J., Rinker, R., Böhm, W., Najjar, W., Draper, B., and Beveridge, R. 1999. Cameron: High level language compilation for reconfigurable systems. In *Conference on Parallel Architectures and Compilation Techniques*. Newport Beach, CA.

Hsieh, C. and Lin, T. 1992. VLSI architecture for block-matching motion estimation algorithm. *IEEE Transactions on Circuits, Systems for Video Technology 2*, 169–175.

KAUL, M., VEMURI, R., GOVINDARAJAN, S., AND OUAISS, I. E. 1999. An automated temporal partitioning and loop fission for FPGA-based reconfigurable synthesis of DSP applications. In *36th Design Automation Conf.* New Orleans, LA.

KENEDY, K. AND ALLEN, R. 1987. Automatic translation of FORTRAN programs to vector forms. *ACM Transactions on Programming Languages and Systems 9*, 491–542.

LEE, M. ET AL. 2000. Design and implementation of the Morphosys reconfigurable computing processor. *Journal of VLSI and Signal Processing Systems*.

LI, Y., CALLAHAN, T., DARNELL, E., HARR, R., KURKURE, U., AND STOCKWOOD, J. 1999. Hardware-software co-design of embedded reconfigurable architectures. In *Design Automation Conf. (DAC)*.

MAESTRE, R. ET AL. 1999. Kernel scheduling in reconfigurable computing. In *Design and Test Europe* (*DATE*). Munich, Germany.

PETERSON, J. B., O'CONNOR, R. B., AND ATHANAS, P. M. 1996. Scheduling and partitioning ANSI-C programs onto multiple FPGA CCM architectures. In *IEE Symposium on FPGAs for Custom Computing Machines*. Napa, CA,

SINGH, H., LEE, M. H., LU, G., KURDAHI, F. J., BAGHERZADEH, N., AND FILHO, E. M. C. 2000. MorphoSys: An integrated reconfigurable system for data-parallel and computation-intensive applications. *IEEE Trans. Comput. 49*, 5, 465–481.

VENKATARAMANI, G. 2001. A compiler framework for mapping applications to a coarse-grained reconfigurable architecture. M.S. Thesis, University of California Riverside.

WAINGOLD, E., TAYLOR, M., SRIKRISHNA, D., SARKAR, V., LEE, W., LEE, V., KIM, J., FRANK, M., FINCH, P., BARUA, R., BABB, J., AMARSINGHE, S., AND AGRAWAL, A. 1997. Baring it all to software: Raw machines. *IEEE Computer 30*, 86–93.

WAWRZYNEK, J. AND CALAHAN, T. J. 1998. Instruction-level parallelism for reconfigurable computing. In *8th International Workshop on Field Programmable Logic and Applications*. Berlin, Germany.

YE, Z. A., MOSHOVOS, A., HAUCK, S., AND BANERJEE, P. 2000. CHIMAERA: A high-performance computer architecture with a tightly-coupled reconfigurable unit. In *International Symposium on Computer Architecture* (*ISCA*). Vancouver, BC, Canada.

Received March 2002; revised August 2002; accepted November 2002