Hindawi Publishing Corporation
Journal of Electrical and Computer Engineering
Volume 2012, Article ID 862469, 26 pages
doi:10.1155/2012/862469

Research Article

Automated Generation of Custom Processor Core from C Code

Jelena Trajkovic,[1, 2] Samar Abdi,[3] Gabriela Nicolescu,[2] and Daniel D. Gajski[1]

[1] *Center for Embedded Computer Systems, University of California, Irvine CA 92697, USA*
[2] *École Polytechnique de Montréal, Montreal, QC, Canada H3C 3A7*
[3] *Electrical and Computer Engineering Department, Concordia University, Montreal, QC, Canada H4B 1R6*

Correspondence should be addressed to Jelena Trajkovic, jelena.tr@gmail.com

Received 7 June 2011; Revised 14 November 2011; Accepted 21 November 2011

Academic Editor: Yuan Xie

We present a method for construction of application-specific processor cores from a given C code. Our approach consists of three phases. We start by quantifying the properties of the C code in terms of operation types, available parallelism, and other metrics. We then create an initial data path to exploit the available parallelism. We then apply designer-guided constraints to an interactive data path refinement algorithm that attempts to reduce the number of the most expensive components while meeting the constraints. Our experimental results show that our technique scales very well with the size of the C code. We demonstrate the efficiency of our technique on wide range of applications, from standard academic benchmarks to industrial size examples like the MP3 decoder. Each processor core was constructed and refined in under a minute, allowing the designer to explore several different configurations in much less time than needed for manual design. We compared our selection algorithm to the manual selection in terms of cost/performance and showed that our optimization technique achieves better cost/performance trade-off. We also synthesized our designs with programmable controller and, on average, the refined core have only 23% latency overhead, twice as many block RAMs and 36% fewer slices compared to the respective manual designs.

1. Introduction

In order to implement an application, the designer typically starts from an application model and a set of constraints, such as performance and cost (Figure 1). One of the primary design decisions is to define the hardware-software (HW/SW) partitioning of the application [1]. In case of noncritical applications, the entire application may be implemented in software. The designer selects the target processor and compiles the applications for the fixed instruction set of the selected processor. The generated binary is executed on the target processor. This is the most flexible solution, since the target processor is programmable. As such, the designer may change the application code and recompile to modify the implementation. In case when part of an application, or the entire application, has tight performance, power, or area constraints, the designer may decide to implement the critical part of the application in hardware. For hardware implementation, the designer may use preexisting intellectual property (IP), use high-level synthesis (HLS) tools or manually design the hardware. The generated hardware is highly optimized, but nonprogrammable.

In order to obtain the advantages of both approaches, that is, programmability and high design quality, the designer may opt for application-specific processor (ASP). Application-specific processor cores are being increasingly used to address the demand for high performance, low area, and low power consumption in modern embedded systems. In case of application-specific instruction set processors (ASIPs), the designer first decides on the instruction set (IS) then generates the hardware and the compiler to support the chosen IS. The other possibility is to generate data path first, and then to generate the controller either manually or automatically [2]. The advantage of the latter method is the possibility to remove IS and decoder from the design process and the design itself, respectively. Removing the decoder significantly simplifies the design and verification. Furthermore, the controller generates a set of control signals that directly drives the data path, which allows for having any combination of control signals possible, not just the

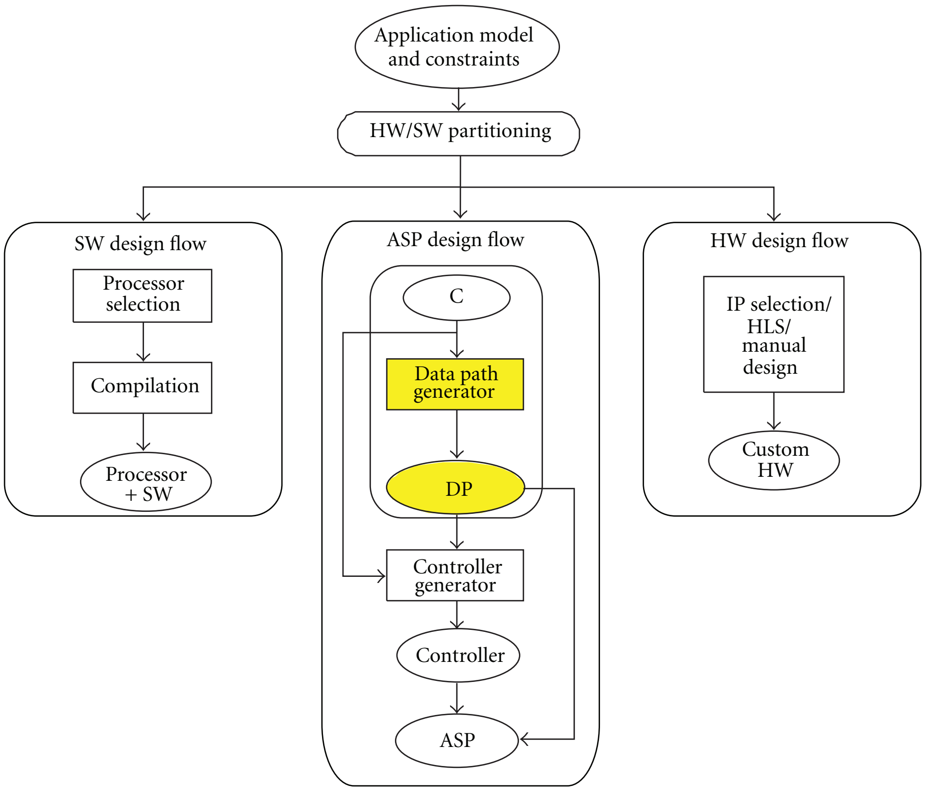


Figure 1: Proposed design technique within system-level design flow.

combinations that are allowed by instructions from a predefined IS. Moreover, it facilitates having compiler [3] that does not need to be modified every time that IS changes. Therefore, in this work, we adopt the methodology where the data path gets created separately from the controller.

In general, the challenges of ASP design are as follows:

(1) design of the data path, that is, selection of the best set of components (functional units, register files, buses, memories, etc.) and connections;

(2) design of the controller;

(3) ability to scale well with large code size;

(4) controllability of the design process.

In this work, we deal with (1) and (4), and the aspect of (3) that applies to data path creation, where (2) has been published in [3, 4].

1.1. Custom Processor Design Methodology. The design of application-specific cores is nontrivial and the problem is further exacerbated by the size and complexity of the application as well as the short time to market. Traditional C to RTL automation tools have so far had little impact. This is because they combine the optimization of data path architecture and controller, which makes the process unscalable and uncontrollable. We overcome this problem by applying a design strategy where the data path and controller are designed separately. The architecture is derived by analyzing the C code and designer-specified constraints. The C code is then compiled into either control words (control word is a collection of control signals that drive the data path) for programmable cores or FSM for hardwired cores.

Our target processor core is similar to an ASIP, with the exception of the instruction decoder. The C application is directly compiled into microcode that drives the data path. This approach removes the unnecessary restriction of an instruction set and provides the abovementioned advantages. The target processor core template is shown in Figure 2. First, we construct the data path on the right-hand side by selecting, configuring, and connecting various functional and storage units. Then, we develop the microcoded or hardwired controller (on the left-hand side) for the constructed data path. During core construction, the data path is automatically refined to meet the given performance and resource constraints. The generated data path is *pareto-optimal* for the given application, but it may execute any other application if it contains all the components required by the application. However, the execution may not be optimal. By separating data path and controller design, we allow simplified optimization of the controller by removing the data path optimization parameters from the equation. The scheduling and binding, that is, controller generation, are performed once the data path has been designed. The controller generation may be manual or automated, as presented in [3]. Details of separation of data path generation from scheduling and binding may be found in [2, 5].

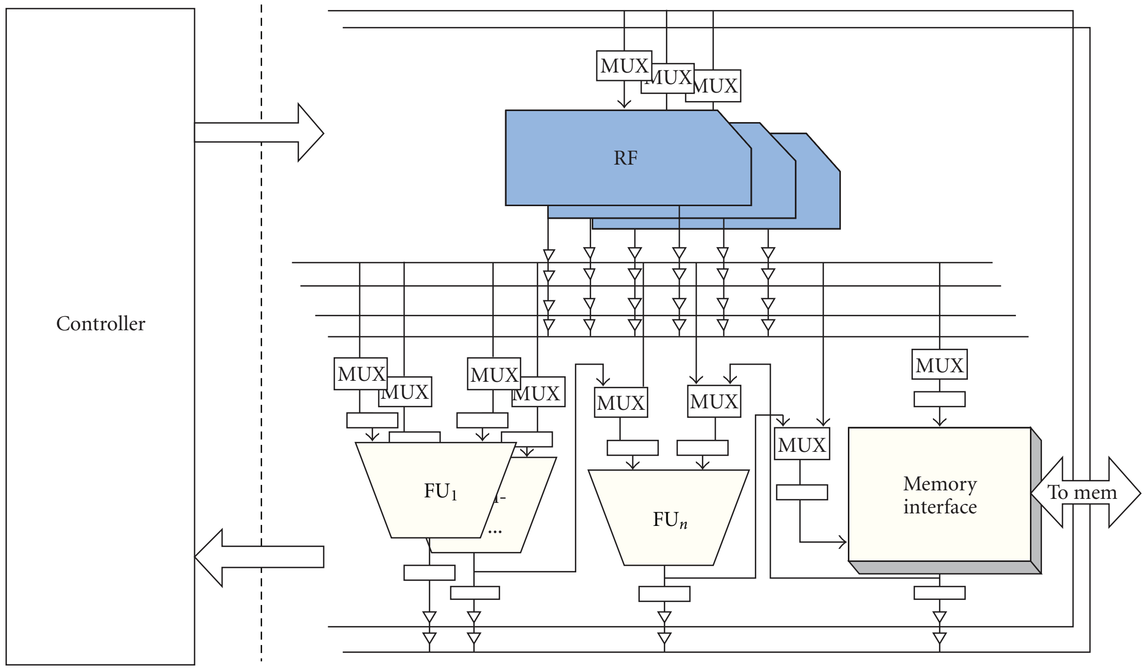


FIGURE 2: Proposed processor core template: the data path and the controller are created separately.

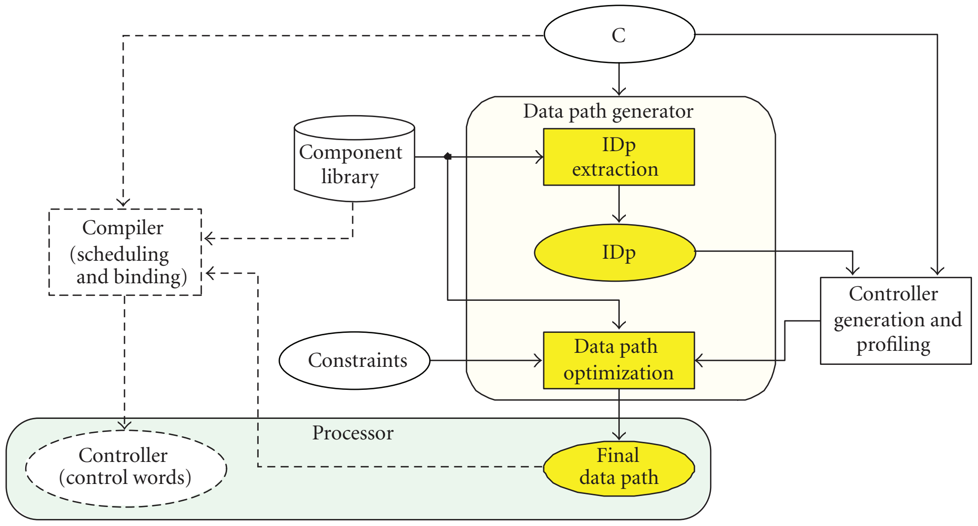


FIGURE 3: Data path extraction steps.

Automating the design process in the proposed way has several advantages.

(1) Designers use their expertise to guide the tool instead of performing cumbersome and error-prone HDL coding.

(2) The tool automatically produces a design in a fraction of time while having only marginal performance degradation and almost the same total area as the manual design.

(3) The designer may change *only* the constraints and iterate over the optimization phase.

Figure 3 shows the steps of proposed extraction technique. In the first step, called *Initial Data path* (IDp) *Extraction* (Section 2), source code for a given application is analyzed in order to extract code properties that will be mapped onto hardware components and structures and the *Initial Data path* is created. The Initial Data path is used for controller generation and profiling of given application code. The profiled code is then analyzed and the data path undergoes several iterations of refinement and estimation during the *Data path Optimization* (Section 3) step. Data path Optimization step first selects the portion of code to be optimized (Section 3.1), converts partial resource constraints to timing overhead (Section 3.3), and estimates execution characteristics of intermediate candidate designs (Section 3.4) until the specified overhead is met. The final data path is used for scheduling and binding, and controller generation. As explained above, the controller generation is out of scope of this work.

2. Initial Data Path Extraction

The stepwise process of creating the initial data path is shown in Figure 4. Based on a target-independent scheduling of the C-code instructions, we identify a set of properties for the given code. Then, we use the mapping function to map the code properties to available hardware elements in the Component Library and to generate a set of hardware components and structures/templates that correspond to the reference C code.

The code properties include operators and their frequency, data types and their values, control and data dependencies, existence of loops, loop nests, size of the loop body, and the available parallelism. The hardware elements include different types of parameterizable functional units, like adders, multipliers, and comparators; storage units, such as registers, register files, and memory; interconnect components, like busses, multiplexers, and tristate buffers. We consider different data path structures/templates: templates with a different connectivity schemes, like bus-based, multiplexer-based, and dedicated connections; templates with pipelined data path, pipelined units or both, and so forth. For instance, the available parallelism translates into the number of instances of functional units, the number of registers, and the number of register file ports. It may also determine, together with type of dependencies in the code, if the data path should be pipelined. Also, the data types and number of used bits to represent the data determines the bit-width of used components and buses.

The maximum requirements of a given application is obtained from the application's "as late as possible" (ALAP) schedule (produced by Pre-Scheduler). The ALAP schedule is in a form of Control Data Flow Graph (CDFG) and is target independent. The underlying assumption is that there are sufficient resources to execute all existing operations. Front end of any standard compiler may be used as a Pre-Scheduler. In this case, the CDFG is output of front end of the Microsoft Visual C++ compiler. We choose ALAP because it gives good notion of the operations that may be executed in parallel. Please note that only the properties required for the data path generation (described in this section) are extracted from the CDFG.

The IDp *Extractor* traverses the given ALAP schedule, collecting the statistics for each operation. For each operation (addition, comparison, multiplication, etc.) maximum number of its occurrences in each cycle is computed. The tool also records the number of potential data transfers for both source and destination operands. For example, if the ALAP schedule assigns three additions in the same cycle, then three units that perform addition will be allocated, together with sufficient number of busses, register files, and memories to provide for reading and writing of all operands. However, since the resources are selected from the elements available in the Component Library, the resulting architecture may have different number of instances than desired. One such example would be the case where the application requires three adders, but only a register file with one input and two output ports is available, and no forwarding is possible. Therefore, it would be reasonable to allocate *only one* adder, because there would be no sources for operands of the remaining two adders.

In addition to application's schedule, Component Library (CL) is another input of the Initial Data path Extraction. The Component Library consists of hardware elements, that are indexed by their unique name and identifier. The record for each component also contains its type, number of input and/or output ports, and name of each port. In case of a functional unit, a hash table is used to link the functional unit type with the list of all operations it may perform. Later in the section, we also present a set of heuristics that measure how well the available storage components match the given requirements. The heuristics use quantifiers for the code properties, translate them to the required number of hardware elements, and map them to the available components in the Component Library.

We chose a load-store data path model. This widely used model has specialized operations that access memory (load and store instructions) and any other operation reads operands from the register file and writes the results back to the register file. Also, we explore the data path models that do not have any forwarding path. Therefore, to ensure that the interconnect is not a bottleneck, for the Initial Data path, we perform a greedy allocation of connectivity resources (Figure 7). This means that the output ports of all register files are connected to all the source buses. Similarly, the input ports of all register files are connected to all the destination busses. The same connection scheme applies to the functional units and the memory interface.

2.1. Application Requirements

2.1. Application Requirements. In order to design an application-specific processor core, we must first determine the application requirements. These requirements must be quantized in some form for analysis and eventual selection of matching components. There are primarily three aspects of the application that we are concerned with, namely, computation, communication, and storage. In terms on the architecture, these aspects translate to operator usage, operand transfer, and type of storage.

The set of properties of the C code include data types, operators, variables, parallelism, loops, and dependencies. The components include functional units, storage elements (registers, register files, memories), and interconnect like buses, multiplexers, and tristate buffers, while structures/templates refer to different connectivity templates, like bus-based interconnect, multiplexer-based interconnect, dedicated connections, data path or component pipelining, and load/store architecture model. For example, the data types and number of bits used to represent them would determine the bit width of used components; the available parallelism would influence the number of instances of

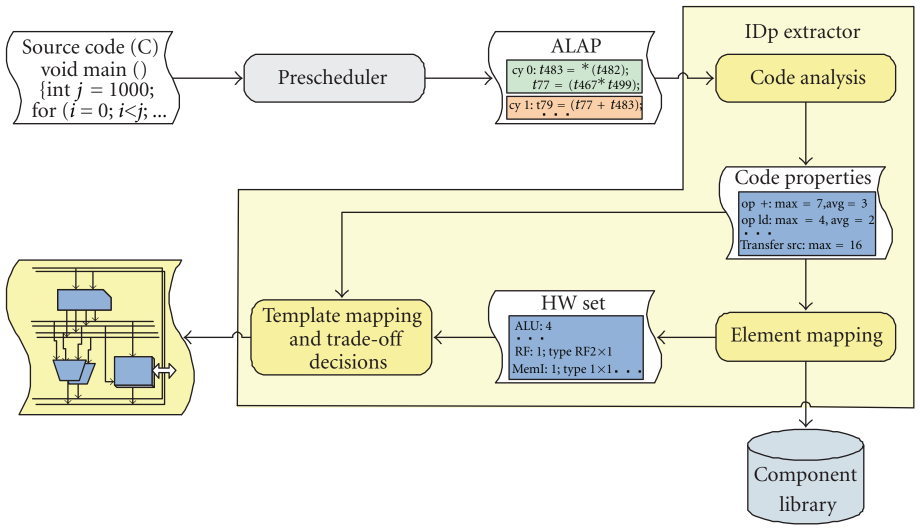


FIGURE 4: Initial data path (IDp) extraction flow.

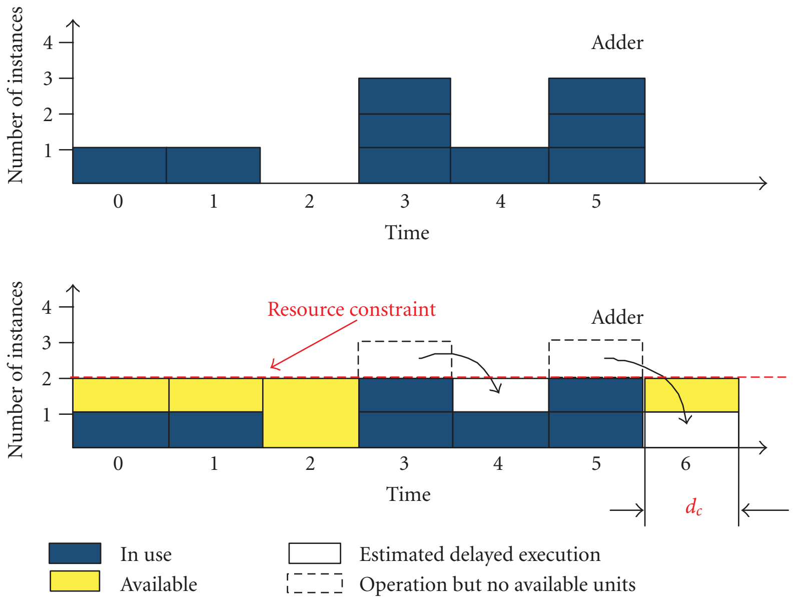


FIGURE 5: d_c Computation for given resource constraint.

functional units, number of registers or register file ports, and pipelining.

While the selection of function units and register files corresponds directly to the operations and variables, respectively, the communication needs more analysis. The operands for each operation are classified into either *source operands* or *destination operands*. The simplest case is the arithmetic operation, which typically has two or more source operands and one destination operand. The load operation has one source operand, the memory address, and one destination operand, the read variable. On the other hand, the store operation has two source operands: the address and the variable.

In a typical load/store architecture, like the one used in our work, the source and destination operands correspond to buses that go in and out of the register file. Each transaction, such as the fetching or storing of operand, must be performed using a bus. If multiple operations are scheduled to execute in parallel, multiple source and destination buses are required. One of the code properties we consider for design is the maximum number of source and destination buses that will be used. These numbers can be obtained easily by analyzing the prescheduled C source code.

The first step is to extract properties form the C code. Code property may be extracted from high level code, its

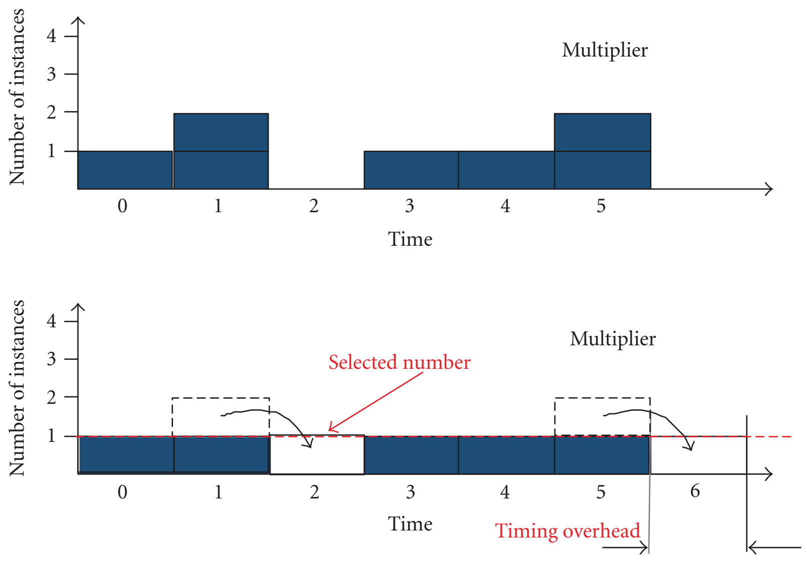


FIGURE 6: Balancing the number of instances of multiplier for given timing overhead.

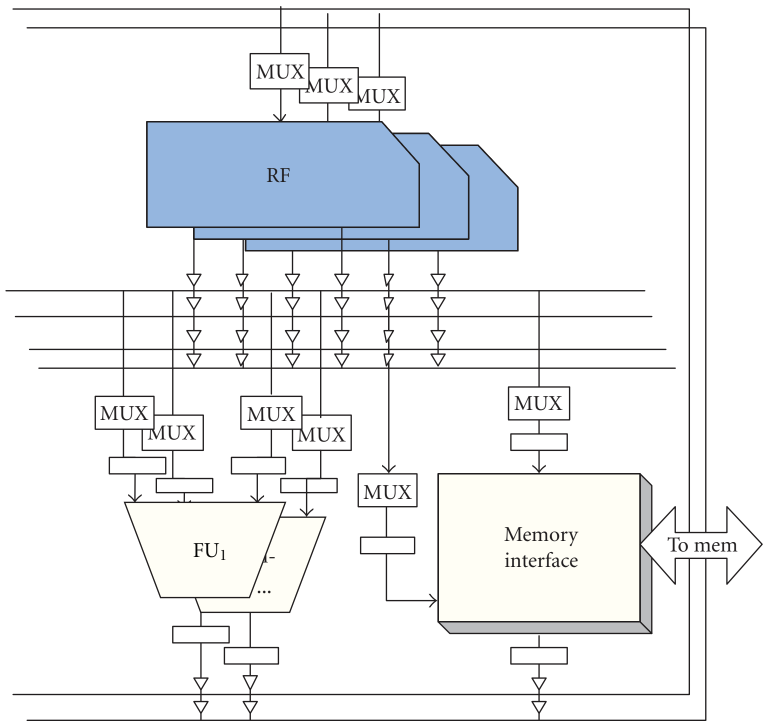


FIGURE 7: Used data path template.

CDFG, or any other intermediate representation generated by a compiler front end. We chose to start form an architecture-independent schedule that assumes no resource constraints, such as ASAP and ALAP. We chose ALAP as the starting point since our experiments show that the parallelism is more evenly distributed across cycles than in corresponding ASAP. The following properties are extracted from code's ALAP schedule:

(i) *OP*: a set of operations for the given application code;

(ii) m_{op}: the maximum number of concurrent usages of operand op;

(iii) m_s and m_d: the maximum number of concurrent data transfers of the source and destination operands, respectively.

2.1.1. Example. We present results for m_{op} value for various operations in typical multimedia functions to provide an idea of the application profile. The example shown here is

Mp3. The size is over 10000 lines (Table 4). The *Mp3* decoder is the example for decoding mp3 frames into pulse-code modulated (PCM) data for speaker output.

The operations are divided into separate classes depending on their type and implementation requirements. We classify the operations into *arithmetic, logic, comparisons, and load/store*. The tables below present the maximum number of concurrent operations (m_{op}) of each operation type.

The m_{op} values for the arithmetic operations, such as *add, subtract, multiply, divide,* and *remainder* are shown in the left-hand side of Table 1. It can be seen that *Mp3* has division operation that may require a divider or be implemented as software library in C. We can also note the high degree if concurrency. This is because the loop in the function is completely unrolled and all the array indexes are changed to different variables for optimization in software. A tool that performs component allocation for maximum performance based on available concurrency would produce a huge design. This extreme example points to the need for designer controllability in making component allocation decisions. The example also points to the fact that application optimizations for traditional processor architectures may not necessarily produce better processor designs. Indeed, the quality of application-specific processor design strongly depends on the structure of the C code in consideration.

The right-hand side of Table 1 shows the concurrency available for logic or bit-wise operations. We can see that not many of thebit-wise operations have been used, even though they are computationally less expensive than arithmetic operations. Some compiler optimizations make transform arithmetic operations involving constants into shifts and other logic operations. This is called strength reduction and is also useful code optimization that potentially may result in lower cost processor designs.

The left-hand side of Table 2 shows the concurrency available in load/store operations for the benchmarks. This concurrency typically does not translate into execution since memories typically have 1-2 ports for reading or writing data.

There is no concurrency available in the comparison operations as seen from right-hand side of Table 2. This is to be expected from the C code structure. Comparison operations are typically used for condition evaluation in if-then-else statements or loops. In the case of loops, the comparison is the only operation in a basic block. Since we are analyzing an ALAP for each basic block of the application, we consider concurrency inside the basic block only. In the case of if-then-else statements, the comparison is the last operation in a basic block that does not have any other comparisons. Therefore, comparisons cannot be executed in parallel due to the ALAP data structure constraints.

Available concurrency in variable reads and writes is shown by the number of concurrent Read Variable operation (262) and the number of Write Variable operations (196). For each basic block ALAP, the operations in the starting state are read operations that fetch the operands from the register file. Similarly, the terminating states write the variables back to the register file. The larger the basic block, the higher the possible concurrency in reading and writing variables, especially if the variables inside the basic block are independent. For our example, we have large basic blocks. This is due to the loop unrolling optimization on the source code, as discussed earlier. Furthermore, the *array elements*

Table 1: Application requirement for arithmetic and logic operations for Mp3 application.

OP—operation	m_{op}	OP—operation	m_{op}
Add	223	And	2
Sub	109	Or	1
Mul	73	Xor	1
Div	1	Neg	2
Div_Un	1	Not	1
Rem	1	Shr	90
Rem_Un	1	Shr_un	2
		Shl	15

Table 2: Application requirement for load and store and comparison operations for Mp3.

OP—operation	m_{op}	OP—operation	m_{op}
Ldind_I1	1	LessThan	1
Ldind_I2	1	LessThan_Un	1
Ldind_I4	32	LessOrEqual	1
Ldind_U1	3	LessOrEqual_Un	
Ldind_U2	2	Equal	1
Ldind_U4	10	NotEqual	1
Stind_I1	1	GreaterOrEqual	1
Stind_I2	1	GreaterOrEqual_Un	1
Stind_I4	20	GreaterThan	1
		GreaterThan_Un	1

Table 3: Register file (RF) mapping function values and functional unit mapping.

RF configuration	Value	Functional unit	Value
RF 2×1	715	Alu	223
RF 3×1	713	Multiplier	73
RF 4×1	711	Divider unsigned	1
RF 4×2	710	Divider signed	1
RF 6×3	705	Comparator	1
RF 8×4	700		
RF 16×8	**668**		

Table 4: Parameter values and code size (LOC).

Benchmark	$(P_l,P_f,P_{fl})[\%]$	LoC	Gen. T [sec]	
			Nonpipe	Pipe
bdist2	(60,50,45)	61	0.2	0.8
Sort	(80,60,45)	33	0.1	0.1
dct32	(18,65,50)	1006	1.3	2.3
Mp3	(30,55,50)	13898	15.6	42.6
inv/forw 4×4	(50,45,55)	95	0.2	0.8

are represented as independent variables which lead to high concurrency available for reading and writing variables. Since the variables are read from output port of register files and written to the input ports, these numbers are indicative of a desirable I/O configuration for the register file.

Finally, the available concurrency in data transfers is shown by the number of data transfers of source m_s and the data transfers of destination m_d operands. The values for Mp3 applications are 1360 and 742, respectively. These numbers correspond to the concurrency in transaction of source and destination operands that were defined earlier in the section. Again, we find that high concurrency in the operations translates into high values of m_s and m_d. This is to be expected since the number of source and destination operand transactions correspond directly to the number of operations in most cases. The only exception is the store operation that only has destination operand transactions. However, most other operations, particularly arithmetic and logic operations, typically have more source operand transactions than destination operand transactions.

2.2. Mapping Functions and Heuristics. For component allocation, we have derived several heuristic that measure how well the selected components match the given requirements. In order to allocate a register file, we have derived a function that measures how good any of the register files available in Component Library match the application requirement. The function used to evaluate the register files is given by the following formula:

$$H_{rf(x)} = 2 * \operatorname{abs}(x\_in - rq\_in) + \operatorname{abs}(x\_out - rq\_out), \quad (1)$$

where x is the candidate component form the library, and x_in and x_out are the number of input and output ports of the component x. Also rq_in and rq_out are the number of required inputs and outputs, respectively. Required number of outputs correspond to the number of source operands read using *Read Variable* operation, and number of inputs correspond to the number of *Write Variable* operations. The heuristic is chosen to give priority to the input port requirement in order to allow storage of as many results as possible. The value of the function is computed for each candidate register file component from the Component Libry, and the one with the smallest value of the function H_rq is chosen and allocated to the data path. Determining the register file size requires estimation of the maximum number of variables and temporary variables that may occur in the application. While counting local and global variables is straight forwarder task, estimating the temporaries require scheduling and binding to be done for the given data path. Therefore, the register file size determination is out of the scope of this work. For all the practical purposes, we use the rule of thumb: for the Initial Data path, we count the number of all the variables in *ALAP* code and allocate additional 25% registers in the register file.

Allocation of the source and the destination buses depends on the number of source operands and destination operands that the application may require at the same time (in the same cycle). Therefore, the number of source busses equals to m_s and the number of the destination busses equals to m_d that was recorded while traversing the application *ALAP* schedule.

As for the memory interface allocation, we first consider number of sources and destination buses and chose the maximum of them to serve as the required number of ports rq_mi. We compute the value of H_mi for each candidate x component according to

$$H\_mi(x) = x\_in - rq\_mi, \quad (2)$$

where x_in is the number of input ports of memory interface, and rq_mi is a required number of memory interface input ports. The component with the minimum value of the heuristic is selected. In the corner case, where rq_mi is less than any of x_in, the memory interface with the minimum (or maximum) number of ports is chosen. Similarly, in case where rq_mi is less greater than any of x_in, the memory interface with the maximum number of ports is chosen.

Furthermore, in order to select functional units for available operators, we define

(i) $Ops(FU)$ as a set of operations that a functional unit FU performs,

(ii) *Selected* set of selected functional units for the implementation of a final data path,

(iii) n_{FU} the number of instances of the functional unit FU in the *Selected*.

A matching heuristics $H(OP, Ops(FU)) \rightarrow$ *Selected* maps the set of operations OP to a set of functional units *Selected* to implement the custom data path. The heuristics H determines both the type and the number of instances of a functional unit. Algorithm 1 describes the heuristics used in this work. $|\mathrm{Ops}(FU_i)|$ represents the cardinal number of set $\mathrm{Ops}(FU_i)$. According to heuristics H, a functional unit FU_i will be selected to perform an operation op_i if it performs the greatest total number of operands alongside the chosen one. Therefore, this heuristics prioritizes functional units with a higher possibility for sharing. As for the number of instances, the heuristics includes additional units of a chosen type to the set *Selected* only if the maximum number of occurrences of operand op_i is greater than the number of units n_{FU} of that type currently in *Selected*, that is, if $m_{\mathrm{op}} > n_{FUi}$. For example, if application requires 3 units to perform additions and 4 units to perform subtractions, and an ALU is chosen for both addition and subtraction operator, the tool will allocate only 4 instances of the ALU.

To ensure that the interconnect is not a bottleneck in the Initial Data path, the connection resources are allocated in greedy manner. This means that output ports of all register files are connected to all source buses. Similarly, input ports of all register files are connected to all destination busses. The same connection scheme applies to the functional units and the memory interface, making it possible to transfer any source operand to any functional unit or memory interface and the result back into a register file. Note that data forwarding is not presented in this paper and that it is a part of our current research.

```
for all $op_i \in OP$ do
    Select $FU_i$ such that
    $op_i \in Ops(FU_i)$ &&
    $|\text{Ops}(FU_i)| = \mathbf{max}(|Ops(FU_k)|, \forall k$ such that $op_i \in Ops(FU_k))$
    if $m_{op} > n_{FUi}$ then
        Add $[(m_{op} - n_{FUi}) \times FU_i]$ to Selected
    end if
end for
return Selected
```

Algorithm 1: H(OP,Ops(FU)).

2.2.1. Example. The heuristic described in this section was applied to compute the initial estimates of components and their configuration for our application examples. In particular, we computed H_rf values corresponding to each application for all the register files in the component database. We also computed the required number of source and destination buses for each design. The functional unit mapping heuristic was used to select the type and instances of functional units from the component database. The initial design decisions for the given applications are shown in this section.

Left-hand side of Table 3 shows the register file heuristic values (H_rf) for all possible I/O configurations available in the database. The configuration is indicated in the name of the register file type. For example, RF 4×2 refers to a register file with 4 outputs and 2 inputs. The minimum heuristic value for each application is highlighted in bold. In other words, the configuration that produces the lowest H_rf value is selected. Typically, for larger applications with higher available parallelism in computation, we find that the register file with most input and output ports is selected.

The desirable number of source and destination busses exactly matches the number of data transfers: it is 1360 for the source and 742 for the destination busses. However, it may be unreasonable to have a huge number of buses. In most of such cases, design decisions on the number of function units or the configuration of register files may limit the number of source or destination buses. We will discuss such structural dependencies and their impact on bus selection in the following section.

Memory selection and organization is an important issue in processor design. In general, it is possible to have architectures with multiple memories. However, the task of the memory manager becomes quite complicated in such scenario. For the target processor, the memory is not considered to be the part of the datapath itself. Instead, a fixed single-port memory interface is assumed to provide access to all addressable data. Therefore, in this work we have not focused on memory configuration. The available memory interface with one read and one write ports has been selected for all the applications.

The functional units selected for our example applications are listed in the right-hand side of Table 3. The selection is based on the heuristics described earlier. We have mapped all arithmetic and logic operations to ALUs. Multipliers and dividers are expensive components and have been assigned selectively. We also distinguish between signed divider and unsigned divider. All comparison operations have also been mapped to a common comparator. As in the case of buses, some desirable numbers for function unit allocation require very large area. However, as we will shortly see, the allocation is adjusted based on the structural dependencies between the components.

2.3. Structural Restrictions and Dependencies. In the previous sections, we discussed matching of C code structures to architectural components and templates. Since the components must be connected to each other, there exist several structural dependencies between them. As a result, the number of components may not be independently decided. There are several cases in which we must make trade-offs based on structural restrictions and dependencies. We will discuss some of these restrictions and dependencies in this section. Based on the structural restrictions, we will define an initial data path for the application-specific processor corresponding to each C example.

One of the primary bottlenecks is the number of register files and their configuration. If multiple or distributed register files were used, we would need to provide algorithms for mapping of both variables and temporaries to each register file or each partition. This is a significant problem because we must know the variables and all temporaries in the generated code and provide binding rules for them. Furthermore, we must have the scheduling information for the operations. Scheduling, binding, and mapping are complex problems that are out of the scope of this work. Indeed, one of the fundamental assumption of our processor design paradigm is the separation of allocation from scheduling and code generation. We make the trade-off in optimal concurrency and restrict ourselves to a single register file. However, concurrency is still available by configuring the number of input and output ports of the register file. Therefore, the restriction stems from our methodology and the computational complexity of data partitioning.

Another restriction that we imposed on the data path template is that it does not support forwarding. Therefore, the source operands may come only from register file output ports and from the *constant* value stored in the microcode itself (i.e., in the control word). It must be noted that this restriction is only to manage the problem size and there does not exist any technical reason for restricting forwarding. Program analysis can give us information for deciding where to insert the forwarding paths. Also, in this work, only one *constant* value is assumed to be stored in any microcode for a single cycle (i.e., in one control word). Hence, the total maximum number of source buses equals the number of register file output ports (+1 in case the number of output ports is odd). In all of the cases of the benchmarks presented here, we have imposed an upper limit on the number of source buses as the number of register file output ports. By the same token, the number of destination buses is at most equal to the number of register file input ports.

We have assumed that the output of the function units is written directly to the register file. Therefore, for the practical purposes, it makes sense to limit the number of instances of any functional unit type to be not more than the number of register file input ports. This decision has been made because in case where all of the units of the same type are utilized at the same time, they need to be able to store their values after executing the operation. Due to the above reason, we also limit the total number of functional units to be not more than the total number of register file input ports. Therefore, we may need to adjust the number of functional unit with the highest number of instances, for a given application. For example, the $Mp3$ application requires 223 ALUs to exploit all the available parallelism. However, since there are total of four other types of functional units, we allocate only four ALUs.

We can conclude from the above arguments that the register file allocation imposes the most stringent constraints on the data path structure. Therefore, it makes sense to start with allocating the register file, followed by all other component allocations. Once all the components are allocated, we may readjust the number of instances of functional units, if needed.

The resulting initial data path for $Mp3$ consists of one register file with 16 outputs and 8 inputs (Rf 16×8), 4 ALUs, 2 Multipliers, 1 Comparator, 1 Divider Unsigned, and 1 Divider Signed. The number of source and destination buses is equal to the number of register file output and input ports, respectively. As it can be seen, because the applications has high level of parallelism, it utilizes the register files with the largest number of ports and the large number of instances of functional units.

3. Data Path Optimization

The initial data path is used for compiling and profiling the given application code. The compiled code is then analyzed: first, the basic blocks (BBs) to be optimized are selected (Section 3.1). The designer may chose from a single BB to all BBs in the code. Nevertheless, the entire application runs on the generated data path. Then, for each selected BB, a usage plot (usage per cycle) for each component is created (Section 3.2). For each specified resource constraint, the estimation algorithm computes the number of extra cycles that the application needs when the number of instances is as specified by the constraint. As multiple resource constraints may be specified, the smallest estimated number of extra cycles is selected to be the Timing Overhead (Section 3.3). A subset of possible designs is created and the number of execution cycles is estimated for each design [6]. The optimization goal is to determine the number of unconstrained components and the structure of the data path such that the execution is extended by no more than the Timing Overhead (Section 3.4).

3.1. Critical Code Extraction

3.1. Critical Code Extraction. We define critical code as the set of BBs in the source code that contributes the most to the execution time and have the largest potential for optimization. Our selection criterion is based on the relative size and execution frequencies of the BBs in the application. Large BBs have high potential for optimization, since they have several operations that may potentially be performed in parallel. On the other hand, even minor optimization of BBs that have high frequency will yield high overall performance gains. Therefore, we keep 3-ordered lists of BBs, sorted by length (l_i), frequency (f_i), and frequency-length product, where i is the block id. For user-defined parameters P_{fl}, P_l, and P_f, a BB is considered critical if it meets any of the below conditions:

$$f_i \cdot l_i \geq P_{fl} \cdot \sum_{j=0}^{N} f_j \cdot l_j, \tag{3}$$

$$l_i \geq P_l \cdot \max_{j=0}^{N} \left(l_j\right), \tag{4}$$

$$f_i \geq P_f \cdot \max_{j=0}^{N} \left(f_j\right). \tag{5}$$

If any of the parameters equals to 100%, the data path will be optimized for the entire code.

3.2. Usage Plot Creation. A usage plot shows cycle-by-cycle usage for each data path resource. One usage plot is created for each selected BB for each component type (in case of functional units and buses), and for data ports of the same type, that is, input or output ports (in case of the storage units). Usage plots are extracted from the schedule generated for the Initial Data path. The usage plot is annotated with the frequency of a corresponding BB. The example of a usage plot for adders is shown in Figure 5. If we assume that the type and number of instances of all other components do not change, we can conclude that we need 3 instances of adder to execute this BB in ≤6 cycles.

3.3. Timing Overhead Computation. Changing the number of instances of a resource may affect the application execution time. If the designer specifies resource constraint for adders to be two (Figure 5), the number of cycles required for execution of the BB would increase. There would be one extra operation in cycle three and in cycle five that could not execute in the originally specified cycle. The estimation algorithm finds a consecutive cycle that has an available adder (such as cycle four and cycle six) and uses it for delayed execution of the extra operation. By modeling the delayed execution the tool estimates, the extra number of cycles, d_c. If this is the only resource constraint specified, *Timing Overhead* equals to d_c (one cycle in Figure 5). For all other unconstrained resources, the number of instances needs to be sufficient so that the given BB executes in maximum seven cycles.

The estimation is done for a single resource type at a time and, therefore, the input is a set of usage plots for that resource for selected basic blocks. The task of estimation step is to quantify the effect of change in the number of instances of the resource to the application execution time. In order to do so, we compute the number of extra cycles (d_c) that is required to execute a single basic block with the desired number of units (*NewNumber*) using "Spill" algorithm (Algorithm 2).

We keep a counter (*DemandCounter*) of operations/data transfers that were originally scheduled for the execution in the current cycle on an instance of the resource *r* but could not possibly be executed in that cycle with the *NewNumber* of instances. For example, in both cycles 3 and 5 (in bottom of the Figure 5), there is one operation (shown in dashed lines) that cannot be executed if only two adders are used. Those operations need to be accounted for by the *DemandCounter*.

In each cycle, we compare the number of instances in use in a current cycle ($X.InUse$) to the *NewNumber*. If the number in the current cycle is greater, the number of "extra" instances is added to the *DemandCounter*, counting the number of operations/transfers that would need to be executed later. On the other hand, if the number in the current cycle is less than the *NewNumber*, there are available resources that may execute the operations/transfers that were previously accounted for with *DemandCounter*. In the bottom of Figure 6, the available resources are shown in yellow and the "postponed" execution of "extra" operations is shown by arrows. The "Spill" algorithm models in this way the delayed execution of all "extra" operations. After going through all cycles in a given block, the *DemandCounter* equals to the number of operations that need to be executed during the additional cycles d_c.

The "Spill" algorithm uses only statically available information and provides the overhead for a single execution of a given basic block. In order to estimate the resulting performance, we incorporate execution frequencies in the estimation. The estimated total execution time equals sum of products of block's d_c and block's frequency for each block selected for the optimization. We must note that this method does not explicitly account for interference while changing the number of instances of other resources than the specified ones.

3.4. Balancing Instances of Unrestricted Components. We assume that it is acceptable to trade off certain percentage of the execution time in order to reduce number of used resources (hence to reduce area and power and increase component utilization). Therefore, we select a subset of all possible designs to be estimated. The designs in the selected subset are created as follows:

(i) set the number of instances for the constrained resources to their specified value,

(ii) set the number of instances of unconstrained resources (functional units, buses, storage elements, and their ports) to the same values as in the Initial Data path,

(iii) assume the same connectivity as in the Initial Data path,

(iv) randomly select a resource type and vary its number of instances.

For the example depicted in Figure 6, where a multiplier is selected to have its number of instances varied, there will be two candidate designs created: with one and with two multipliers. The candidate design with no multipliers would not make sense, since there will be no units to perform the operations that were originally performed by the multiplier (Algorithm 1). Also, there is no need to consider three multipliers, since two already satisfy the constraints. It may happen that even if we increase the number of instances of some component, the *Timing Overhead* cannot be met. The algorithm then backtracks to the minimal number that causes violation and reports the *"best effort"* design. In the simple case, shown in the Figure 6, if the Timing Overhead is 1 cycle, having only 1 unit results in the acceptable overhead.

4. Experimental Setup

We implemented the IDp Extraction and the Data path Optimization in C++. We used programmable controller unit, NISC compiler [2, 3] to generate schedule and Verilog generator [7, 8] for translating architecture description from ADL to Verilog. For synthesis and simulation of the designs, we used Xilinx ISE 8.1i and ModelSim SE 6.2 g running on a 3 GHz Intel Pentium 4 machine. The target implementation device was a Xilinx Virtex II FPGA xc2v2000 in FF896 package with speed grade −6. The benchmarks used were *bdist2* (from MPEG2 encoder), *Sort* (implementing bubble sort), *dct32* (from MP3 decoder), *Mp3* (decoder), and *inv/forw 4* × *4* (functions *inverse4* × *4* and *forward4* × *4* are amongst top five most frequently executed functions from H.264). The functions and programs vary in size from 100 lines of C code to over 10000 lines (Table 4). Function *bdist2* is part of the MPEG2 encoder algorithm. *Sort* is a bubble sort implementation that is widely used in various applications. Function *dct32* is a fixed-point implementation of a discrete cosine transform. The *Mp3* decoder is the example for decoding mp3 frames into pulse-code modulated (PCM) data for speaker output. It also contains the *dct32* as one of the functions. Finally, *inv/forw 4* × *4* is a forward and inverse integer transform that is used in H.264 decoder. Profiling information was obtained manually.

Table 4 shows input parameters, benchmark length, and generation time. Parameters P_l, P_f, P_{fl} are defined in

```
in: $Usage\ Plot(UP)\ for\ a\ Resource\ r$
in: $Number\ of\ Instances: NewNum$
out: $d_c$//in number of cycles
for all $X = cycle \in UP$ do
    $CycleBudget = NewNumber - X.InUse$;
    if $CycleBudget \geq 0 \&\& DemandCounter \geq 0$ then
        $CanFit = \mathbf{min}(CycleBudget, DemandCounter)$
        $DemandCounter+ = CanFit$
    else
        $DemandCounter+ = CycleBudget$
    end if
end for
$d_c = \lceil DemandCounter/NewNumber \rceil$
return $d_c$
```

Algorithm 2: Spill.

Section 3.1. We decided on parameter values using the profiling information. The selected values of parameters ensure that blocks that affect the execution time the most are selected for optimization. The following column shows the number of lines of the C code (LoC). The largest C code has 13,898 lines of code, proving the ability of the proposed approach to handle large industrial scale applications. The last two columns present average generation time for nonpipelined and pipelined designs. Even for industrial size application, generation time is less than one minute.

In this paper, we present three sets of experiments. The first set of experiments illustrates design space exploration using the automatic extraction of data path from application C code. The second set of experiments compares our selection algorithm to manual selection of components from C code. The last set of experiments compare the presented extraction technique to HLS and manual design in order to establish quality of generated designs.

5. Results: Interactive Design Exploration

Results of the exploration experiments are shown in Figures 8, 9, 10, 11, 12, and 13. Used baseline data path architectures are MIPS-style manual designs (pipelined and nonpipelined) [5] with an ALU, a multiplier, two source and one destination bus, and a 128-entry register file with one input and two output ports. Only for the *Mp3* application, we have added a divider unit to this processor for comparison with the generated data path. In order to perform fair comparison, the size of storage elements has been customized for every application such that the resources (area) are minimized. Also, for comparison with automatically generated pipelined design, the *pipelined version* of manual design was used as a baseline. In-house compiler is used to create schedule for all baseline and generated data paths. This guarantees that the execution time depends on the data path architecture and does not depend on the compiler optimizations.

While exploring different designs for selected applications, we specified the resource constraints on the number of ALUs and number of output and input ports of register file (RF). The tool extracts a data path from the C code such that it complies to the specified constraint, and resulting data paths are named as

(i) *ALU-N*, where $N \in \{1, 2, 3\}$ is the specified number of ALUs,

(ii) *RFOxI*, where $(O, I) \in \{(2, 1), (4, 2), (6, 3), (8, 4)\}$ are the number of output and input ports.

In case of data path denoted by *RFOxI*, *two* resource constraints were used to generate the design: one for the number of output and the other for the number of input ports while all the remaining elements (like functional units, memories, and connectivity) are decided by the tool as described in Section 3.

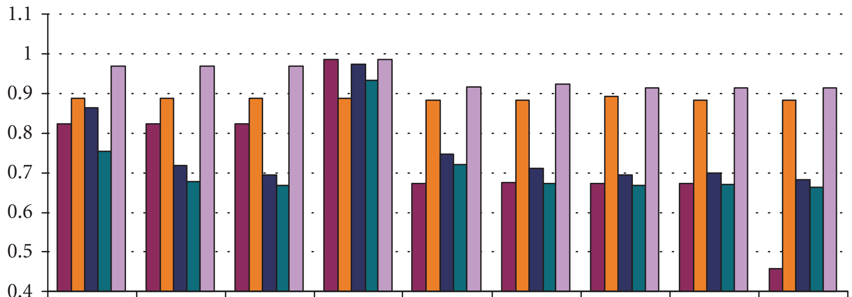
Figures 8 and 9 show the number of execution cycles for generated architectures normalized to the number of cycles for the baseline architecture. These graphs include two additional data paths that are generated only to illustrate tool behavior and to explore theoretical limits of the used data path model (Figure 7). Those additional configurations are

(i) *RF 16 × 8*: a configuration that was generated using RF 16 × 8 constraint,

(ii) *IDp*: an Initial Data path.

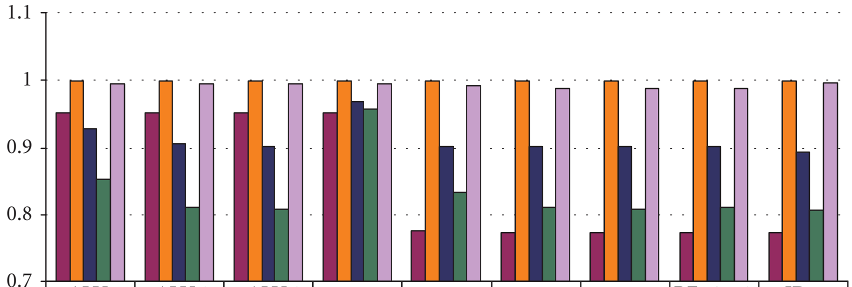
Table 5 summarizes generated architectures for all the configurations that are presented in this paper. Each benchmark and each configuration have a nonpipelined and a pipelined architecture generated, and those are presented in rows marked with N and Y in the column “Pipe.” The table lists the difference from the corresponding baseline architecture. For example, “#R = 64” means that the generated data path has 64 registers in register file, “Rf 4 × 2” means that there is a register file with 4 output and 2 input ports, and “—” means that there is no change to the baseline data path parameters.

For generated nonpipelined data paths (Figure 8), normalized execution cycles range from 0.98 to 0.46. All of the benchmarks experience only a small improvement for RF2 × 1 configuration because this configuration is the most similar



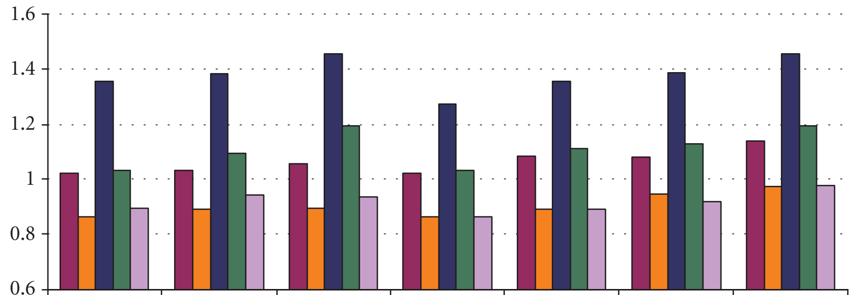
	ALU 1	ALU 2	ALU 3	RF2 × 1	RF4 × 2	RF6 × 3	RF8 × 4	RF16 × 8	IDp
🟥 bdist2	0.82	0.82	0.82	0.98	0.67	0.68	0.67	0.67	0.46
🟧 Sort	0.89	0.89	0.89	0.89	0.88	0.88	0.89	0.88	0.88
🟦 dct32	0.86	0.72	0.7	0.97	0.75	0.71	0.7	0.7	0.68
🟩 Mp3	0.76	0.68	0.67	0.93	0.72	0.67	0.67	0.67	0.67
🟪 inv/forw 4 × 4	0.97	0.97	0.97	0.98	0.92	0.92	0.92	0.92	0.92

Figure 8: Relative number of execution cycles on *nonpipelined* data paths generated for different resource constraints.



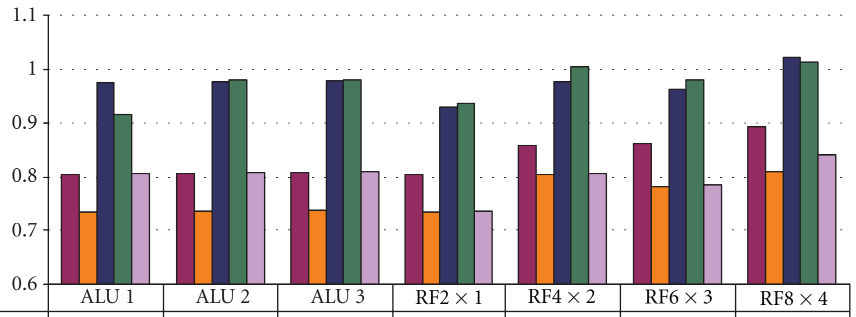
	ALU 1	ALU 2	ALU 3	RF2 × 1	RF4 × 2	RF6 × 3	RF8 × 4	RF16 × 8	IDp
🟥 bdist2	0.95	0.95	0.95	0.95	0.77	0.77	0.77	0.77	0.77
🟧 Sort	1	1	1	1	1	1	1	1	1
🟦 dct32	0.93	0.91	0.9	0.97	0.9	0.9	0.9	0.9	0.89
🟩 Mp3	0.85	0.81	0.81	0.96	0.83	0.81	0.81	0.81	0.81
🟪 inv/forw 4 × 4	1	1	1	0.99	0.99	0.99	0.99	0.99	1

Figure 9: Relative number of execution cycles on *pipelined* data paths generated for different resource constraints.



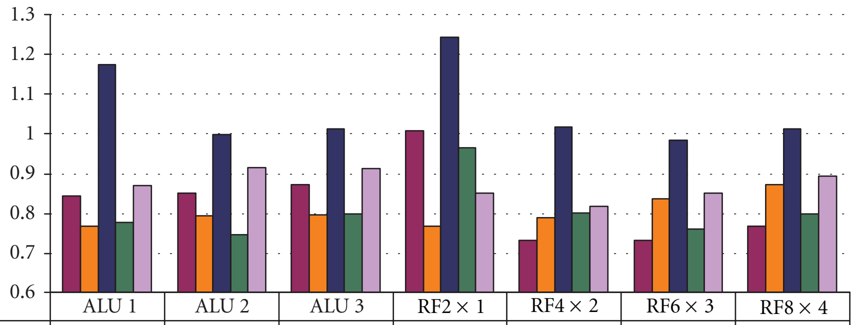
	ALU 1	ALU 2	ALU 3	RF2 × 1	RF4 × 2	RF6 × 3	RF8 × 4
🟥 bdist2	1.03	1.03	1.06	1.03	1.09	1.08	1.14
🟧 Sort	0.87	0.9	0.9	0.87	0.89	0.95	0.98
🟦 dct32	1.36	1.38	1.46	1.28	1.36	1.39	1.46
🟩 Mp3	1.03	1.1	1.2	1.03	1.11	1.13	1.2
🟪 inv/forw 4 × 4	0.9	0.94	0.94	0.87	0.89	0.92	0.98

Figure 10: Relative cycle time for *nonpipelined* data paths generated for different resource constraints.



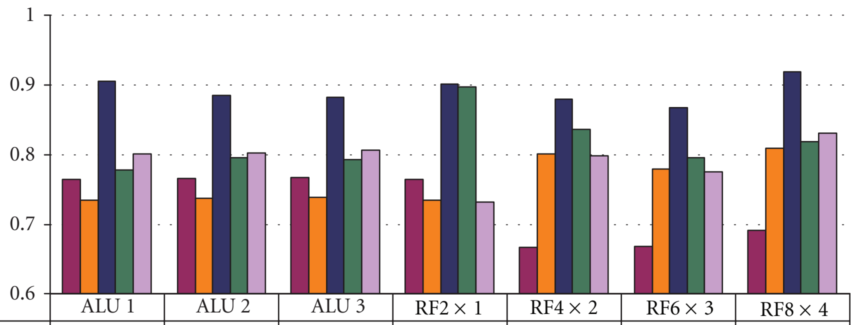
	ALU 1	ALU 2	ALU 3	RF2 × 1	RF4 × 2	RF6 × 3	RF8 × 4
🟥 bdist2	0.8	0.81	0.81	0.8	0.86	0.86	0.89
🟧 Sort	0.74	0.74	0.74	0.74	0.8	0.78	0.81
🟦 dct32	0.98	0.98	0.98	0.93	0.98	0.96	1.02
🟩 Mp3	0.91	0.98	0.98	0.94	1	0.98	1.01
🟪 inv/forw 4 × 4	0.81	0.81	0.81	0.74	0.81	0.79	0.84

FIGURE 11: Relative cycle time for *pipelined* data paths generated for different resource constraints.



	ALU 1	ALU 2	ALU 3	RF2 × 1	RF4 × 2	RF6 × 3	RF8 × 4
🟥 bdist2	0.85	0.85	0.87	1.01	0.73	0.73	0.77
🟧 Sort	0.77	0.79	0.8	0.77	0.79	0.84	0.87
🟦 dct32	1.17	1	1.01	1.24	1.02	0.98	1.01
🟩 Mp3	0.78	0.74	0.8	0.97	0.8	0.76	0.8
🟪 inv/forw 4 × 4	0.87	0.91	0.91	0.85	0.82	0.85	0.89

FIGURE 12: Relative execution time for *nonpipelined* data paths generated for different resource constraints.



	ALU 1	ALU 2	ALU 3	RF2 × 1	RF4 × 2	RF6 × 3	RF8 × 4
🟥 bdist2	0.76	0.77	0.77	0.76	0.66	0.67	0.69
🟧 Sort	0.73	0.74	0.74	0.73	0.8	0.78	0.81
🟦 dct32	0.9	0.88	0.88	0.9	0.88	0.87	0.92
🟩 Mp3	0.78	0.79	0.79	0.9	0.84	0.79	0.82
🟪 inv/forw 4 × 4	0.8	0.8	0.81	0.73	0.8	0.78	0.83

FIGURE 13: Relative execution time for *Pipelined* data paths generated for different resource constraints.

Table 5: Difference in components and parameters between respective baseline and generated design.

Benchmark	Pipe	ALU1	ALU2	ALU3	RF 2 × 1	RF 4 × 2	RF 6 × 3	RF 8 × 4	RF 16 × 8	IDp
bdist2	N	#R = 64	#R = 64	#R = 64	#R = 64	#R = 64	#R = 64	#R = 64	#R = 64	Rf 8 × 4, 3 Alu, 2 Mul
	Y	#R = 32	#R = 32	#R = 32	#R = 32	#R = 32	#R = 32	#R = 32	#R = 32	Rf 8 × 4, 3 Alu, 2 Mul
Sort	N	#R = 16	#R = 16	#R = 16	#R = 16	#R = 16	#R = 16	#R = 16	#R = 16	Rf 6 × 3, 1 Alu
	Y	#R = 16	#R = 16	#R = 16	#R = 16	#R = 16	#R = 16	#R = 16	#R = 16	Rf 6 × 3, 1 Alu
dct32	N	Rf 4 × 2	Rf 6 × 3	Rf 8 × 4	—	2 Alu	3 Alu	3 Alu	3 Alu	Rf 16 × 8, 4 Alu, 2 Mul
	Y	Rf 4 × 2	Rf 4 × 2	Rf 6 × 3	—	2 Alu	2 Alu	2 Alu	3 Alu	Rf 16 × 8, 4 Alu, 2 Mul
Mp3	N	—	Rf 6 × 3	Rf 8 × 4	—	2 Alu	3 Alu	3 Alu	3 Alu	Rf 16 × 8, 4 Alu, 2 Mul
	Y	Rf 4 × 2	Rf 6 × 3	Rf 6 × 3	—	2 Alu	2 Alu	2 Alu	2 Alu	Rf 16 × 8, 4 Alu, 2 Mul
inv/forw 4 × 4	N	#R = 16	#R = 16	#R = 16	#R = 16	#R = 16	#R = 16	#R = 16	#R = 16	Rf 16 × 8, 4 Alu, 1 Mul
	Y	#R = 16	#R = 16	#R = 16	#R = 16	#R = 16	#R = 16	#R = 16	#R = 16	Rf 16 × 8, 4 Alu, 1 Mul

to the baseline. Also, the number of cycles is not exactly the same but slightly reduced. This effect is due to replacing of buses in architecture specification with multiplexers which allows for more efficient handling by the compiler. This effect is particularly emphasized in case of *bdist2*. For this benchmark, the improvements are small (0.82) for all ALU configurations and may be attributed to the effect of explicit multiplexer specification that results in more efficient compiler handling and shorter prologue/epilogue code. We see the further reduction to 0.67 for RF 4 × 2 configuration which exploits the parallelism of operations executed on different units. No further improvement is seen for further increase in the number of register file ports. However, execution on the IDp, which has two ALU and two multiplier units, experiences additional improvement in number of cycles to 0.47.

Benchmark *Sort* is sequential in nature and, therefore, it does not experience significant improvement regardless of the number of ALUs or register file ports that are introduced. Both benchmarks *dct32* and *Mp3* have abundance of available parallelism. The *dct32* benefits the most from having two ALUs (ALU*2*-0.72), and the increase in number of ports or adding more units (in IDp) contributes by only 4% of additional improvement. Similarly, *Mp3* executes in 0.68 of the number of baseline execution cycles for ALU2 configuration. Note that specifying two ALUs as resource constraint for both benchmarks results in an increase in the number of RF ports and buses: since both instances of ALU and one instance of multiplier are significantly utilized, the resulting configurations have RF 6 × 3 and full connectivity scheme. On the other hand, both benchmarks suffer from significant increase of prologue/epilogue code which sets back the savings in number of cycles that are obtained by the "body" of the benchmark. Adding more ALUs does not help in case of benchmark *inv/forw 4 × 4*. The benchmark benefits the most from additional register file ports, because this configuration exposes limited parallelism between the operations that execute on functional units of different type.

As for the pipelined configurations, shown in Figure 9, across all the benchmarks, maximum reduction in the number of execution cycles for generated data paths (0.77) is less than a maximum reduction for the nonpipelined designs since the pipelining itself exploits some degree of available parallelism. In case of *bdist2,* there is no improvement with increased number of ALUs since the tool allocates single RF 2 × 1. Same as for the nonpipelined configurations, the minimal normalized number of cycles is reached for RF 4 × 2 due to the increased simultaneous use of ALU and multiplier. On the other hand, benchmark *Sort* does not change the number of execution cycles since pipelining takes advantage of already limited available parallelism.

For configurations ALU 1, ALU 2, and ALU 3, the tool allocates, together with sufficient connectivity resources:

(i) for *dct32*: RF 4 × 2, RF 4 × 2 and RF 6 × 3, respectively;

(ii) for *Mp3*: RF 4 × 2, RF 6 × 3 and RF 6 × 3, respectively.

The most of the execution cycle reduction is brought by an increase in the number of register file ports in case of configuration ALU1. Increasing both number of ALUs and ports brings down the normalized cycles only by 0.02 and 0.04 down to 0.90 and 0.81 for *dct32* and *Mp3*, respectively. For all the RF configurations, both benchmarks have the same trend for allocation: the tool recognizes a potential for adding more ALUs and, therefore, two ALUs are allocated for all of them, except for RF 16 × 8 configuration of *dct32* where three ALUs are allocated. One would expect the tool would allocate more units for RF 8 × 4 and RF 16 × 8. However, the data dependencies limit concurrent usage of more units than allocated. The results for *IDp* illustrate this: even though *IDp* has three ALUs and two multipliers, further reduction in the number of normalized cycles is only by 0.01 to 0.02 for *dct32* and *Mp3*, respectively. Similarly to *Sort*, *inv/forw 4 × 4* has almost no improvement if the number of instances of increases.

Figures 10 and 11 show normalized cycle time for nonpipelined and pipelined automatically generated designs. We observed the cycle time in order to explain the total execution time of each benchmark. Current version of the tool does not take into account postsynthesis results. However, we believe that this feature is crucial for DFM and are currently working on incorporating prelayout information in data path optimization.

Results for normalized cycle time for designs are intuitive: as complexity of generated data path increases, so does the cycle time. For nonpipelined designs (Figure 10), designs for all benchmarks except *Sort* have larger cycle time than for corresponding baseline. The main contributor to cycle time length is register file: as the number of ports increase, the decoding logic increases and so does the cycle time. In case of *Sort,* cycle time is lower because of the reduction of the register file size. For nonpipelined configurations, the normalized cycle time ranges from 0.85 to 1.46. Pipelined configurations (Figure 11) uniformly have smaller or equal cycle time as the baseline configuration. For each benchmark, there is almost no difference in cycle time across all ALU configurations. *Mp3* is the only benchmark that has significantly lower normalized cycle time for ALU1 configuration (0.91) than for the remaining two ALU configurations (0.98). RF configurations experience the increase in cycle time with an increase in complexity. For RF 6 × 3 in case of *Sort*, *dct32*, *Mp3,* and *inv/forw 4 × 4,* there is a small decrease in cycle time comparing to RF 4 × 2 configurations because the synthesis too manages to decrease combinational delay of interconnect.

Figures 12 and 13 show normalized total execution time. Across all configurations and all benchmarks, except all nonpipelined configurations for *dct32*, total execution time has been reduced. Nonpipelined *dct32* experiences increase in execution time for all but ALU2 configuration: the reduction in number of cycles is not sufficient to offset the large increase in the cycle time. The reduction in number of cycles is less than expected because of explosion of prologue/epilogue code. The nonpipelined configurations reduce the execution time up to 0.73, 0.77, 1.00, 0.74, and 0.82 for *bdist2*, *Sort*, *dct32*, *Mp3,* and *inv/forw 4 × 4*, respectively. Normalized execution times for all nonpipelined configurations, except for the *Sort*, are greater than the corresponding normalized number of cycles. The *Sort* has further decrease in execution time due to significant cycle time reduction (resulting from "minimized" data path comparing to the baseline). Furthermore, for *dct32* and *Mp3*, that perform the best for ALU2, several other configurations have minimum normalized number of cycles. Pipelined configurations uniformly experience smaller normalized execution time compared to the nonpipelined. The minimums are 0.66, 0.73, 0.88, 0.79, and 0.73 for *bdist2*, *Sort*, *dct32*, *Mp3,* and *inv/forw 4 × 4*, respectively. For all applications, each normalized execution time is smaller than the corresponding normalized number of execution cycles. Furthermore, the configurations that perform in minimal time are the same as the one that performs in minimal number of cycles.

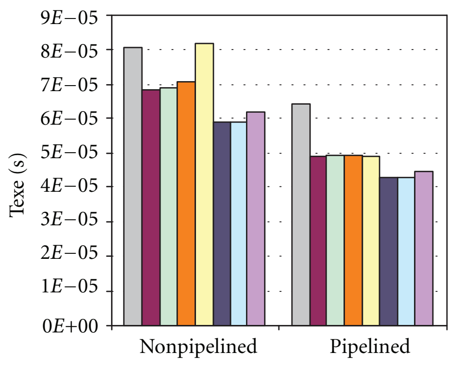
In order to find a data path with a minimum execution time and the best configuration, we plot for each benchmark absolute execution time in Figure 14. The leftmost bar shows the execution time on a baseline architecture. The best implementation for *bdist2* is pipelined RF 4 × 2. RF 6 × 3 has only slightly longer execution time, but since it uses more resources, it is a less desirable (recommendable) choice. Benchmark *Sort* benefits from reduction of resources and, therefore, the best configuration is ALU1. For this benchmark, all of the pipelined configurations perform worse than corresponding nonpipelined. Benchmark *dct32*, despite having plethora of available parallelism, performs good only for nonpipelined *Baseline*, ALU2, and RF 6 × 3 configurations. The pipelined configurations do not perform as well as nonpipelined. To improve the current generated pipelined architectures, we may consider use of multicycle and pipelined functional units which may reduce the cycle time. Furthermore, if there is only single function to be performed on the generated hardware module, both prologue and epilogue code may be eliminated and the speedup of "parallel" architectures would increase. Here, we presented the results for all the applications with prologue/epilogue code because we believe that the application execution needs to have data received and sent proceeding and following the execution of the benchmark body. Therefore, benchmark is a function that needs to be called and, therefore, the prologue and epilogue codes are required. In this case, the number of registers that need to be stored and restored, and hence the length of prologue and epilogue code, needs to be estimated. Nonpipelined designs for *Mp3* perform better than pipelined, for the same reason. Overall, the best design would be for ALU2 configuration with 32% performance improvement over the baseline. Similarly, benchmark *inv/forw 4 × 4* has smaller execution time for all nonpipelined configurations than for the pipelined ones. The peak performance improvement, 37%, is achieved for RF 4 × 2 configuration.

6. Results: Selection Algorithm Quality

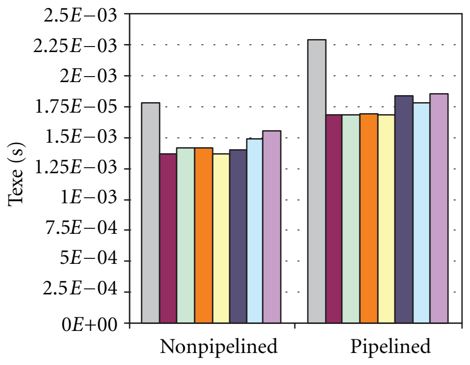
Table 6 shows the comparison of manually designed architectures and the automatically generated ones. The manual designs were created by computer engineering graduate students. The students were asked to select the components for the templatized data path, as the one in Figure 7, based on the application C code. Running and profiling the code on the host machine with the same input as used for the automatic generation data were allowed.

There is the only one column for manual designs in Table 6 because the designers had *the same* component/parameter selection for nonpipelined and for pipelined data paths. However, our experiments in Section 5 show that often there is less resources required for pipelined configurations. Such examples are configurations ALU2, ALU3, RF 6 × 3, and RF 8 × 4 for *dct32* in Table 5. The nonpipelined and pipelined configurations presented in Table 6 are those that have the smallest number of execution cycles for the given benchmark, as seen in Figures 8 and 9.

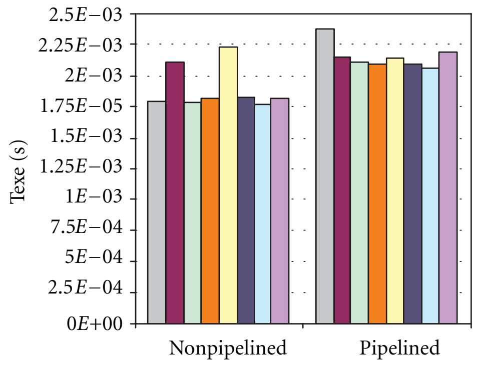
It is interesting to notice that for manual designs, in most cases, the number of instances and parameters of selected register files and functional units outnumbers the one in the best generated architectures. For example, for benchmark *bdist2,* manual designer anticipated use of four ALUs. The nonpipelined IDp for benchmark *bdist2* needs only three and pipelined IDp only one ALU, which shows that the designer overestimates the number of ALUs. Also, the optimal automatically generated data path uses only one ALU in both nonpipelined and pipelined cases. However, for this benchmark, the designer underestimated register file



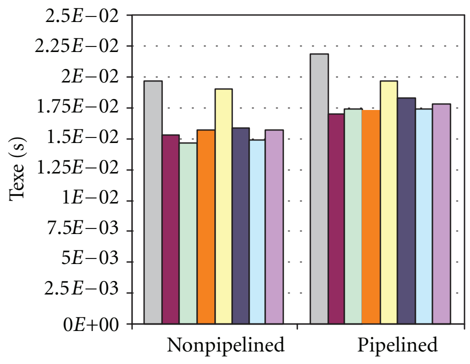
(a) *bdist2*



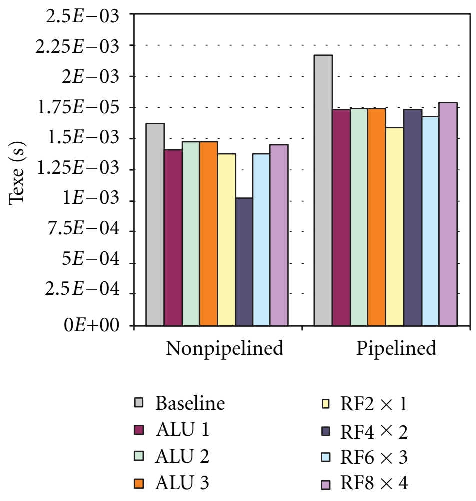
(b) Sort



(c) *dct32*



(d) Mp3



(e) *inv/forw 4 × 4*

Figure 14: Total execution time on generated data paths.

size: in case where there are more functional units, more operations may be performed in parallel and, therefore, there will be more operands/registers required. The tendency to allocate manually more resources than actually required may be explain the best on the example of function *inverse4* × *4* from *H.264*, shown in Algorithm 3.

The designer allocates four ALUs based on an observation that code in lines 7, 8, 9, and 10 is not dependent and an assumption that once all of the operations from line 2 to line 5 are completed, the entire block of lines 7 to 10 will be executed at the same time. However, code in lines 2 to 5 has data dependencies, requires sequential execution, and performs memory access. Therefore, it makes sense to compute expressions in line 7 and line 8 as soon as t0 and t2 are available. Hence, no need for 4 ALU in the data path.

Similarly, for *dct32*, data dependencies are not "visible" from C code. Therefore, the designer allocates four ALUs, two multipliers, a comparator, and two adders. IDp for *dct32* has only three ALUs, two multipliers, and a comparator even though it has a register file RF 16 × 8. IDp configuration

Table 6: Comparison between components and parameters of manual and automatically generated design.

Benchmark	Manual	Automatic	
		Nonpipe	Pipe
bdist2	#R = 32, Rf 8 × 4, 4 Alu, 1 Mul	#R = 64, Rf 4 × 2, 1 Alu, 1 Mul, 1 Comp	#R = 32, Rf 4 × 2, 1 Alu, 1 Mul, 1 Comp
Sort	#R = 32, Rf 4 × 2, 1 Alu	#R = 32, Rf 2 × 1, 1 Alu, 1 Mul, 1 Comp	#R = 32, Rf 2 × 1, 1 Alu, 1 Mul, 1 Comp
dct32	#R = 48, Rf 8 × 4, 4 Alu, 2 Mul, 1 Comp, 2 Adders	#R = 128, Rf 8 × 4, 3 Alu, 1 Mul, 1 Comp	#R = 128, Rf 4 × 2, 2 Alu, 1 Mul, 1 Comp
Mp3	#R > 16, Rf 16 × 8, 4 Alu, 8 Mul 1 Or, 1 Comp, 1 NotEq Comp, 1 Div	#R = 128, Rf 8 × 4, 3 Alu, 1 Mul, 1 Comp, 1 Div	#R = 128, Rf 2 × 1, 1 Alu, 1 Mul, 1 Comp, 1 Div
inv/forw 4 × 4	#R = 32, Rf 8 × 4, 4 Alu, 1 Comp	#R = 16, Rf 4 × 2, 1 Alu, 1, Mul, 1 Comp	#R = 16, Rf 2 × 1, 1 Alu, 1 Mul, 1 Comp

```
...
t0 = *(pblock++);
t1 = *(pblock++);
t2 = *(pblock++);
t3 = *(pblock);

p0 = t0 + t2;
p1 = t0 − t2;
p2 = SHIFT(t1, 1) − t3;
p3 = t1 + SHIFT(t3, 1);
 ...
```

Algorithm 3: Part of *inverse4 × 4* C code.

performs in 0.68 of the baseline, which is only 2% better than configurations *ALU3*, RF 8 × 4, and RF 16 × 8 that have less resources. The number of registers in the register file is computed based on the number of units (eight without the comparator), the fact that each unit has two inputs and one output, and assumption that for each source/destination the data memory will be used twice. Therefore,

$$\#R = 8 \times (2 + 1) \times 2 = 48, \tag{6}$$

that is, the designer decides on 48 registers. Practically, with these many units, there are more than 48 registers required for temporary variables, if we want to avoid access to memory for fetching data and storing results.

Manual selection of components and parameters for *Mp3* shows the same properties: the number of functional units was overestimated, the number of registers in the register file was underestimated, and the pipelining was selected after the decision on units had been made. The designer profiled the application and found that among all computationally intensive functions, function *synth_full* contributes 35% to total execution. The designer identified eight multiplications and four additions that may be executed in parallel in this function. Also, only the lower bound for the number of registers in the register file was specified.

In order to better understand cost/performance trade-off for manually and automatically generated data paths, we defined a total cost of a design C_{design} as a sum of slices and a sum of RAMs for all the selected components:

$$C_{\text{design}} = \sum_{\text{components}} \text{slice} + \sum_{\text{components}} \text{RAM}. \tag{7}$$

We synthesized all available components and assigned cost in terms of slices and RAMs. We generated both nonpipelined and pipelined versions of manual design, so that we can perform fair comparison of cost and performance. We assumed that when pipelining was selected all inputs and outputs of functional units have a pipeline register (uniform pipelining, such as shown in Figure 7). Cost of pipeline registers was added to the total cost of pipelined data path designs. The performance is measured in the number of execution cycles, since neither the designers nor the tool were given any synthesis information as an input.

Figures 15, 16, 17, 18, and 19 show set of cost/performance graphs for all presented benchmarks. All automatically generated designs have significantly lower cost compared to the ones manually derived from C code. Relative to the corresponding manual designs, the automatically generated ones have from 7.41% (*dct32* nonpipelined) to 82.13% (pipelined *inv/forw 4 × 4*) less slices and from 8.22% (nonpipelined *bdist2*) to 87.46% (pipelined *Mp3*) less RAMs. As seen in the performance graphs, overhead in the number of cycles is negligible for all designs except for the above-mentioned pipelined *Mp3*. *Mp3* has 18.84% overhead, but 81.50% and 87.46% less slices and RAMs, which according to us is a reasonable trade-off. For all the remaining designs, overhead of the number of cycles ranges from 0.00% to 0.68% relative to the corresponding manual design.

This experiment showed that translating C code into simple hardware design is a nontrivial process. Selecting components and their parameters, tracking data and control dependencies, and estimating operation sequencing and available parallelism based on high level description results in underutilization of data path elements. On the other hand, using our tool, within the same time, a designer may explore many different alternatives and create working solution that satisfies his or her needs.

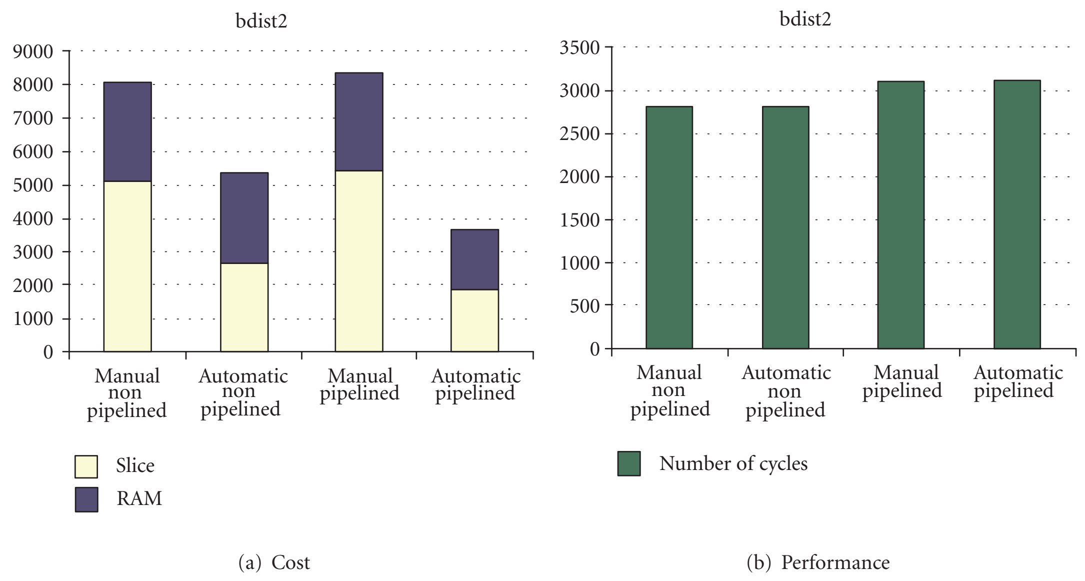


FIGURE 15: *bdist2*: number of slices, number of RAMs, and number of cycles for manually selected and automatically generated data paths.

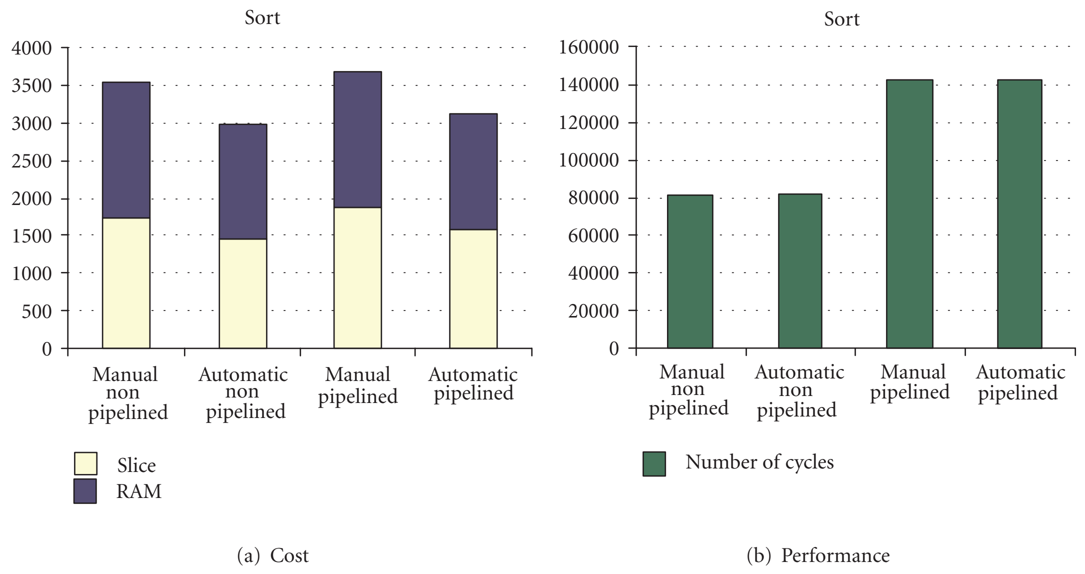


FIGURE 16: Sort: number of slices, number of RAMs, and number of cycles for manually selected and automatically generated data paths.

7. Results: Design Refinement Quality

In this section, we present design refinement quality for two applications *dct32* and *Mp3* and compare automatically generated data paths to implementations using HLS tool and MicroBlaze soft processor.

7.1. Dct32. Figure 20 plots values for the number of cycles (No.cycle), clock cycle time (Tclk), and total execution time (Texe), while Figure 21 plots values for number of slices and bRAMs (Slice and BRAM) for several different data paths for *dct32* benchmark. All the values have been normalized to the corresponding values of a *manually* designed data path for the same application. Note that the same C code has been used as a starting point for all designs, including manual. The graphs show following data paths:

(i) *Baseline*—corresponds to a pipelined version of a baseline design for the *dct32* used in 5,

(ii) *ALU1-N and RF 4 × 2-N*—generated *nonpipelined* data paths for constraints ALU 1 and RF 4 × 2, respectively,

(iii) *RF 4 × 2-P*—generated *pipelined* data path for constraint RF 4 × 2,

(iv) *HLS*—a design generated by academic high level synthesis tool [9],

(v) *MicroBlaze*—a design implemented on soft processor MicroBlaze [10].

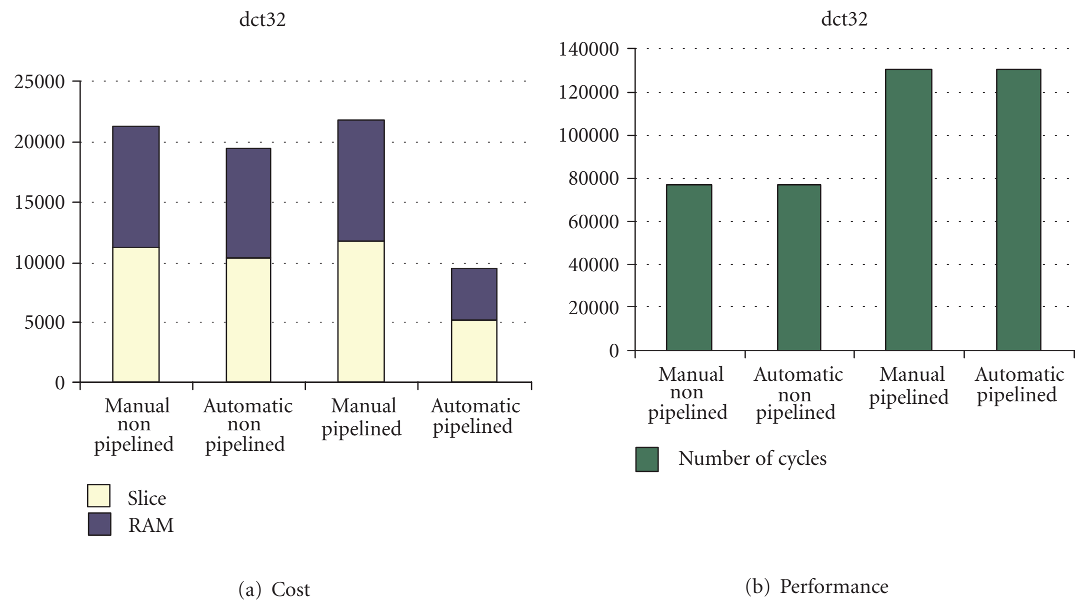


FIGURE 17: *dct32*: number of slices, number of RAMs, and number of cycles for manually selected and automatically generated data paths.

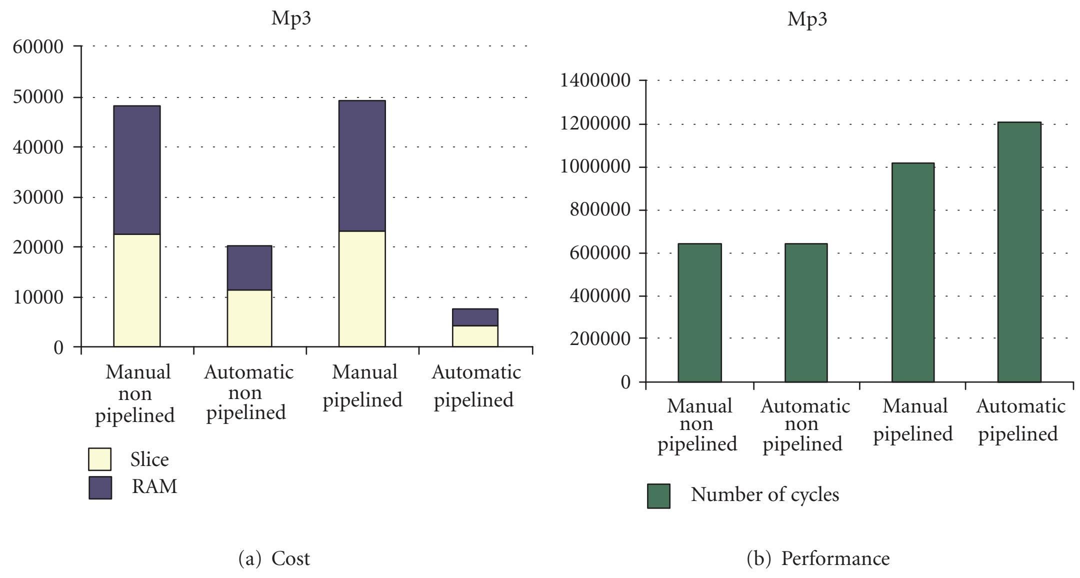
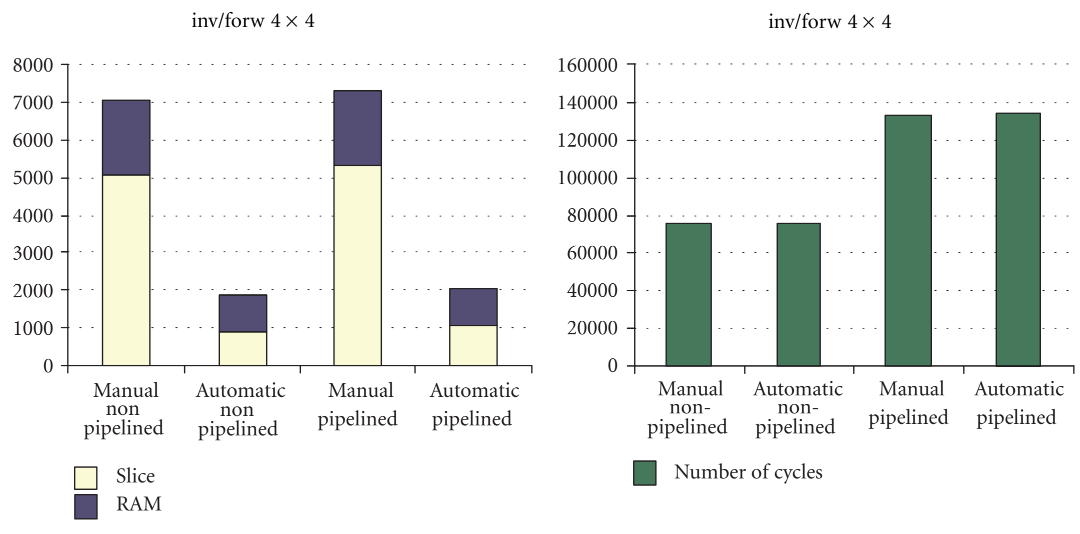


FIGURE 18: Mp3: number of slices, number of RAMs, and number of cycles for manually selected and automatically generated data paths.

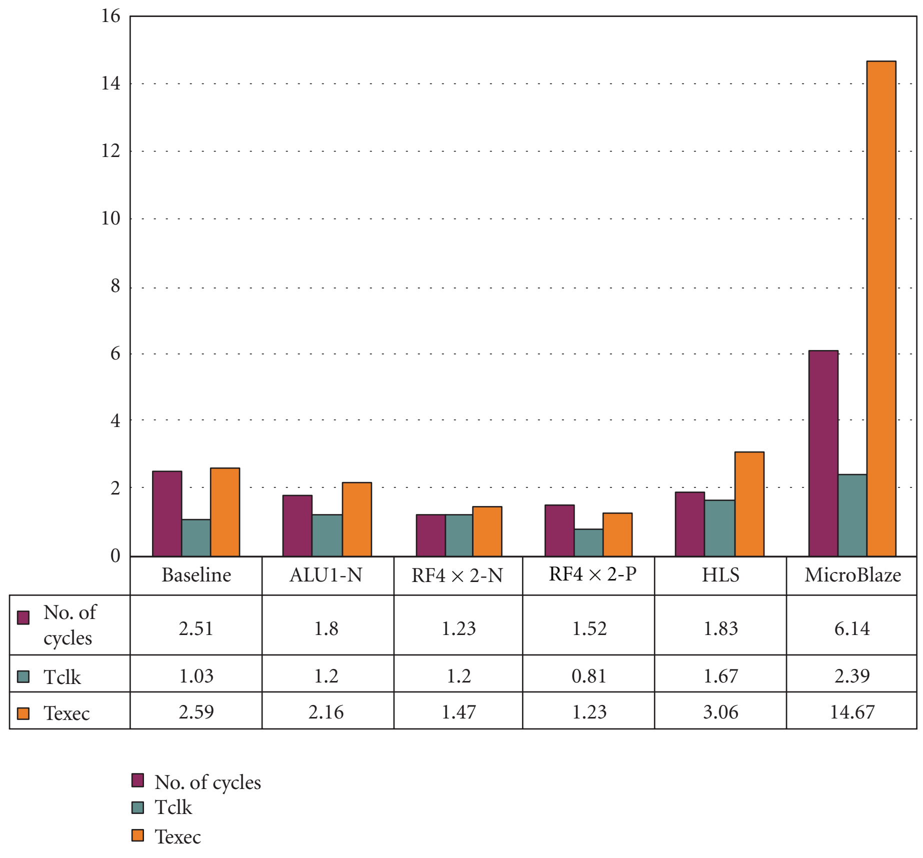
To alleviate different assumptions of different tools and designers for wrapping the function by send/receive primitives, we present here the results for the body of the *dct32* function, contrary to experiments in 5. The manual implementation has been designed by third-party RTL designer [11]. It is important to notice that the largest normalized value across all performance metrics for the *Baseline* and all the generated designs is 2.59 times the corresponding metric of the manual design, where HLS and MicroBlaze reach 3.06 and 14.67, respectively. Hence, for the generated designs, none of the compared metrics are several orders of magnitude larger than the manual design. The overhead of number of cycles for generated designs range from 23% (i.e 1.23 on the graph) to 80% of the manual design, while cycle time experiences from 25% (0.85 in the figure) speedup to 25% (1.25 in the figure) slowdown. The best generated design RF 4 × 2-*P* has 1.23 times longer execution time comparing to the manual.

Baseline and all generated architectures have from 0.53 to 0.64 times slices and 2.29 times block RAMs (bRAMs) compared to the manual design. This is because the tool attempts to map all storage elements to bRAM on FPGA. On the other hand, the design generated by HLS tool uses 1.36 times slices and only 0.14 times bRAMs due to the heavy use of registers and multiplexers. The generated designs outperform the design produced by HLS tool with



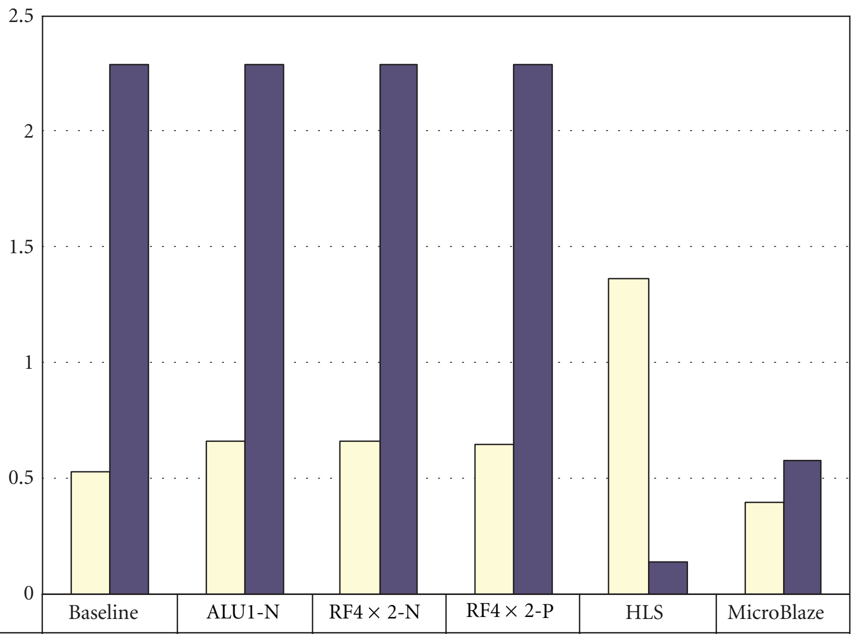
(a) Cost (b) Performance

FIGURE 19: *inv/forw 4 × 4*: number of slices, number of RAMs, and number of cycles for manually selected and automatically generated data paths.



	Baseline	ALU1-N	RF4 × 2-N	RF4 × 2-P	HLS	MicroBlaze
■ No. of cycles	2.51	1.8	1.23	1.52	1.83	6.14
■ Tclk	1.03	1.2	1.2	0.81	1.67	2.39
■ Texec	2.59	2.16	1.47	1.23	3.06	14.67

FIGURE 20: Performance comparison for *dct32* relative to manual implementation.



	Baseline	ALU1-N	RF4 × 2-N	RF4 × 2-P	HLS	MicroBlaze
□ Slice	0.53	0.66	0.66	0.64	1.36	0.4
■ BRAM	2.29	2.29	2.29	2.29	0.14	0.57

Figure 21: Area comparison for *dct32* relative to manual implementation.

respect to all the metrics except the number of used bRAMs. Moreover, the average generation time for *dct32* is 2.3 seconds while it took 3 man-weeks for the manual design. The fastest extracted design has only 23% of execution overhead and a negligible generation time compared to the manual design. Hence, we believe that the proposed data path extraction from C code is valuable technique for creation of an application-specific data path design. Moreover, all the generated designs outperform MicroBlaze: the best automatically generated design RF 4 × 2-*P* has 1.23 times longer execution time, where MicroBlaze has 14.67 times longer one. However, MicroBlaze utilizes smaller area: 0.4 slices and 0.57 bRAMSs, where our best design utilizes 0.64 slices ad 2.29 bRAMs.

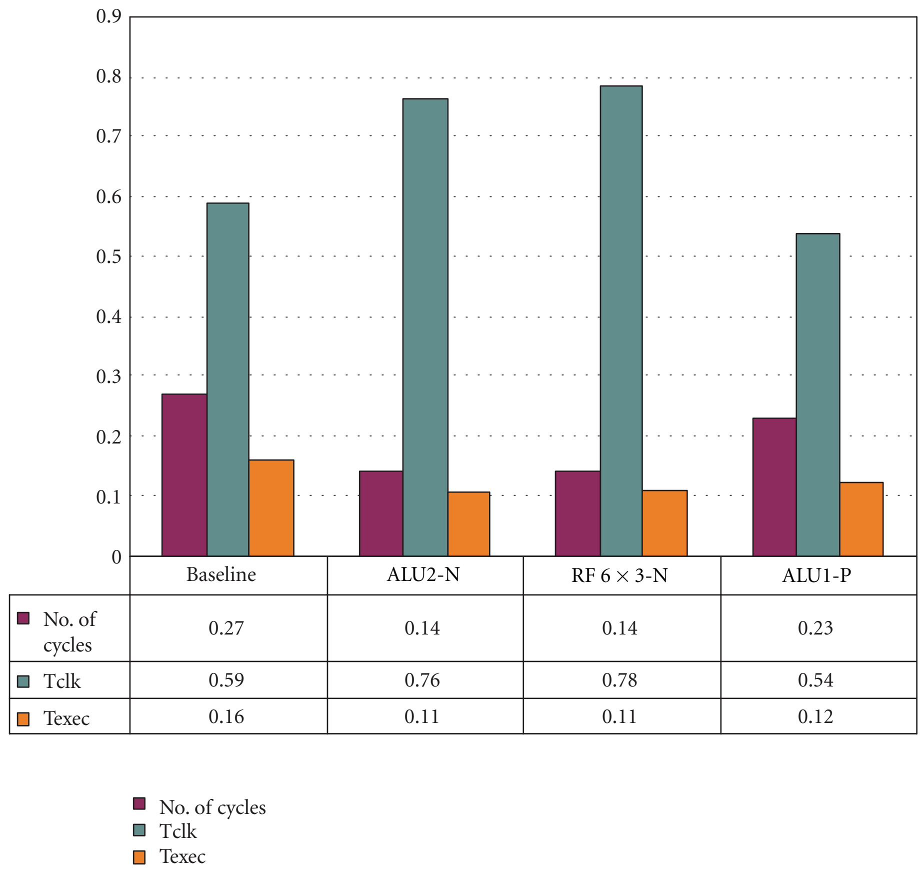
7.2. Mp3. Figures 22 and 23 plot the same performance and area metrics as the previous two figures, but relative to the implementation on *MicroBlaze*, because manual implementation for *Mp3* is not available. Also, the HLS tool could not be used, due to the capacity issues, hence we present the results for all the remaining data path designs. We, also, used the the same C code as an input to all tools. The graphs show following data paths:

(i) *Baseline*—corresponds to a pipelined version of a baseline design for the *dct32* used in Section 5,

(ii) *ALU2-N*—generated *nonpipelined* data paths for constraints ALU 2,

(iii) *RF 6 × 3-P and ALU1-P*—generated *pipelined* data path for constraint RF 4 × 2 and ALU 1, respectively.

With respect to performance, all the designs outperform the MicroBlaze implementation. The number of cycles improvements range from 73% to 88% (i.e., 0.27 to 0.12 on the graph), and the cycle time improvement form 24% to 46% (0.76 to 0.54). This significant improvement in both number of execution cycles and the cycle time directly translates to significant savings in total execution time. The best execution time is only a fraction on execution time on MicroBlaze (0.11, i.e., 89% improvement) and it is achieved while running on RF 4 × 2-*N* and RF 4 × 2-*P* data path configurations. The significant performance improvements are because of the use of extra resources: register file with more input/output ports and more functional units that facilitate efficient use of available parallelism. This directly translates into the use of more slices: from 3.53 to 6.12 times more than in the MicroBlaze implementation. Similarly to *dct32*, the tool maps all the storage elements to bRAMs on FPGA, and hence the high bRAM overhead compared to the MicroBlaze implementation.



	Baseline	ALU2-N	RF 6 × 3-N	ALU1-P
🟪 No. of cycles	0.27	0.14	0.14	0.23
🟩 Tclk	0.59	0.76	0.78	0.54
🟧 Texec	0.16	0.11	0.11	0.12

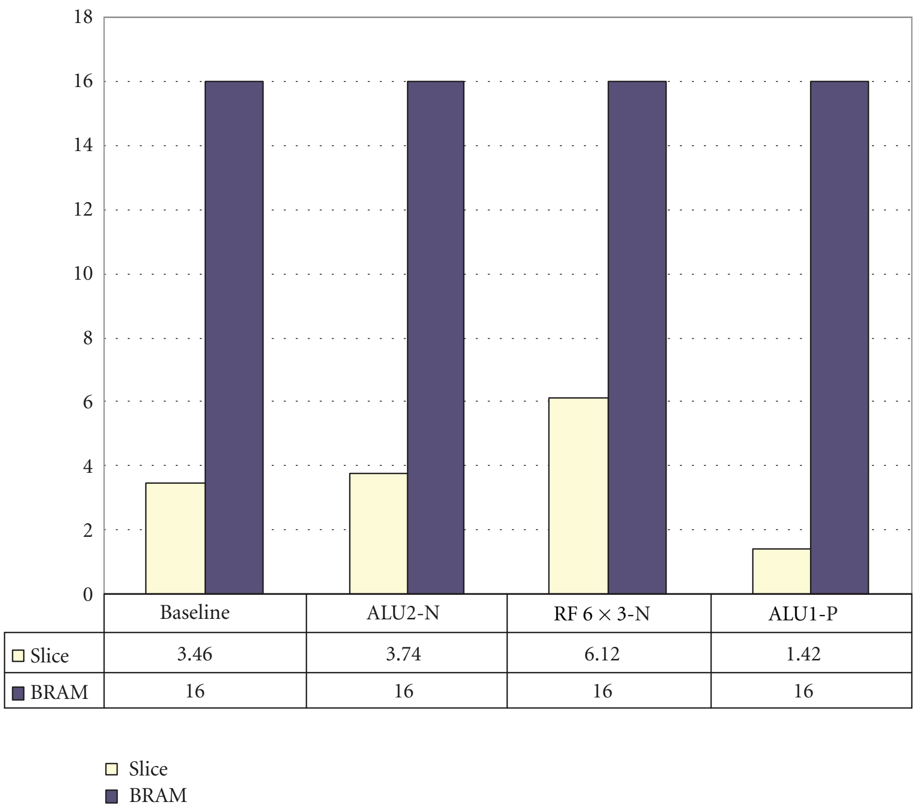
Figure 22: Performance comparison for *Mp3* relative to implementation on MicroBlaze.

8. Related Work

In order to accomplish performance goals, ASIPs and IS extension use configurable and extensible processors. One such processor is Xtensa [12] that allows the designer to configure features like memories, external buses, protocols, and commonly used peripherals [13, 14]. Xtensa also allows the designer to specify a set of instruction set extensions, hardware for which is incorporated within the processor. Our work presents algorithms that generate custom data path, which determines a set of (micro-) operations that can be preformed. This approach is orthogonal to ASIP/IS extension approach. The automatically generated data path can be used in ASIP or added to a processor, as custom hardware unit. In order to do so, one would need to identify the instructions that are to be executed on such a data path. The automated extraction of instructions from a set of operations is a topic of our future work.

The Tensilica XPRES (Xtensa PRocessor Extension Synthesis) Compiler [15] automatically generates extensions that are formed from the existing instructions in style of VLIW, vector, fused operations, or combination of those. Therefore, automated customizations are possible only within bound of those combinations of existing instructions. IS extensions also require the decoder modifications in order to incorporate new instructions. For example, having VLIW-style (parallel) instructions requires multiple parallel decoders [15], which not only increase hardware cost (that may affect the cycle time) but also limit the possible number of instructions that may be executed in parallel. Our approach also automatically generates data path which defines a set of possible (micro-) operations. However, in our approach, the decoding stage has been removed. Therefore, there is no increase in hardware complexity and no limitations on the number and type of operations to be executed in parallel. In case where the code size exceeds the size of on-chip memory, due to the number of operations that are specified to be executed in parallel, "instruction" caches and compression techniques may be employed, both of them have been in scope of our current research.

The IS extensions, in case of Stretch processor [16], are implemented using configurable Xtensa processor and Instruction Set Extension Fabric (ISEF). The designer is responsible for, using available tools, identifying the critical portion of the code "hot spot" and rewriting the code so the "hot spot" is isolated into the custom instruction. The custom instruction is then implemented in ISEF. Thus, the application code needs to be modified which



	Baseline	ALU2-N	RF 6 × 3-N	ALU1-P
☐ Slice	3.46	3.74	6.12	1.42
■ BRAM	16	16	16	16

Figure 23: Area comparison for *Mp3* relative to implementation on MicroBlaze.

requires expertise and potentially more functional testing. The designer is expected to explicitly allocate the extension registers. In contrary, our approach allows but does not require C code modifications and does not require the designer to manipulate the underlying hardware directly. In both previous cases, it is required that the designer has expertise in both software and hardware engineering.

On the other hand, C-to-RTL tools, such as Catapult Synthesis [17], Behaviour Synthesizer Cyber [18], and Cynthesizer [19] generate the data path and the controller simultaneously, which may lead to capacity issues, like the one in the case of GSM algorithm [20]. Catapult [17] allows control over resource sharing, loop unrolling, and loop pipelining. It also provides technology-specific libraries [21] that allow specific hardware instances to be inferred from C code. However, this requires code modifications. As reported by Mentor Graphics, code modifications took one week while the synthesis took one day. Also, the biggest listed C code had 480 lines of code. Other examples published in [20] include W-CDMA 3G modem algorithm called EPC and 2D graphics acceleration algorithm IDCT (which has the same complexity as *dct32* used here). Unfortunately, no number of lines of code was reported. Behavior Synthesizer Cyber [18], in addition to the abovementioned, provides various knobs for fine tuning, such as multiple clocks, gated clocks, synchronous/asynchronous resert, and synchronous/asynchronous/pipelined memory. The C code is extended to describe hardware by adding support for bit-length and in-out declarations; synchronization, clocking and concurrency; various data transfers (last two often not required). Such description is called behavioral C or BDL. Therefore, as seen in [23], the existing C code needs to be modified for in/out declaration, fifo requests, and so forth. In addition to control over loop unrolling and pipelining, Cynthesizer [19] also provides control over operator balancing, array flattening, chaining, mapping of arrays or array indexes to memory, and so forth. The designer may also select a part of design to be implemented in gate level design in a given number of cycles. Some of examples of implemented algorithms include multimedia applications [24]: H.264, video scaling, DCT, IDCT, motion estimation, NTSC encoder, VC1; security algorithms [25]: AES, DES, SHA, and MD5 encryption/decryption standards; digital media and security applications for wireless devices [26]: Viterbi encoders and decoders and proprietary noise rejection algorithms.

In case of all of the tools, the data path is built "on the fly" and heavily codependent on controller generation. Moreover, the resulting controller is usually in FSM style. The use of the FSM imposes size constraints for the design. Some of the tools, like Behaviour Synthesizer Cyber [18], and Cynthesizer [19], do provide FSM partitioning or hierarchical FSMs in order to expand beyond these constraints. To overcome capacity issues Catapult then uses its hierarchical engine to synthesize each function to concurrent hierarchical blocks with autonomous FSMs, control logic, and datapaths [27]. The fundamental difference is in the separation of a data path generation from a controller generation: this allows us to analyze code and perform profiling before the data path generation. Moreover, the separation of data path and controller generation reduces the problem size, therefore, reducing the size and quantity of the data structures that a tool needs to maintain and manipulate on. Besides, while all the above-mentioned tools do allow that a designer gives guideline to a tool, there is no mechanism by which a designer may influence a choice of particular components (other than inferring via code change in case of Catapult). Therefore, after the design has been made, designer may not make any modifications in the datapath. Contrary, the proposed technique separates creation of the data path and the controller, which automatically overcomes size constraint. Also, the designer may specify a subset of components and have the remaining of the data path automatically generated. Finally, the data path can be modified as little or as much after the automatic generation. Therefore, we find that providing designer with ability to control the automated design process and the ability to handle any size of C code are valuable assets in data path generation.

Many traditional HLS algorithms, such as [28, 29], create data path while performing scheduling and binding. The work in [28] uses ILP formulation with emphasis on efficient use of library components, which makes it applicable to fairly small input code. The work in [29] tries to balance distribution of operations over the allowed time in order to minimize resource requirement hence the algorithm makes decisions considering only local application requirements. The work in [30] takes into account global application requirements to perform allocation and scheduling simultaneously using simulated annealing. In contrast with the previous approaches, we separate data path creation from the scheduling and/or binding, that is, controller creation. This separation allows potential reuse of created data path by reprogramming, controllability over the design process, and use prelayout information for data path architecture creation.

The work in [31–35] separate allocation from binding and scheduling. The work in [31] uses "hill climbing" algorithm to optimize number and type of functional unit allocated, while the work in [32] applies clique partitioning in order to minimize storage elements, units, and interconnect. The work in [33] uses the schedule to determine the minimum required number of functional units, buses, register files, and ROMs. Then, the interconnect of the resulting data path is optimized by exploring different binding options for data types, variables, and operations. In [34], the expert system breaks down the global goals into local constraints (resource, control units, clock period) while iteratively moves toward satisfying the designer's specification. It creates and evaluates several intermediate designs using the schedule and estimated timing. However, all of the afore-mentioned traditional HLS techniques use FSM-style controller. Creation and synthesis of such state machine that correspond to thousands of lines of C code which, to the best of our knowledge, is not practically possible. In contrast to this, having programmable controller allows us to apply our technique to (for all practical purposes) any size of C code, as it was shown in Section 4.

Similarly to our approach, the work in [35] does not have limitations on the input size, since it uses horizontally microcoded control unit. On the other hand, it requires specification in language other than C and it produces only nonpipelined designs, none of which is the restriction of the proposed technique.

9. Conclusions and Future Work

In this paper, we presented a novel solution to constructing a processor core from a given application C code. We first create an initial data path design by matching code properties to hardware elements. Then, we iteratively refine it under given user constraints. The proposed technique allows handling of any size of C code, controllability of the design process, and independent optimization of data path and controller. We presented results for wide range of benchmarks, including industrial size applications like the MP3 decoder. Each data path architecture was generated in less than a minute allowing the designer to explore several different configurations in much less time than required for manual design. We define a total cost of a design in terms of total number of slices and RAMs for all selected components, and performance in terms of number of the execution cycles. Our experiments showed that up to 82.13% of slices and 87.46% of RAMs were saved. The number of execution cycles was 18.84% more in case of a single benchmark and for the remaining benchmarks, the maximum increase in the number of cycles was 0.68%. We measured design refinement quality on an example of *dct32* for which we synthesized all the designs on an FPGA board. We also showed that the best generated data path architecture is only 23% slower and had 2.29 times more BRAMs and 0.64 times slices utilized compared to the manual design. In the future, we plan of optimizing the generated core area, performance, and power by automatically determining the best control and data path pipeline configuration.

References

[1] D. D. Gajski, F. Vahid, S. Narayan, and J. Gong, *Specification and Design of Embedded Systems*, Prentice-Hall, Englewood Cliffs, NJ, USA, 1994.

[2] D. Gajski, "Nisc: the ultimate reconfigurable component," Tech. Rep. TR 03-28, University of California-Irvine, 2003.

[3] M. Reshadi and D. Gajski, "A cycle-accurate compilation algorithm for custom pipelined datapaths," in *Proceedings of the 3rd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and Systems Synthesis CODES+ISSS*, pp. 21–26, September 2005.

[4] M. Reshadi, B. Gorjiara, and D. Gajski, "Utilizing horizontal and vertical parallelism with a no-instruction-set compiler for custom datapaths," in *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pp. 69–74, October 2005.

[5] D. Gajski and M. Reshadi, "Nisc application and advantages," Tech. Rep. TR 04-12, University of California-Irvine, 2004.

[6] J. Trajkovic and D. D. Gajski, "Generation of custom co-processor structure from C-code," Tech. Rep. CECS-TR-08-05, Center for Embedded Computer Systems, University of California-Irvine, 2008.

[7] B. Gorjiara, M. Reshadi, P. Chandraiah, and D. Gajski, "Generic netlist representation for system and pe level design exploration," in *Proceedings of the 4th International Conference on Hardware/Software Codesign and System Synthesis*, pp. 282–287, ACM, New York, NY, USA, 2006.

[8] B. Gorjiara, M. Reshadi, and D. Gajski, "Generic architecture description for retargetable compilation and synthesis of application-specific pipelined ips," in *Proceedings of the Proceedings of International Conference on Computer Design (ICCD '06)*, 2006.

[9] D. Shin, A. Gerstlauer, R. Dömer, and D. D. Gajski, "An interactive design environment for C-based high-level synthesis," in *IESS*, A. Rettberg, M. C. Zanella, R. Dömer, A. Gerstlauer, and F.-J. Rammig, Eds., vol. 231 of *IFIP*, pp. 135–144, Springer, 2007.

[10] Xilinx: MicroBlaze Soft Processor Core, 2008, http://www.xilinx.com/tools/microblaze.htm.

[11] R. Ang, http://www.cecs.uci.edu/presentation_slides/ESE-Back-End2.0-notes.pdf.

[12] Tensilica: Xtensa LX, 2005, http://www.tensilica.com/products/xtensa_LX.htm.

[13] Automated Configurable Processor Design Flow, White Paper, Tensilica, 2005, http://www.tensilica.com/pdf/Tools_white_paper_final-1.pdf.

[14] Diamond Standard Processor Core Family Architecture, White Paper, Tensilica, 2006, http://www.tensilica.com/pdf/DiamondWP.pdf.

[15] D. Goodwin and D. Petkov, "Automatic generation of application specific processors," in *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, 2003.

[16] Stretch: S5000 Software-Configurable Processors, 2008, http://www.stretchinc.com/products/devices.php.

[17] Mentor Graphics Catapult Synthesis, 2008, http://www.mentor.com/esl/catapult/overview//index.cfm.

[18] NEC CyberWorkBench, 2008, http://www.necst.co.jp/product/cwb/english/index.html.

[19] Forte Design System Cynthesizer, 2008, http://www.forteds.com/products/cynthesizer.asp.

[20] Mentor Graphics Catapult Synthesis—Ericsson Success Story, 2011, http://www.mentor.com/esl/success/ericsson-success.

[21] Mentor Graphics Technical Publications: Designing High Performance DSP Hardware using Catapult C Synthesis and the Altera Accelerated Libraries, 2008, http://www.mentor.com/techpapers/fulfillment/upload/ mentorpaper_36558.pdf.

[22] Mentor Graphics Technical Publications: Alcatel Conquers the Next Frontier of Design Space Exploration using Cat-apult C Synthesis, 2008, http://www.mentor.com/techpapers/fulfillment/upload/mentorpaper_22739.pdf.

[23] K. Wakabayashi, "C-based synthesis experiences with a behavior synthesizer, Cyber," in *Proceedings of the Proceedings of the Conference on Design, Automation and Test in Europe (DATE '99)*, p. 83, 1999.

[24] Forte Design System Cynthesizer—Applications: Digital Media, 2008, http://www.forteds.com/applications/digitalmedia.asp.

[25] Forte Design System Cynthesizer—Applications: Security, 2008, http://www.forteds.com/applications/security.asp.

[26] Forte Design System Cynthesizer—Applications: Wireless, 2008, http://www.forteds.com/applications/wireless.asp.

[27] Mentor Graphics Catapult Datasheet, 2010, http://www.mentor.com/esl/catapult/upload/Catapult_DS.pdf.

[28] B. Landwehr, P. Marwedel, and R. Dömer, "OSCAR: optimum simultaneous scheduling, allocation and resource binding based on integer programming," in *Proceedings of the European Design Automation Conference*, pp. 90–95, IEEE Computer Society Press, Grenoble, France, 1994.

[29] P. G. Paulin and J. P. Knight, "Force-directed scheduling for the behavioral synthesis of ASIC's," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 8, no. 6, pp. 661–679, 1989.

[30] S. Devadas and A. R. Newton, "Algorithms for hardware allocation in data path synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 8, no. 7, pp. 768–781, 1989.

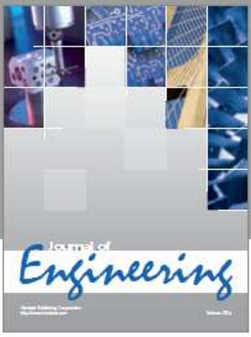
[31] P. Gutberlet, J. Müller, H. Krämer, and W. Rosenstiel, "Automatic module allocation in high level synthesis," in *Proceedings of the Conference on European Design Automation (EURO-DAC '92)*, pp. 328–333, 1992.

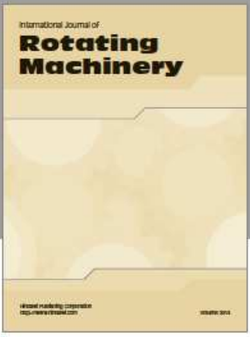
[32] C. J. Tseng and D. P. Siewiorek, "Automated synthesis of data paths in digital systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 5, no. 3, pp. 379–395, 1986.

[33] F.-S. Tsai and Y.-C. Hsu, "STAR: an automatic data path allocator," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 11, no. 9, pp. 1053–1064, 1992.

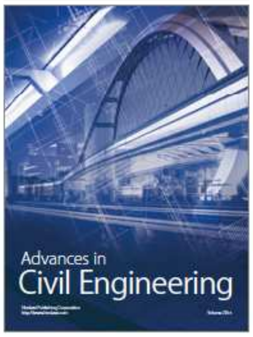
[34] F. Brewer and D. D. Gajski, "Chippe: a system for constraint driven behavioral synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 9, no. 7, pp. 681–695, 1990.

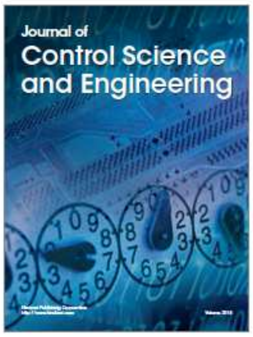
[35] P. Marwedel, "The MIMOLA system: detailed description of the system software," in *Proceedings of the Design Automation Conference*, ACM/IEEE, 1993.


Journal of
Engineering

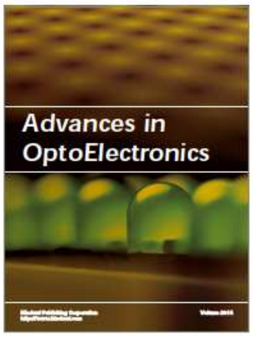
The Scientific World Journal

International Journal of
Rotating Machinery

Journal of
Sensors

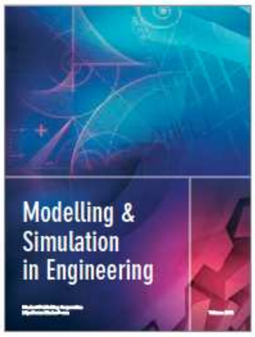
International Journal of
Distributed Sensor Networks

Advances in
Civil Engineering


Journal of
Control Science and Engineering

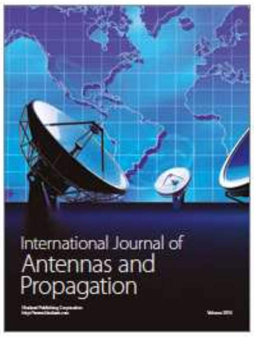
Journal of
Robotics

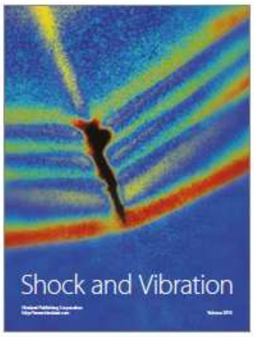
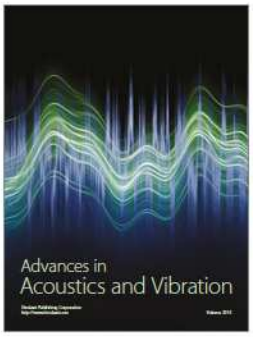
Journal of
Electrical and Computer Engineering

Advances in
OptoElectronics

VLSI Design

International Journal of
Navigation and Observation

Modelling & Simulation in Engineering

International Journal of
Aerospace Engineering

International Journal of
Chemical Engineering

International Journal of
Antennas and Propagation

Active and Passive Electronic Components

Shock and Vibration

Advances in
Acoustics and Vibration