Application-Specific Processors

12

Tulika Mitra

Abstract

General-Purpose Processors (GPPs) and Application-Specific Integrated Circuits (ASICs) are the two extreme choices for computational engines. GPPs offer complete flexibility but are inefficient both in terms of performance and energy. In contrast, ASICs are highly energy-efficient, provide the best performance at the cost of zero flexibility. Application-specific processors or custom processors bridge the gap between these two alternatives by bringing in improved power-performance efficiency within the familiar software programming environment. An application-specific processor architecture augments the base instruction-set architecture with customized instructions that encapsulate the frequently occurring computational patterns within an application. These custom instructions are implemented in hardware enabling performance acceleration and energy benefits. The challenge lies in inventing automated tools that can design an application-specific processor by identifying and implementing custom instructions from the application software specified in high-level programming languages. In this chapter, we present the benefits of application-specific processors, their architecture, automated design flow, and the renewed interests in this class of architectures from energy-efficiency perspective.

Acronyms

ALU	Arithmetic-Logic Unit
ASIC	Application-Specific Integrated Circuit
BERET	Bundled Execution of REcurring Traces
CAD	Computer-Aided Design
CCA	Configurable Compute Accelerator
CFG	Control-Flow Graph

T. Mitra (✉)
Department of Computer Science, School of Computing, National University of Singapore, Singapore, Singapore
e-mail: tulika@comp.nus.edu.sg

S. Ha, J. Teich (eds.), *Handbook of Hardware/Software Codesign*,
DOI 10.1007/978-94-017-7267-9_13

CFU	Custom Functional Unit
CIS	Custom Instruction-Set
DFG	Data-Flow Graph
DISC	Dynamic Instruction-Set Computer
DSP	Digital Signal Processor
FPGA	Field-Programmable Gate Array
GPP	General-Purpose Processor
GPU	Graphics Processing Unit
ILP	Integer Linear Program
IR	Intermediate Representation
ISA	Instruction-Set Architecture
ISEF	Stretch Instruction-Set Extension Fabric
MAC	Multiply-Accumulator
MIMO	Multiple Input Multiple Output
MISO	Multiple Input Single Output
PFU	Programmable Functional Unit
PRISC	Programmable Instruction-Set Processor
RAM	Random-Access Memory
RISC	Reduced Instruction-Set Processor
RISPP	Rotating Instruction-Set Processing Platform
SFU	Specialized Functional Unit
VLIW	Very Long Instruction Word

Contents

12.1 Introduction

Over the years, the General-Purpose Processors (GPPs) has been established as the de facto choice for execution engine. Microprocessors – single chip GPPs – were invented in 1971 and have enjoyed unprecedented performance growth aided by technology scaling (Moore's law [47] for transistor density) and microarchitectural innovations (out-of-order execution, branch prediction, speculation, cache memory) [53]. GPPs are designed to support a wide range of applications through software programmability at reasonable power-performance point. The software is the key here to provide application-specific functionality or specialization.

The generic nature of GPPs is also the reason behind their inefficiency, both in terms of power and performance [30]. The GPPs need to support a comprehensive Instruction-Set Architecture (ISA) as the abstraction and interface to the software. But processing a computation expressed in a generic ISA involves significant overhead in the front end of the processor pipeline – such as instruction fetch, instruction decode, and register access – that does not contribute toward the core computations, which are the arithmetic or logical operations.

At the other end of the spectrum, we have the Application-Specific Integrated Circuits (ASICs) as the completely specialized execution engines. ASICs can provide unprecedented power-performance benefits compared to the GPPs as they completely eliminate the instruction processing overhead and only perform the required computations. But the efficiency comes at the cost of flexibility as ASICs does not provide any programmability (hence the name non-programmable accelerators for computations implemented in an ASIC). Any change in the application incurs complete redesign and fabrication cost. Thus ASICs are only suitable for critical computational kernels, such as video encoding/decoding, whose performance per watt requirements cannot be met through the GPPs.

The domain-specific processors offer an interesting design choice between the two extreme alternative of GPPs and ASICs. Graphics Processing Units (GPUs) and Digital Signal Processors (DSPs) are well-established examples of domain-specific processors. The ISA and the microarchitecture are carefully designed to accommodate the applications in a specific domain. Thus domain-specific processors attempt to balance specialization with programmability. Still these processors cannot possibly cover all the application domains, and programming them is not as straightforward as GPPs.

In this chapter, we will focus on an interesting and compelling alternative called the *application-specific processors* or *custom processors* [34]. We will use the terms application-specific processors and custom processors interchangeably in this chapter. An application-specific processor is a general-purpose programmable processor that has been configured and extended with respect to a particular application or application domain [24]. Configurability refers to the ability to select and set the size of different microarchitectural features such as number and types

of functional units, register file size, instruction and data cache size, associativity, etc. according to the characteristics of the application domain. For example, if an application does not use floating-point operations, the floating-point functional units can be eliminated from the underlying microarchitecture. Optimal setting of the configuration parameters for an application domain is a complex design space exploration problem [38] that has been investigated thoroughly [20, 29, 51] in the past decade.

However, in this chapter, we will focus on the extensibility part of the application-specific processors where the instruction-set architecture of the general-purpose processor is augmented with application-specific *Instruction-set extensions*, also known as the *custom instructions* . The custom instructions capture the key computation fragments or patterns within an application. The custom instructions are added to the base ISA of the general-purpose processor. The computation corresponding to each custom instruction is synthesized as a *Custom Functional Unit (CFU)*, and all the CFUs are included in the processor's data path alongside existing functional units, such as Arithmetic-Logic Units (ALUs), multipliers, and so on. The front end of the processor pipeline, for example, the decode stage, needs to be suitably changed to take in these new instructions. Similarly, the compiler and associated software tool chain are modified to support the custom instructions such that a new application can exploit additional instructions in the ISA. As a custom instruction combines a number of base instructions, it amortizes the front-end processing overhead. Moreover, the synthesis process of the CFU can be optimized so that the operations within a CFU can be parallelized and chained together to offer a very competitive critical path delay that is far shorter than the sum of the delays of the individual operations. These factors together lead to substantially improved performance and energy efficiency for application-specific processors compared to the GPPs. Thus, application-specific processors offer an easy and incremental path toward specialization, while still staying within the comfortable and familiar software programming environment of the GPPs.

Initially, application-specific processors were enthusiastically and successfully adopted in the embedded systems domain. They are an excellent match for this domain because an embedded system is designed to provide a well-defined set of functionalities throughout its lifetime. Thus the custom instructions can be constructed to accelerate the specific computations involved in providing the required functionality. The focus at that time has been on the automated design of the custom instructions. More concretely, given an application, how do we automatically enumerate the potential custom instructions (*custom instructions enumeration*) and choose the appropriate ones under area, energy, and/ or performance constraints (*custom instructions selection*)? A flurry of research activity in the past 15 years on design automation of custom processors has made significant advances, even though some open problems and challenges still remain unresolved. A number of processor vendors have offered commercial customizable processor along with the complete automation tool chain to enumerate, select, and synthesize the custom instructions, followed by the synthesis of the application-specific processor including the custom instructions, and the compiler to fully realize the potential of the custom instructions

from software with minimal engagement from the programmers. The Xtensa customizable processors from Tensilica [39] are the best examples of this design paradigm. In general, however, the interest in application-specific processors has been primarily restricted to the embedded systems area till very recently.

In the past five years or so, a number of technological challenges have brought the application-specific processors to the forefront even in the general-purpose and high-performance computing domains. First, the energy efficiency rather than performance has increasingly become the first-class design goal [48] for any system, be it battery-powered embedded and mobile platforms, or high-performance servers with continuous access to electrical power sockets. Second, the failure of Dennard scaling [19] back in around 2005 has had a devastating effect on the microprocessor design. As power per transistor does not scale any more with feature scaling from one technology generation to another, increasing transistor density following Moore's law leads to increasing power density for the chip. Thus the core has to operate at a frequency that is lower than the default frequency enabled through technology scaling, so as to keep the power density of the chip within acceptable limits. Moreover, complex microarchitectural features have long ceased to provide any further performance improvements [50] as we have hit the instruction-level parallelism wall [67] and the memory wall [69]. This loss of frequency scaling and microarchitectural enhancement have kept the single-core performance at a standstill for the past ten years or so. Instead, the abundance of on-chip transistors as per Moore's law has been employed to build multi- or even many-core chips consisting of identical general-purpose processor cores [22]. The multi-cores are perfect match for applications with high thread-level parallelism. But the sequential fragment of the application still suffers from poor single-core performance and thereby limits the performance potential of the entire application according to Amdahl's law [3].

More importantly, though, multi-core scaling is also coming to an end in the near future [21]. As we increase the number of core on chip, the failure of Dennard scaling implies that the total chip power increases. But the packaging and cooling solutions are not sufficient to handle this increasing chip power. Thus we can have more core on chip; but we can only power on a subset of these cores to meet the chip power budget. This phenomenon has been termed *dark silicon* [43], where a significant fraction of the chip remains unpowered or dark. The dark silicon era naturally leads to the design of heterogeneous multi-core architectures [44] rather than the homogeneous multi-core designs prevalent today. Depending on the applications executing on the architecture, only the cores that are well suited for the current workload can be switched on at any point in time, leading to performance- and energy-efficient computing [61]. In other words, the cheap silicon area (that would have remained unused anyway due to limited power budget) is being traded to add execution engines that will be used sparingly and judiciously. Thus, the dark silicon era has automatically paved the way for specialized cores and have generated renewed interest in application-specific or custom processors [11, 62].

The objectives and challenges in designing application-specific processors are somewhat different, though, for embedded computing systems and general-purpose

high-performance computing systems. First, unlike embedded systems that execute only a fixed set of applications throughout its lifetime, general-purpose computing systems encounter a diverse set of workloads that may be unknown at design time. So, while we can profile the applications and design the best set of custom instructions to accelerate the embedded workload, the same approach is not feasible in general-purpose systems. This leads to the design of custom instructions that are somewhat parameterized and can be reused across workloads [65]. Another possible direction is the design of custom functional units or a flexible fabric that can be reconfigured to support a varied set of custom instructions [32]. Second, custom instruction enumeration in the context of embedded systems has been generally restricted to computations and has mostly excluded storage elements inside a custom instruction. The implication of this choice is that custom instructions are small and the gain in performance comes from repeated occurrence – either statically in the program code or dynamically due to its presence in a loop body – of the same custom instructions. But the performance gain and energy improvement are still modest with small custom instructions compared to the potential when large custom instructions are formed combining computations with storage elements and without any restrictions in terms of input/output operands [30, 71]. Thus bigger custom instructions with storage elements are recommended for general-purpose high-performance computing systems.

The rest of the chapter is organized as follows. We will start off with a brief overview of the architecture of the application-specific processors and the corresponding design automation flow. This will be followed by detailed discussion on custom instructions enumeration and selection algorithms. We will next present customizable architectures with dynamic or reconfigurable custom instructions and the necessary algorithms to identify custom instructions and exploit such architectures. Finally, we will provide a quick review of recent attempts to bring customization to general-purpose computing platforms.

12.2 Architectural Overview and Design Flow

In this section, we provide an overview of the application-specific processor architecture and the corresponding design flow.

12.2.1 Application-Specific Processor Architecture

The generic architecture of an application-specific processor or custom processor is shown in Fig. 12.1. The instruction-set architecture of the base processor is modified to include the application-specific instructions or custom instructions. A custom instruction encapsulates a frequently occurring computational pattern involving a number of basic operations (see Fig. 12.1b). Each custom instruction is implemented as a CFU. The pipeline data path of the base processor core is augmented with the CFUs alongside existing functional units (ALU, multiplier,

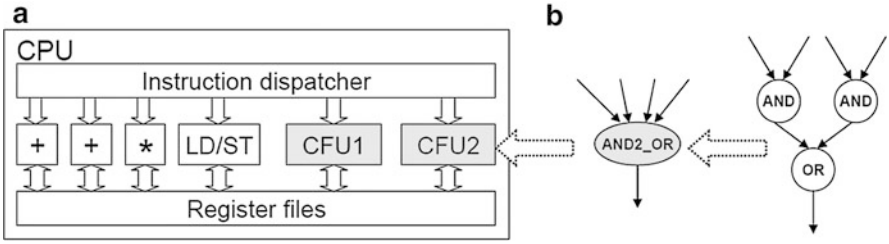

Fig. 12.1 Architecture of an application-specific processor with two CFUs

load/store units, etc.) and are treated the same way as shown in Fig. 12.1a. The CFUs can access the input and output operands stored in the register file just like regular functional units. Thus a custom instruction can be fetched, decoded, and dispatched to the respective CFU just like normal instructions. The biggest advantage of custom instructions is the improved latency and decreased power consumption to execute the computational pattern. For example, in Fig. 12.1b, the custom instruction consists of three basic operations (two AND and one OR). In the base processor core, this computation pattern requires fetch, decode, and execution of three instructions. This is reduced to only one custom instruction immediately saving the fetch, decode, dispatch time, and energy. More importantly, as the custom instruction is synthesized in hardware, the implementation can take advantage of parallelism. For example, the two AND operations can be executed in parallel. Also the critical path now consists of AND-OR operation. If the critical path can fit inside the cycle time of the base processor, the CFU can execute the custom instruction in one clock cycle. This is the case when the clock cycle time of the processor is determined by a complex operation such as Multiply-Accumulator (MAC) or even addition operation whereas the basic operations on the critical path are much simpler (such as logical operations) such that multiple of them can be chained together in a single cycle. If the critical path exceeds the cycle time, then the CFU will execute the custom instruction in multiple cycles (possible pipelined), say N, where $N = (critical\ path\ latency)/(clock\ period)$. Still, N is likely much less than M, the minimum number of cycles required to execute the basic operations in the computational pattern individually. A positive side effect of the custom instructions approach is the increased code density and hence reduced code size, which is an important issue in embedded systems. A custom instruction also reduces the number of register accesses significantly as the intermediate results (the results of the AND operations) need not be written back and read from the register file. In the example pattern in Fig. 12.1b, the custom instruction needs four register reads and one register write as opposed to six register reads and three register writes for the original three instruction sequence.

A custom instruction may require more input and output operands compared to the typical two-input, one-output basic Reduced Instruction-Set Processor (RISC) instructions. Yu and Mitra [71] showed experimentally that the performance potential of custom instructions improves with increasing number of input and

output operands. Later, Verma et al. [66] proved that under fairly weak assumptions, increasing the number of nodes in a pattern (which typically results in increased input/output operands) can never lead to reduced speedup. But supporting more input and output operands within the framework of normal processor pipeline requires some additional support from the underlying microarchitecture and the ISA, which will be discussed in Sect. 12.3.4. The maximum number of input and output operands supported per custom instruction in an architecture defines the custom instruction identification algorithm for that architecture.

12.2.2 Design Flow

The main design effort in tailoring an extensible processor is to define the custom instructions for a given application to meet certain design goals. Identifying suitable custom instructions is essentially a hardware-software partitioning problem that divides the computations between the software execution (using base instructions) and hardware execution (using custom instructions). Various design constraints must be satisfied in order to deliver a viable system, including performance, silicon area cost, power consumption, and architectural limitations. This problem is frequently modeled as a single-objective optimization where a certain aspect is optimized (e.g., performance), while the other aspects are considered as constraints.

The generic design flow for application-specific processors is presented in Fig.12.2. The input to this design flow is the reference software implementation of the application written in a high-level programming language such as C or C++. In the application-specific processor design flow, the compiler performs additional steps toward customizing the base processor core: identifying the computational patterns that can potentially serve as custom instructions, selecting a subset of these patterns under various constraints to maximize the objective function, and finally synthesizing the new CFUs and generating the binary executables under the new instruction set. This automated hardware-software codesign approach ensures that large programs can be explored and the software programmers can easily adapt to the design flow without in-depth hardware knowledge.

In a generic compiler, high-level language statements are first transformed by the compiler front end to an Intermediate Representation (IR). Various machine-independent optimizations are carried out on the IR. Then, the back end of the compiler generates binary executables for the target processor by binding IR objects to actual architectural resources: operations to instructions, operands to registers or memory locations, and concurrencies and dependencies to time slots, through instruction binding, register allocation, and instruction scheduling, respectively. Various machine-dependent optimizations are also performed in the back end. The custom instruction identification, selection, and binary executable generation with custom instructions are all performed in the back end on the IR.

The IR consists of a Control-Flow Graph (CFG) and a Data-Flow Graph (DFG) (also called a data-dependency graph). The CFG is a graph structure where the nodes are the basic blocks – sequence of instructions with a single-entry and single-exit

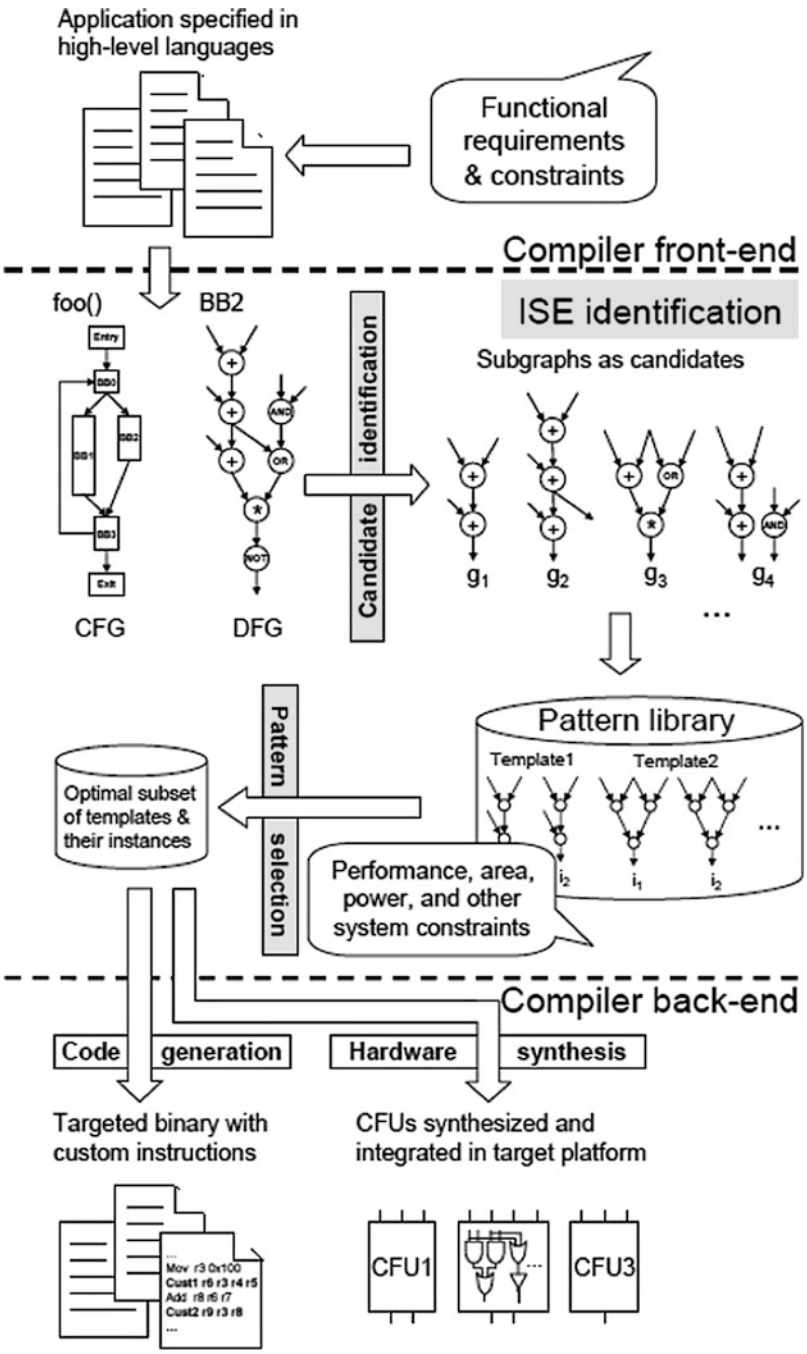

Fig. 12.2 The design flow for application-specific processors [52]

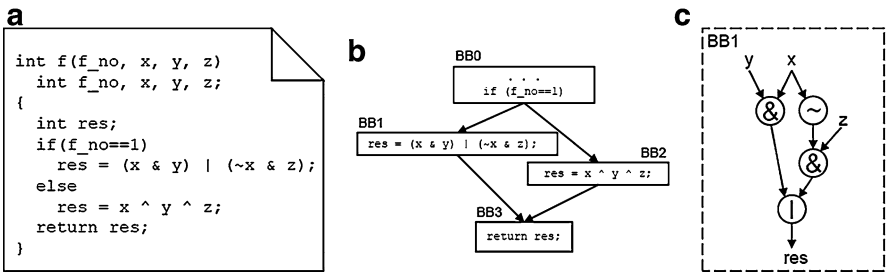

Fig. 12.3 Control-flow graph and data-flow graph

point. The edges among the basic block represent control-flow transfer from one basic block to another through conditional statements, loops, function calls, etc. For each basic block, the computation within a basic block is captured by a DFG where the nodes represent the operations and the edges represent dependency among the operations. Figure 12.3 shows an example of CFG and DFG corresponding to a code segment. In the base processor, each operation in the DFG typically corresponds to an instruction in the ISA. However, a custom instruction can cover a cluster of operations and is hence captured as a subgraph of the DFG.

Normally, custom instructions identification is restricted to the DFG within each basic block. However, basic blocks are usually quite small consisting of only few instructions. Thus there is limited opportunity to extract large custom instructions within basic blocks and obtain significant performance improvement. A limit study by Yu and Mitra [71] showed significant benefit in identifying custom instructions across basic block boundaries. Thus it is essential to create larger blocks containing multiple basic blocks, for example, traces, superblocks, and hyperblocks, and provide architectural support to for these extended blocks. In that case, the DFG can be built for these larger blocks.

The custom instruction identification is essentially a subgraph enumeration problem. For each DFG, the computational patterns that satisfy certain constraints are enumerated as potential custom instruction candidates. If a pattern appears multiple times either within the same basic block or different basic blocks, these are considered as different instances of the same computation pattern. This step builds a library of potential candidate patterns. The next step evaluates the different patterns and selects a subset that optimizes certain goals under the different constraints. This step requires profiling data. The application is profiled on the base processor with representative input data sets. The profiling identifies the *hot spots* where most of the computation time is spent, and these hot spots are ideal candidates for implementation as custom instructions and may benefit from faster execution on CFUs. The performance benefit of each pattern combined with the frequency of execution of that pattern from profiling information is used in the custom instructions selection process.

Finally, the selected patterns are passed on to the last step of the design framework. Each selected pattern is synthesized in hardware as a CFU, and the CFUs are added to the data path of the processor. The processor control is modified to accommodate these new instructions. The new instructions are acknowledged in the instruction binding stage of the compiler either by simple peephole substitution or by the pattern matcher to produce an executable that exploits the custom instructions.

12.3 Custom Instructions Identification and Selection

We start off this section by first presenting the terminology and definitions related to custom instructions identification from a data-flow graph.

12.3.1 Formal Definitions

The custom instructions are identified on the data-flow graphs corresponding to the basic blocks of a program. A *DFG* G is a directed acyclic graph that captures the flow of data within a basic block. The set of nodes or vertices $V(G)$ represent the operations, while the set of edges $E(G)$ represent the flow of data among the operations. An edge $e = (u, v)$ where $e \in E(G)$, $u, v \in V(G)$ denotes a data dependency from u to v where v can execute only after u completes execution. Figure 12.4 shows a data-flow graph consisting of the blue and the red nodes.

Each graph G is also associated with a supergraph G^+ that contains additional nodes V^+ and edges E^+. The additional nodes V^+ represent the input and output variables of the basic block, while the additional edges E^+ connect the nodes in V^+ to the nodes in $V(G)$. The green nodes in Fig. 12.4 correspond to the additional

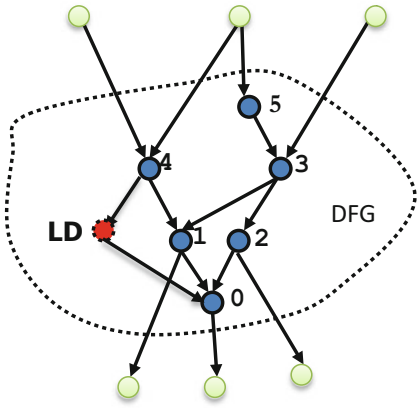

Fig. 12.4 Data-flow graph

nodes to create the supergraph corresponding to the DFG. The DFG requires three input variables and three output variables.

Given an edge $e = (u, v)$, the node u is called the *immediate predecessor* of node v, while v is the *immediate successor* of u. Let us define $IPred(v) = \{u|(u, v) \in E(G)\}$ and $ISucc(v) = \{u|(v, u) \in E(G)\}$ as the *immediate predecessor set* and *immediate successor set* of node v, respectively. The in-degree and out-degree of node v are $|IPred(v)|$ and $|ISucc(v)|$, respectively. A *path* $v_0 \rightsquigarrow v_n$ is a sequence $\langle v_0, v_1, \ldots, v_n \rangle$ of nodes where $v_i \in V(G)$ for $0 \leq i \leq n$ such that $(v_i, v_{i+1}) \in E(G)$ for $0 \leq i \leq (n-1)$. We define predecessor set $Pred(v)$ as the set of nodes that can reach v through a path in the graph, i.e., $Pred(v) = \{u|u \rightsquigarrow v \in G\}$. Similarly, we define successor set $Succ(v)$ as the set of nodes that can be reached from v through a path in the graph, i.e., $Succ(v) = \{u|v \rightsquigarrow u \in G\}$.

A *source* node in the supergraph G^+ has no predecessor (zero in-degree), while a *sink* node has no successor (zero out-degree). The source nodes represent the input operands (including immediate operands), while the sink nodes represent the output operands corresponding to the DFG. Together the source nodes and the sink nodes correspond to V^+. The remaining nodes $V(G)$ are the *internal* nodes of the data-flow graph that represent the operations (arithmetic and logical operations, load/store, etc.) supported by the ISA of the baseline processor. The green nodes in Fig. 12.4 are the source and sink nodes, while the blue and red nodes are the internal nodes of the data-flow graph.

A custom instruction or a *pattern* C is a subgraph of the DFG G induced by the set of vertices $C \subseteq V(G)$. The subgraph consists of vertices $V(C) \subseteq V(G)$ and edges $\{(u, v) \in E(G)|u, v \in V(C)\}$. For example, the set of vertices $\{0, 1, 2\}$ form a pattern. A node u is an *input* of pattern C if $u \notin V(C)$, $v \in V(C)$, and there exists an edge $(u, v) \in E(G^+)$. Similarly, a node u is an *output* of pattern C if $u \in V(C)$, $v \notin V(C)$, and there exists an edge $(u, v) \in E(G^+)$. Let $In(C)$ and $Out(C)$ be the set of input and output nodes of pattern C, respectively. The input nodes represent the input values or variables used by the custom instruction, while the output nodes present values produced by the custom instruction that will be used by other operations, either in G or in another basic block. Many architectures impose constraints on the number of inputs and outputs per custom instruction, known as the *I/O constraint*. The nodes $3, 4, LD$ are the input and nodes $0, 1, 2$ are all output corresponding to the pattern $\{0, 1, 2\}$ in Fig. 12.4.

An architecture may impose restrictions on the kind of operations that may be included as part of a custom instruction. Most architectures do not allow memory operations (load/store) and control operations (branch) to be part of custom instructions. A node is *valid* if it can be part of a custom instruction; otherwise it is *invalid*. By definition, the source and the sink nodes are not part of custom instructions and hence are invalid. Internal nodes can be invalid too if the corresponding operation cannot be accommodated within a custom instruction. For example, the red node in Fig. 12.4 corresponds to a load operation, and it is an invalid node in addition to the green source/sink nodes. Let $X(G) \in V(G)$ be the set of internal invalid nodes of the DFG.

The following special patterns are of interest in custom instructions enumeration problem.

Connected pattern A pattern C is connected if for any pair of nodes $u, v \in V(C)$ in the pattern, there exists a path between u and v within the pattern in the undirected graph that underlies the directed induced subgraph of C. $\{0, 1, 2\}$ is a connected pattern in Fig. 12.4.

Disjoint pattern A pattern is disjoint if it is not connected. A disjoint pattern consists of two or more connected patterns. The pattern $\{3, 4, 5\}$ is a disjoint pattern consisting of two connected patterns $\{4\}$ and $\{3, 5\}$.

MISO pattern A pattern C with only one output ($|Out(C) = 1|$) is called a Multiple Input Single Output (MISO) pattern. Clearly, a MISO pattern should be a connected pattern. $\{3, 5\}$ is a MISO pattern with 3 as the output. Note that $\{0, 1, 2\}$ is not a MISO pattern as it has three outputs.

MIMO pattern A pattern with multiple input and multiple output is called a Multiple Input Multiple Output (MIMO) pattern. We can further distinguish between connected MIMO pattern and disjoint MIMO pattern.

Convex pattern A pattern C is convex if any intermediate node t on any path in the DFG G from a node $u \in V(C)$ to a node $v \in V(C)$ belongs to C, i.e., $t \in V(C)$. A pattern is non-convex if there exist at least one path in the DFG G from a node $u \in V(C)$ to a node $v \in V(C)$ that contains an intermediate node $t \in V(G) \setminus V(C)$. A non-convex pattern is infeasible because it cannot be executed as a custom instruction in an atomic fashion. The pattern $\{0, 1, 2, 4\}$ is a non-convex pattern because there is a path from 4 to 0 that contains the invalid LD node.

Valid pattern A pattern C is a valid custom instruction candidate if (a) the pattern does not contain any invalid node $V(C) \cap X(G) = \phi$, (b) the pattern is convex, and (c) the pattern satisfies the I/O constraints imposed by the architecture, i.e., $In(C) \leq N_{in}$ and $Out(C) \leq N_{out}$ where N_{in} and N_{out} are the maximum number of inputs and outputs allowed per custom instruction, respectively.

Maximal valid pattern A pattern C is a maximal valid pattern if it is a valid pattern, and there is no $v \in V(G) \setminus V(C)$ such that the subgraph induced by the set of vertices $(V(C) \cup v)$ is a valid pattern. For example, $\{0, 1, 2, 3, 5\}$ is a maximal valid pattern.

We will primarily concentrate on techniques to enumerate connected patterns as the disjoint patterns can be easily constructed from the connected patterns.

In the context of pattern enumeration, it is useful to define *topologically sorted level* of the nodes in the directed acyclic graph G. Each node in $V(G)$ that is only connected to the source nodes V^+ has level 0. The level of any other node $level(v) = l$ if the longest path (in terms of number of edges) from some source node to v is of length $l + 1$. We can also define the order of the nodes in G according to the topological sort; if G contains an edge $e = (u, v)$, then v should appear after

u in this *topologically sorted order*. In the example DFG of Fig. 12.4, nodes 4, 5 belong to level 0, node 3 belongs to level 1, and nodes 1, 2 belong to level 2, and node 0 belongs to level 3. A topologically sorted order for the valid nodes of the DFG is 5, 4, 3, 2, 1, 0.

12.3.2 Enumeration of MISO Patterns

The simplest custom instruction enumeration problem is the one of identifying maximal MISO (MaxMISO) patterns. A linear-time algorithm to identify MaxMISO patterns has been presented in [2]. Note than a MaxMISO pattern has a single output. So it is efficient to start with the sink nodes and proceed level by level to the source nodes. We initialize a MaxMISO pattern $C = \{v\}$ with any node v that has only one output. We can then iteratively add in the predecessors' nodes $IPred(v)$ at the next level to the pattern C as long as $u \in IPred(v)$ does not contribute a new output to the pattern or u is an invalid node ($u \in X$). The process can continue with the predecessors of the newly added nodes in the pattern till we have no more predecessors to consider. As the intersection of MaxMISOs should be empty, i.e., two MaxMISOs cannot overlap, it is easy to see that the algorithm will have linear time. In contrast, Cong et al. [18] identify all valid MISO patterns. This problem, in the most general case, has exponential complexity because each node can potentially be included or excluded from a candidate pattern. Thus, Cong et al. [18] impose restrictions on number of input operands and/or area constraint to limit the number of candidate patterns resulting in efficient pattern generation.

12.3.3 Exhaustive Enumeration of All Valid Patterns

The exhaustive enumeration of all possible valid connected patterns of a DFG G under the convexity and a specified I/O constraint is quite challenging. At first glance, in the worst case, the number of possible patterns can be $(2^{|V(G)|}$ as each vertex can be either included or excluded to form a pattern. But closer examination of the problem reveals that most of the patterns are not valid due to convexity and/or I/O constraints. Chen at al. [13] and Bonzini and Pozzi [9] have proven that the number of such valid patterns for a graph G with bounded in-degree (which is true for data-flow graphs of programs) is at most $|V(G)|^{N_{in}+N_{out}}$. If the I/O constraint is quite tight, then the enumeration is fast. A number of algorithms have been proposed in the literature to solve this problem efficiently. Gutin et al. [28] have designed an algorithm with worst-case complexity of $O(|E(G)| \times N_{in}^2 \times (n + |V(G)|^{N_{out}}))$ where n is the number of valid patterns. This bound is theoretically optimal if the number of valid patterns n is asymptotically smaller than $|V(G)|^{N_{in}+N_{out}}$. As a follow-up work, Reddington et al. [59] have proposed a version of this algorithm that has $|V(G)|^{N_{in}+N_{out}+1}$ worst-case complexity, but runs much faster in practice. At the

point of this writing, the algorithm by Reddington et al. [59] is known to be the fastest algorithm in practice for exhaustive enumeration of all valid convex patterns under I/O constraints.

In the following, we present a few representative algorithms so that the readers can better appreciate the problem and the possible solutions.

12.3.3.1 Search-Tree-Based Enumeration Algorithm

We present the first and the simplest algorithm proposed for this purpose by Atasu et al. [7,55], which is based on a search tree. It has been later shown by Reddington and Atasu [58] that the search-tree-based exhaustive algorithm has worst-case complexity of $|V(G)|^{N_{in}+N_{out}+1}$, which is quite close to the theoretical complexity.

The main insight behind the search-tree-based algorithm is that if a pattern C is not convex, then adding in the nodes that appear in lower levels in G (according to the topological sort) compared to the nodes in C would not make the resulting pattern convex. For example, the pattern $\{0, 1, 3\}$ in Fig. 12.4 is non-convex. Including the nodes 4 and 5 to this pattern will not remove the non-convexity.

Similarly, if a pattern C violates the output constraint, then adding in the nodes that appear in lower levels in G (according to the topological sort) compared to the nodes in C would not decrease the number of output operands. For example, the pattern $\{0, 1, 2\}$ in Fig. 12.4 requires three output operands. Adding the nodes that are at lower level in topological sort 3, 4, 5 to this pattern will not reduce the number of output operands of the resulting pattern. In other words, we can easily prune away all such patterns without examining them explicitly. This effective pruning of the search space leads to the efficiency of the algorithm.

The search tree is a binary tree of nodes that represent the possible patterns. The root represents an empty pattern. The nodes are added in this tree in *reverse topological order*. Let the order of the nodes of G in reverse topological order by $v_1, v_2, \ldots v_{|V(G)|}$. The branches at level i corresponds to including (the 0-branch) or excluding (the 1-branch) the node v_i in the pattern. The pattern along the 0-branch is the same as the parent node and can be ignored. The search tree can be constructed in this manner till we examine the node $v_{V(G)}$ at level $|V(G)|$. Figure 12.5 shows the search tree corresponding to the DFG in Fig. 12.4. The shaded regions represent the pruned design space.

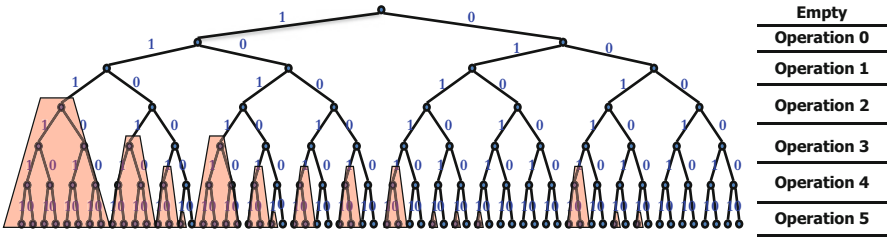

Fig. 12.5 Search tree for pattern enumeration

As mentioned earlier, the complexity of the search is reduced by employing the convexity and the output constraint. Suppose that the output constraint has been violated at a pattern. Then adding the nodes that appear later in the reverse topological order cannot possibly reduce the number of outputs for the pattern. Similarly, if the convexity constraint is violated, then it is not possible for the pattern to regain feasibility by adding the nodes that appear later in the reverse topological order. Therefore, when we reach a search tree node where either the convexity constraint or the output constraint is violated, the subtree rooted at that node can be completely eliminated from the search space.

Clearly, the search-tree-based exhaustive algorithm can prune based on the output constraint, but it cannot prune based on the input constraint. Each pattern simply needs to be checked for violation of the input constraint. Later, Pozzi et al. [55] extended the algorithm in [7] to include a simple input check. This input check is based on the observation that if a source node or an invalid node is the input to a pattern, then it is a permanent input and cannot be absorbed by adding in additional nodes to the pattern. Moreover, if an ancestor node of a pattern has been considered and excluded (0-branch), then the ancestor node also becomes a permanent input for the pattern. So if a pattern at a node in the search tree violates the input constraint based on the permanent inputs, then the subtree rooted at that node can also be eliminated. This new pruning criteria reduces the search space of the algorithm further.

12.3.3.2 Hierarchical Algorithm

The search-tree-based enumeration algorithm generates all the patterns within a single step. In contrast, Yu and Mitra [72, 73] proposed a multi-step algorithm that proceeds to generate all the feasible patterns in a hierarchical fashion. It breaks up the pattern generation process into three steps corresponding to cone, connected MIMO, and disjoint MIMO patterns. Cones are special kinds of patterns. A cone is a rooted DAG in the data-flow graph such that either there is a path from the root node r to every other node in the cone ($downCone(r)$) or there is a path from every other node to the root node ($upCone(r)$). An $upCone(r)$ is a MISO pattern if it has only one output. For example, $\{3, 5\}$ is an upCone rooted at 3 and it is also MISO pattern, but $\{0, 1, 2\}$ is an upCone rooted at 1 although it is not a MISO pattern. The pattern $\{1, 2, 3\}$ is a downCone rooted at 3.

The first step generates upCones and downCones. Recall that a MISO pattern is a upCone with only one output node. Therefore, the first step implicitly generates all the MISO patterns. The second step combines two or more cones to generate connected MIMO patterns, and finally the third step combines two or more cones/MIMO patterns to generate disjoint MIMO patterns. The hierarchical algorithm is based on the intuition that it is advantageous to separate out connected and disjoint MIMO pattern generation. The reason is the following. On the one hand, connected MIMO pattern generation algorithm does not need to consider nodes that are far apart and have no chance of participating in a connected pattern together. Therefore, the design space is reduced considerably. On the other hand, lots of infeasible patterns are filtered out during connected pattern generation step

and are not considered subsequently during disjoint pattern generation step. Thus the separation of concerns speeds up the algorithm substantially.

Theorem 1. *Any connected MIMO pattern C with $In(C)$ input operands and $Out(C)$ output operands can be generated by combining convex upCones with at most $In(C)$ input operands or convex downCones with at most $Out(C)$ output operands.*

In other words, it is possible to generate any feasible connected MIMO patterns by combining one or more cones. For example, the pattern $\{1, 2, 3, 5\}$ in Fig. 12.4 can be generated by combining $upCone(3) = \{3, 5\}$ with $downCone(3) = \{1, 2, 3\}$. The above theorem provides a key search space reduction technique by excluding some combination of cones. Specifically, to generate all the connected MIMO patterns, the hierarchical algorithm only needs all upCones that satisfy convexity/input constraints and all downCones that satisfy convexity/output constraints. This allows the algorithm to prune aggressively.

Theorem 2. *Any connected component C_i of a feasible disjoint pattern D_i must be a feasible connected pattern.*

This theorem states that a feasible disjoint pattern can be generated from one or more feasible connected patterns. The possible combination of feasible patterns is much smaller than that of arbitrary patterns, resulting in more efficient enumeration.

12.3.4 Exhaustive Enumeration of All Maximal Convex Patterns

The I/O constraint restricts the size of the valid patterns as either the input or the output constraint gets easily violated with increasing number of nodes in a pattern. Pothineni et al. [54] first proposed to relax the I/O constraint. It is well known that the speedup potential grows with increasing number of input and output nodes for the patterns [71]. Pothineni et al. observed that the convexity constraint is immutable, along with the exclusion of the invalid nodes in a pattern. They wanted to observe the limits of performance potential without I/O constraints. Later Verma et al. [66] formally proved that it is sufficient to consider only the maximal convex patterns.

Definition 1. A speedup model is monotonic, if for any two convex patterns C_1 and C_2

$$(V(C_1) \subseteq V(C_2)) \implies (Speedup(C_1) \leq Speedup(C_2))$$

Let $SW\_latency(C)$ and $HW\_latency(C)$ be the latency for software and hardware implementation of a pattern, respectively. Then Verma et al. [66] proved the following theorem.

Theorem 3. *The speedup model for pattern generation for RISC processor is monotonic, under the assumption that for any convex pattern* C, $SW\_latency(C) \geq HW\_latency(C)$.

The theorem indicates that under fairly weak assumptions, increasing the number of nodes in a pattern can never reduce the speedup. Consequently, the optimal pattern will also be the maximal pattern, and it is sufficient to enumerate only the maximal convex patterns.

However, the relaxed I/O constraint implies that the custom functional unit has to somehow obtain all the input and output operands from the register file. Cong et al. [17] proposed a shadow register file to overcome the limited bandwidth from the main register file. Jayaseelan and Mitra [36] leveraged the data forwarding or bypassing logic in the processor pipeline to supply additional operands to the CFU. Pozzi and Ienne [56] suggested distributing the register file accesses over multicycle, pipelined execution of the pattern in the CFU. This approach is known as I/O serialization in the literature. A number of algorithms [1, 4, 56, 66] have been proposed to appropriately schedule the I/O over multiple cycles to ensure that the convex pattern can be implemented in practice.

Pothineni et al. [54] defined the incompatibility graph as the first step toward solving the maximal convex pattern enumeration problem. Let $x \in X(G)$ be an invalid node in the DFG G. Clearly, any node $a \in Pred(x)$ cannot be involved with any node $b \in Succ(x)$ in a pattern because it will violate the convexity constraint. This is because there will be a path from a to b that involves the node x and the node x cannot be included in any pattern, violating convexity. Thus the cluster of nodes in $Pred(x)$ is incompatible with the cluster of nodes $Succ(x)$. Similar incompatibility can be defined between predecessor and successor nodes of each invalid node. For example, in Fig. 12.4, $\{0\}$ and $\{4\}$ are incompatible clusters. Then we can define the incompatibility graph as an undirected graph where there is an edge between each pair of incompatible cluster. Any convex pattern cannot include the incompatibility edges. Therefore, enumerating maximal convex subgraphs of G is equivalent to enumerating the maximal independent sets in the incompatibility graph (A set of vertices in a graph is independent if for every pair of vertices, there is no edge connecting the two. A maximal independent set is one which is not a proper subset of any independent set.). The maximal independent sets for the DFG in Fig. 12.4 are $\{0, 1, 2, 3, 5\}$ and $\{4, 1, 2, 3, 5\}$. Verma et al. [66] proved an equivalent result by defining a cluster graph, which is the complement of the incompatibility graph, and hence the maximal convex subgraphs can be enumerated by enumerating the maximal cliques in the cluster graph.

In general, this problem has exponential time complexity in the worst case, because the number of maximal independent sets of a graph with n nodes is upper bounded by $3^{n/3}$ [46]. But due to the clustering performed w.r.t. each invalid node in constructing the incompatibility graph, Atasu et al. [6] showed that the number of maximal convex patterns is $O(2^{|X(G)|})$ where $X(G)$ is the set of invalid nodes in the DFG G and can be enumerated in as many computational steps. Moreover, Reddington and Atasu [58] have proved that no polynomial-time maximal convex

pattern enumeration algorithm can exist. But by carefully choosing the order of clustering of the nodes, the enumeration can be performed quite effectively [5,6,40].

12.3.5 Enumeration of Maximum Weighted Convex Patterns

We can associate a weight with each vertex in the DFG. For example, the weight $weight(v)$ of a node v can correspond to the software latency $SW\_latency(v)$ of the operation corresponding to the node. Then the weight of a pattern C can be defined as $\sum_{v \in V(C)} weight(v)$. A pattern C is called the maximum (weighted) convex pattern if it is the maximal convex pattern with the maximum weight.

As mentioned earlier, it is not possible to design polynomial-time algorithm to enumerate all possible maximal convex patterns (as there can be exponential number of them present in a graph). But the problem of finding the maximum convex pattern is equivalent to finding the maximum independent set in the compatibility graph [58], and there exist polynomial-time solutions for this problem by further converting it into a minimum flow problem in a network.

Given a polynomial-time solution for the maximum convex pattern problem, we can design an iterative algorithm that identifies the maximum convex pattern in each iteration, removes those nodes from the DFG, and then repeats the process for the remaining nodes. Such an algorithm can cover the vertices of the DFG with a set of nonoverlapping convex patterns and has been demonstrated to generate high-quality custom instructions [5].

Recently, Giaquinta et al. [23] have studied the maximum weighted convex pattern identification problem under I/O constraint. This problem is useful when the maximal or maximum convex subgraphs might be too big to be realized in practice through I/O serialization. Including the I/O constraint from the beginning can generate feasible patterns that are implementable. At the same time, identifying maximum convex patterns under relatively large I/O constraint is more tractable than the original exhaustive enumeration of all convex patterns under I/O constraints discussed earlier. This problem requires first identifying the maximal convex patterns and then searching for the maximum weighted patterns from among this set.

12.3.6 Custom Instructions Selection

Given the set of candidate patterns, we need to first identify the identical subgraphs using graph isomorphism algorithm. All the identical subgraphs map to a single CFU or custom instruction; that is, a custom instruction has multiple instances. The execution frequencies of custom instruction instances are different and result in different performance gains. The selection process attempts to cover each original instruction in the code with zero/one custom instruction to maximize performance. Zero custom instruction covering an original instruction means that the original instruction is not included in any custom instruction. The selection of the custom instructions can be optimal or nonoptimal (heuristic). One way to optimally select

the custom instructions is by modeling it as an Integer Linear Program (ILP) and solve the ILP using an efficient ILP solver. The problem can also be solved optimally using dynamic programming or branch-and-bound methods.

12.3.6.1 Optimal Solution Using ILP

The ILP formulation presented here was originally proposed in [37] and then modified to this particular context in [37]. Let N be the number of custom instructions identified during the first step defined by $C_1 \ldots C_n$. A custom instruction C_i can have n_i different instances occurring in the program code denoted by $c_{i.1} \ldots c_{i.n_i}$. Each instance has execution frequency given by $f_{i.j}$. Let R_i be the area requirement of the custom instruction C_i and P_i be the performance gain obtained by implementing C_i in custom functional unit as opposed to software (given in number of clock cycles). The binary variable $s_{i.j}$ is equal to 1 if custom instruction instance $c_{i.j}$ is selected and 0 otherwise. The following objective function maximizes the total performance gain using custom instructions:

$$\max : \sum_{i=1}^{N} \sum_{j=1}^{n_i} (s_{i.j} \times P_i \times f_{i.j})$$

The objective function has to be optimized under the constraint that a static instruction can be covered by at most one custom instruction instance. If custom instruction instances $c_{i_1.j_1} \ldots c_{i_k.j_k}$ can all potentially cover a particular static instruction, then

$$s_{i_1.j_1} + \ldots + s_{i_k.j_k} \leq 1$$

In order to model the area constraint or the constraint on the total number of custom instructions, the variable S_i is defined. S_i is a binary variable that is equal to 1 if C_i is selected and 0 otherwise. S_i is defined in terms of $s_{i.j}$.

$$S_i = 1 \text{ if } \sum_{j=1}^{n_i} s_{i.j} > 0$$

$$= 0 \text{ otherwise}$$

However, the above equation is not a linear one. The following equivalent linear equations can model the constraint.

$$\sum_{j=1}^{n_i} s_{i.j} - U \times S_i \leq 0$$

$$\sum_{j=1}^{n_i} s_{i.j} + 1 - S_i > 0$$

where U is a large constant greater than $\max(n_i)$.

If R is the total area budget for all the CFUs, then

$$\sum_{i=1}^{N}(S_i \times R_i) \leq R$$

Similarly, if M is the constraint on the total number of custom instructions, then

$$\sum_{i=1}^{N} S_i \leq M$$

12.3.6.2 Other Approaches

As the ILP-based custom instruction selection may become computationally expensive for large number of custom instruction instances, heuristic selection algorithms are more practical. One idea is to assign priorities to the custom instruction instances. The instances are chosen starting with the highest priority one. In addition, any search heuristic such as genetic algorithm, simulated annealing, hill climbing, etc. can be applied for this problem. While most approaches consider single-objective optimization, the custom instruction selection exposes an interesting multi-objective optimization as well. Bordoloi et al. [10] proposed a polynomial-time approximation algorithm that can help the designers explore the area-performance trade-off for multi-objective optimization. The approach approximates the (potentially exponential size) set of points on the area-performance Pareto curve with only a polynomial number of points such that any point in the original Pareto curve is within ϵ distance (the value of ϵ is decided by the designer) from at least one of the selected points. Custom instruction selection problem has also been considered in the context of real-time systems [31, 45].

12.4 Run-Time Customization

Application-specific processor design as presented so far is quite promising. But it has a drawback that a new application-specific processor has to be designed and fabricated for at least each application domain, if not for each application. This is because a processor customized for one application domain may fail to provide any tangible performance benefit for a different domain. Soft core processors with extensibility features synthesized in Field-Programmable Gate Arrays (FPGAs) (e.g., Altera Nios [49], Xilinx MicroBlaze [60]) somewhat mitigate this problem as the customization can be performed post-fabrication. Still, customizable soft cores suffer from lower frequency and higher energy consumption issues because the entire processor (and not just the CFUs) is implemented in FPGAs. Apart from cross-domain performance problems, extensible processors are also limited by the amount of silicon available for the implementation of the CFUs. As embedded systems progress toward highly complex and dynamic applications (e.g., MPEG-4

video encoder/decoder, software-defined radio), the silicon area constraint becomes a primary concern. Moreover, for highly dynamic applications that can switch between different modes (e.g., run-time selection of encryption standard) with unique custom instructions requirements, a customized processor catering to all scenarios will clearly be a suboptimal design.

Run-time adaptive application-specific processors offer a potential solution to all these problems. An adaptive custom processor can be configured at run time to change its custom instructions and the corresponding CFUs. Clearly, to achieve run-time adaptivity, the CFUs have to be implemented in some form of reconfigurable logic. But the base processor is implemented in ASIC to provide high clock frequency and better energy efficiency. As CFUs are implemented in reconfigurable logic, these extensible processors offer full flexibility to adapt (post-fabrication) the custom instructions according to the requirement of the application running on the system and even midway through the execution of the application. Such adaptive custom processors can be broadly classified into two categories:

- *Explicit Reconfigurability:* This class of processors needs full compiler or programmer support to identify the custom instructions, synthesize them, and finally cluster them into one (or more) configuration that can be switched at run time. In other words, custom instructions are generated off-line, and the application is recompiled to use these custom instructions.
- *Implicit Reconfigurability:* This class of processors does not expose the extensibility feature to the compiler or the programmer. In other words, the extensibility is completely transparent to the user. Instead, the run-time system identifies the custom instructions and synthesizes them while the application is running on the system. These systems are more complex, but may provide better performance as the decisions are taken at run time.

12.4.1 Explicit Run-Time Customization

In this subsection, we focus on extensible processors that require extensive compiler or programmer intervention to achieve run-time reconfigurability.

Programmable Instruction-Set Processor (PRISC) [57] is one of the very first architectures to introduce CFU reconfigurability. The architecture supports a set of configurations, where each configuration corresponds to a computational kernel or custom instruction. There can be only one active configuration at any point in time. However, the CFU can be reconfigured at run time to support different configurations during the execution of an application or different applications. The temporal reconfigurability gives the illusion of a large CFU as multiple configurations can be supported using the same silicon, but it comes at the cost of reconfiguration overhead.

The CFU in PRISC is called Programmable Functional Unit (PFU). The PFU however is restricted in the sense that it can support only two input operands and one output operand. Thus the PFU cannot support large custom instructions that can

potentially provide significant performance benefit. Moreover, each configuration can only include one custom instruction. This effectively restricts PRISC to use only one custom instruction per loop body because it is expensive to reconfigure within a loop body to support multiple instructions.

OneChip [35] reduces the reconfiguration overhead by allowing multiple configurations to be stored in the PFU, but only one configuration can be active at any point of time. Unfortunately, OneChip does not provide enough details regarding how the programmers can specify or design the custom instructions that will be mapped onto the PFU.

Both PRISC and OneChip allow only one custom instruction per configuration. This decision leads to high reconfiguration overhead specially if multiple custom instructions need to be supported within a computational kernel executing frequently, such as the loop body. This restriction is lifted in the next set of architecture that enables both spatial and temporal reconfiguration. That is, multiple custom instructions can be part of a single configuration. This combination of spatial and temporal reconfiguration is a powerful feature that can significantly reduce the reconfiguration overhead.

The Chimaera [70] architecture is one of the first works to consider both temporal and spatial configuration of the custom functional units. The architecture is inspired by PRISC as it tightly couples reconfigurable functional unit (RFU) with a superscalar pipeline. But a crucial difference is that Chimaera RFU can use up to nine input registers to produce the result in one destination register. The architecture, however, suffers from inadequate compiler support. The compiler can automatically map a cluster of base instructions into custom instructions. However, the Chimaera compiler lacks support for spatial and temporal reconfiguration of custom instructions so as to fully exploit run-time reconfiguration.

The Stretch S6000 [25] architecture is a commercial processor that follows this trend of spatial and temporal reconfiguration. Figure 12.6 shows the Stretch S6000 engine that incorporates Tensilica Xtensa LX dual-issue Very Long Instruction Word (VLIW) processor [39] and the Stretch Instruction-Set Extension Fabric (ISEF). The ISEF is a software-configurable data path based on programmable logic. It consists of a plane of arithmetic/logic units (AU) and a plane of multiplier units (MU) embedded and interlinked in a programmable, hierarchical routing fabric. This configurable fabric acts as a functional unit to the processor. It is built into the data path of the base processor and resides alongside other traditional functional units. The programmer-defined application-specific instructions (called extension Instructions) need to be implemented in the ISEF. One or more custom instructions are combined into a configuration, and the compiler generates multiple such configurations. When an extension instruction is issued, the processor checks if the corresponding configuration containing the extension instruction is loaded into the ISEF. If not, the configuration is automatically and transparently loaded prior to the execution of the custom instruction. ISEF provides high data bandwidth to the core processor through 128-bit wide registers. In addition, 64KB embedded RAM is included inside ISEF to store temporary results of computation. With all

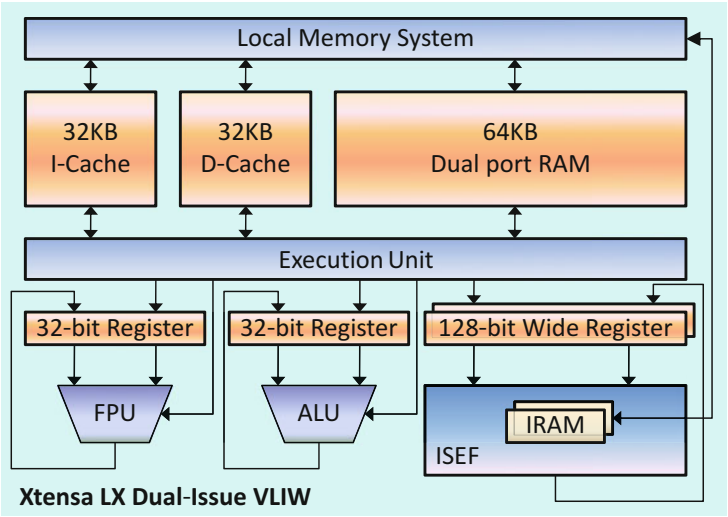

Fig. 12.6 Stretch S6000 data path [25]

these features, a single custom instruction can potentially implement a complete inner loop of the application. The Stretch compiler also fully unrolls any loop with constant iteration counts.

Most reconfigurable application-specific processors use a traditional reconfigurable fabric to implement the custom instructions or a configuration consisting of multiple custom instructions, for example, Stretch S6000 [25] architecture. This approach has the advantage of flexibility but suffers from computational inefficiency. Just-in-time customizable (JiTC) [12] architecture reconciles the conflicting demands of performance and flexibility in extensible processor. The key innovation in this architecture is a Specialized Functional Unit (SFU) tightly integrated into the processor pipeline. The SFU is a multistage accelerator that has been purpose-built to execute most common computational patterns across a range of representative applications in a single cycle. The SFU can be reconfigured on a per cycle basic to support different custom instructions in different cycles. The JiTC compiler identifies the appropriate custom instructions, generates the configuration bitstream for each such instruction to be implemented on the SFU, and replaces the selected patterns in the software binary with custom instructions. The JiTC core can thus provide near-ideal performance of an extensible processor with very little silicon area dedicated for customization. The JiTC core has recently been employed in a low-power many-core architecture [63] for wearables to enable low-cost application-specific customization at run time.

12.4.1.1 Partial Reconfiguration

So far the architecture presented requires full reconfiguration, that is, the entire fabric is reconfigured to support the next configuration. This can result in wasted

reconfiguration cost specially when there is overlap between two consecutive custom instructions. That is, only a subset of custom instructions from the current configuration should be replaced with new custom instructions. Partial reconfiguration comes to rescue in this situation as it provides the ability to reconfigure only part of the reconfigurable fabric. With partial reconfiguration, idle custom instructions can be removed to make space for the new instructions. Moreover, as only a part of the fabric is reconfigured, it saves reconfiguration cost.

Dynamic Instruction-Set Computer (DISC) [68] is one of the earlier attempts in designing an extensible processor to provide partial reconfiguration feature. DISC implements each instruction of the instruction set as an independent circuit module. Thus the individual instruction modules can be paged in and paged out onto the reconfigurable fabric in a demand-driven manner. Moreover, the circuit modules are relocatable. If needed, an existing module can be moved to a different place inside the fabric so as to create enough contiguous free space to accommodate the incoming instruction module. The drawback of DISC system is that both the base and the custom instructions are implemented in the reconfigurable logic leading to performance loss. On the other hand, the host processor remains severely underutilized as its only task is resource allocation and reconfiguration.

Extended instruction set RISC (XiRisc) [41] follows this line of development to couple a VLIW data path with a run-time reconfigurable hardware. The architecture can support four source operands and two destination operands for each custom instruction. One of the interesting developments in XiRisc is that the reconfigurable hardware can hold internal states for several cycles reducing the register pressure on the base processor. However, XiRisc did not include configuration caching leading to higher reconfiguration overhead. Also like most early reconfigurable application-specific processor, XiRisc lacked compiler support to automate the custom instruction design and reconfiguration process.

Molen [64] is an interesting polymorphic processor that incorporates an arbitrary number of reconfigurable functional units. This allows the architecture to execute two more independent custom instructions in parallel in the reconfigurable logic. To support the functionality of the reconfigurable fabric, eight custom instructions are added to the instruction set. Molen requires a new programming paradigm where general-purpose instructions and hardware descriptions of custom instructions co-exist in a program. Molen compiler automatically generates optimized binary code from applications specified in C programming language with pragma annotation for custom instructions. The architecture hides the reconfiguration cost through scheduling where the configuration corresponding to a custom instruction is pre-fetched before the instruction is executed.

12.4.1.2 Compiler Support

Compiler support is instrumental in ensuring greater adoption of application-specific processors by software designers. Unfortunately, as mentioned earlier, most of the run-time reconfigurable application-specific processors suffer from

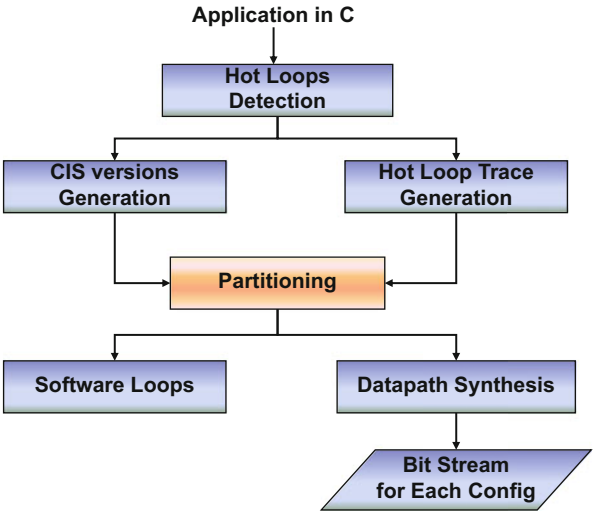

Fig. 12.7 Compiler framework for run-time adaptive extensible processors [33]

inadequate compiler assistance. The burden falls entirely on the programmer to select appropriate the custom instructions and cluster them into one or more configurations. Choosing an appropriate set of custom instructions for an application itself is a difficult problem as discussed in Sect. 12.3. Run-time reconfiguration introduces the additional complexity of the temporal and spatial partitioning of the selected custom instructions into a set of configurations.

Huynh et al. [33] developed an efficient compilation framework that takes in as input an application specified in ANSI-C and automatically selects appropriate custom instructions as well as bundles them together into one or more configurations as shown in Fig. 12.7. First, a profiling step identifies and extracts a set of compute-intensive candidate loop kernels from the application. For each candidate loop, one or more Custom Instruction-Set (CIS) versions are generated (e.g., by changing loop unrolling factor) differing in performance gain and area trade-offs. The control flows among the hot loops are captured in the form of a loop trace obtained through profiling. The hot loops with multiple CIS versions and the loop trace are fed to the partitioning algorithm that decides the appropriate CIS version and configuration for each loop. The algorithm models the temporal partitioning of the custom instructions into different configurations as a k-way graph partitioning problem. A dynamic programming-based pseudo-polynomial-time algorithm determines the spatial partitioning of the custom instructions within a configuration. The selected CIS versions to be implemented in hardware pass through a data-path synthesis tool generating the bitstream corresponding to each configuration. These bitstreams are used to configure the reconfigurable fabric at run time. The source code is modified to exploit the new custom instructions while the remaining code executes on the base processor.

12.4.2 Implicit Run-Time Customization

We now proceed to describe extensible processors that are reconfigured transparently by the run-time system.

Configurable Compute Accelerator (CCA) [15] enables transparent instruction-set customization support through a plug-and-play model that can integrate different accelerators into a predesigned and verified processor core at run time. The compiler framework comprises of static identification of subgraphs for execution on CCA [16]. This is supplemented with run-time selection of custom instructions to be synthesized to CCA. First, the program is analyzed to identify the most frequent computation patterns (custom instructions) to be mapped onto CCA. These patterns are replaced by function calls in the binary code. At run time, when the function corresponding to a custom instruction is encountered for the first time, it executes in the base processor pipeline. But, in parallel, the architecture determines the CCA configuration required for this particular custom instruction. When the same function is encountered again in the future, it can execute on the CCA using the generated configuration.

Unlike CCA that requires compiler-architecture cooperation, the WARP [42] architecture has been designed with completely transparent instruction-set customization in mind. WARP processor consists of a base core, an on-chip profiler, WARP-oriented FPGA, and an on-chip Computer-Aided Design (CAD) module. An application starts executing on the base processor. The on-chip profiler identifies the critical hot-spot kernels (loops) during the execution of the application. These kernels are then passed onto the riverside on-chip CAD (ROCCAD) tool chain through the on-chip CAD module. ROCCAD tool chain decompiles the application binary into high-level representation that is more suitable for synthesis. Next, the partitioning algorithm determines the most suitable loops to be implemented in FPGA. For the selected kernels, ROCCAD uses behavioral and register transfer level (RTL) synthesis to generate appropriate hardware descriptions. Finally, ROCCAD configures the FPGA using just-in-time FPGA compilation tools that optimizes the hardware description and performs technology mapping followed by place and route to map the hardware description onto the reconfigurable fabric. Finally, the application binary is updated to be used to kernels mapped onto the FPGAs.

A unique approach toward run-time customization is proposed in the Rotating Instruction-Set Processing Platform (RISPP) [8] architecture. RISPP introduces the notion of atoms and molecules where atom is the basic data path, while a combination of atoms creates custom instruction molecule. Atoms can be reused across different custom instruction molecules. RISPP reduces the overhead of partial reconfiguration substantially through an innovative gradual transition of the custom instructions implementation from software into hardware. At compile time, only the potential custom instructions (molecules) are identified, but these molecules are not bound to any data path in hardware. Instead, a number of possible implementation choices are available including a purely software implementation. At run time, the implementation of a molecule can gradually "upgrade" to hardware

as and when the atoms it needs become available. If no atom is available for a custom instruction, it will be executed in the base processor pipeline using the software implementation. RISPP requires fast design space exploration at run time to combine appropriate atoms and evaluate trade-offs between performance and area of the custom instructions implementations. A greedy heuristic selects the appropriate implementation for each custom instruction.

12.5 Custom Instructions for General-Purpose Computing

As mentioned earlier, the design goals for specialization in the context of general-purpose applications are somewhat different. The added custom instructions should support a large number of computations (some unknown at design time) so that only a few of them are sufficient to cover significant fraction of execution of a diverse set of applications.

An example of this approach is the specialized processors called quasi-specific cores (QsCores) proposed by Venkatesh et al. [65]. QsCores have been proposed in the context of dark silicon era where the cheap silicon area can be traded in to accommodate few QSCores. Each QsCore is an application-specific processor that can accelerate a set of computations through custom instructions. The main insight here is that there exist nearly identical code fragments within and across applications. These "similar" code fragments can be represented by a single computational pattern that is implemented as a custom instruction, thereby leading to reuse. Unlike the computation patterns we introduced before, QsCores support large hot spot containing hundreds of instructions, complex control flows, and irregular memory accesses as a single pattern as prescribed by Hameed et al. [30]. This enables the QsCores to be an order of magnitude more energy-efficient than general-purpose cores. In the general form of this architecture, a general-purpose processor core is coupled with a number of QsCores – each accelerating different computations – to create a heterogeneous tiles. The entire chip consists of a number of heterogeneous tiles, each responsible for different workloads.

A different approach is taken by Govindaraju et al. [26] where the main idea is to dynamically specialize the hardware according to the phases within an application. They introduce dynamically specialized data paths called DYnamically Specialized Execution Resource (DySER) blocks. The DySER blocks are similar to the CFUs and are integrated in the processor pipeline as additional functional units. Each block is a heterogeneous array of computational units interconnected with a circuit-switched mesh network; but unlike QsCores, there is no memory access involved within DySER block. A DySER block uses specific computational units (arithmetic and logical operations) depending on the common instruction mix of the applications. By interconnecting these operations through the network, a computational pattern can be mapped to the DySER block. The compiler partitions the application into phases, identifies the most frequently executed paths within each phase, and then maps the computations corresponding to each of these paths on

DySER blocks. Note that the identification of computational patterns as discussed in Sect. 12.3 has been restricted to within basic blocks, but Yu and Mitra had quantized the benefit of crossing basic blocks boundaries and generating custom instructions spanning multiple basic blocks along hot paths [71]. DySER reaps these benefits through a concrete architectural design and implementation. There are some similarities between DySER and coarse-grained reconfigurable arrays (CGRAs) [14]. But the main difference is that CGRAs accelerate complete loops, while DySER focuses on computation within a hot path and does not support control flow or load/store that is required to map an entire loop. The other difference lies in using computational units that are decided based on instruction mix rather than generic functional units used in CGRAs and the use of circuit-switched network. The main strength of DySER is that the same specialized hardware can accelerate different applications and diverse domains through dynamic specialization.

Gupta et al. [27] proposed a configurable coprocessor called Bundled Execution of REcurring Traces (BERET) that can leverage recurring instruction sequences in a program's execution. The instruction sequence may include intervening control instructions because of the irregularity of general-purpose code. Essentially each sequence is a hot trace that forms a loop, but is much shorter compared to the original unstructured loop body. Similar to other application-specific processor approaches for general-purpose computing, BERET also aims to support multiple applications. The architecture is based on the observation that the hot trace can be broken down into a sequence of subgraphs that can execute sequentially, while exploiting parallelism and chaining within subgraph to improve performance (as is common in any custom instruction). The concept of subgraph is called bundled execution model in this approach. The observation and insight is that many subgraph structures or patterns are common within as well as across applications. Therefore, if the architecture supports some common subgraphs, any hot trace can be mapped to a series of these subgraphs for acceleration.

12.6 Conclusions

In this chapter, we presented the current state of the art in the application-specific processor design. The application-specific processors, also known as customizable processors or specialized cores, present an exciting alternative in today's energy-constrained design space. We discussed the opportunities and challenges presented by this special class of processors and the progress made in automated design of such cores over the last decade. The renewed interest in application-specific processors for general-purpose computing and even supercomputing domain have opened up interesting new research directions, both in terms of architecture and compiler, that we hope will be pursued extensively in the coming decade.

Acknowledgments This work was partially supported by Singapore Ministry of Education Academic Research Fund Tier 2 MOE2014-T2-2-129.

References

1. Ahn J, Choi K (2013) Isomorphism-aware identification of custom instructions with i/o serialization. IEEE Trans Comput-Aided Des Integr Circuits Syst 32(1):34–46
2. Alippi C, Fornaciari W, Pozzi L, Sami M (1999) A dag-based design approach for reconfigurable VLIW processors. In: Proceedings of the conference on design, automation and test in Europe. ACM, p 57
3. Amdahl GM (1967) Validity of the single processor approach to achieving large scale computing capabilities. In: Proceedings of the spring joint computer conference, 18–20 Apr 1967. ACM, pp 483–485
4. Atasu K, Dimond RG, Mencer O, Luk W, Özturan C, Diindar G (2007) Optimizing instruction-set extensible processors under data bandwidth constraints. In: Design, automation & test in Europe conference & exhibition, DATE'07. IEEE, pp 1–6
5. Atasu K, Luk W, Mencer O, Özturan C, Dündar G (2012) Fish: fast instruction synthesis for custom processors. IEEE Trans Very Large Scale Integr (VLSI) Syst 20(1):52–65
6. Atasu K, Mencer O, Luk W, Özturan C, Dündar G (2008) Fast custom instruction identification by convex subgraph enumeration. In: International conference on application-specific systems, architectures and processors, ASAP 2008. IEEE, pp 1–6
7. Atasu K, Pozzi L, Ienne P (2003) Automatic application-specific instruction-set extensions under microarchitectural constraints. Int J Parallel Program 31(6):411–428
8. Bauer L, Shafique M, Kramer S, Henkel J (2007) Rispp: rotating instruction set processing platform. In: Proceedings of the 44th annual design automation conference. ACM, pp 791–796
9. Bonzini P, Pozzi L (2007) Polynomial-time subgraph enumeration for automated instruction set extension. In: Proceedings of the conference on design, automation and test in Europe. EDA Consortium, pp 1331–1336
10. Bordoloi UD, Huynh HP, Chakraborty S, Mitra T (2009) Evaluating design trade-offs in customizable processors. In: 46th ACM/IEEE design automation conference, DAC'09. IEEE, pp 244–249
11. Borkar S, Chien AA (2011) The future of microprocessors. Commun ACM 54(5):67–77
12. Chen L, Tarango J, Mitra T, Brisk P (2013) A just-in-time customizable processor. In: 2013 IEEE/ACM international conference on computer-aided design (ICCAD). IEEE, pp 524–531
13. Chen X, Maskell DL, Sun Y (2007) Fast identification of custom instructions for extensible processors. IEEE Trans Comput-Aided Des Integr Circuits Syst 26(2):359–368
14. Choi K (2011) Coarse-grained reconfigurable array: architecture and application mapping. IPSJ Trans Syst LSI Des Methodol 4:31–46
15. Clark N, Blome J, Chu M, Mahlke S, Biles S, Flautner K (2005) An architecture framework for transparent instruction set customization in embedded processors. In: Proceedings of the 32nd international symposium on computer architecture (ISCA'05). IEEE Computer Society, pp 272–283
16. Clark N, Kudlur M, Park H, Mahlke S, Flautner K (2004) Application-specific processing on a general-purpose core via transparent instruction set customization. In: 37th international symposium on microarchitecture, MICRO-37 2004. IEEE, pp 30–40
17. Cong J, Fan Y, Han G, Jagannathan A, Reinman G, Zhang Z (2005) Instruction set extension with shadow registers for configurable processors. In: Proceedings of the 2005 ACM/SIGDA 13th international symposium on field-programmable gate arrays. ACM, pp 99–106
18. Cong J, Fan Y, Han G, Zhang Z (2004) Application-specific instruction generation for configurable processor architectures. In: Proceedings of the 2004 ACM/SIGDA 12th international symposium on field programmable gate arrays. ACM, pp 183–189
19. Dennard RH, Gaensslen FH, Rideout VL, Bassous E, LeBlanc AR (1974) Design of Ion-implanted MOSFET's with very small physical dimensions. IEEE J Solid-State Circuits 9(5):256–268

20. Dubach C, Jones T, O'Boyle M (2007) Microarchitectural design space exploration using an architecture-centric approach. In: Proceedings of the 40th annual IEEE/ACM international symposium on microarchitecture. IEEE Computer Society, pp 262–271
21. Esmaeilzadeh H, Blem E, St Amant R, Sankaralingam K, Burger D (2011) Dark silicon and the end of multicore scaling. In: International symposium on computer architecture (ISCA)
22. Geer D (2005) Chip makers turn to multicore processors. Computer 38(5):11–13
23. Giaquinta E, Mishra A, Pozzi L (2015) Maximum convex subgraphs under i/o constraint for automatic identification of custom instructions. IEEE Trans Comput-Aided Des Integr Circuits Syst 34(3):483–494
24. Gonzalez RE (2000) Xtensa: a configurable and extensible processor. IEEE Micro 20(2):60–70
25. Gonzalez RE (2006) A software-configurable processor architecture. IEEE Micro 26(5):42–51
26. Govindaraju V, Ho CH, Sankaralingam K (2011) Dynamically specialized datapaths for energy efficient computing. In: 2011 IEEE 17th international symposium on high performance computer architecture (HPCA). IEEE, pp 503–514
27. Gupta S, Feng S, Ansari A, Mahlke S, August D (2011) Bundled execution of recurring traces for energy-efficient general purpose processing. In: Proceedings of the 44th annual IEEE/ACM international symposium on microarchitecture. ACM, pp 12–23
28. Gutin G, Johnstone A, Reddington J, Scott E, Yeo A (2012) An algorithm for finding input–output constrained convex sets in an acyclic digraph. J Discret Algorithms 13:47–58
29. Halambi A, Grun P, Ganesh V, Khare A, Dutt N, Nicolau A (2008) Expression: a language for architecture exploration through compiler/simulator retargetability. In: Design, automation, and test in Europe. Springer, The Netherlands, pp 31–45
30. Hameed R, Qadeer W, Wachs M, Azizi O, Solomatnikov A, Lee BC, Richardson S, Kozyrakis C, Horowitz M (2010) Understanding sources of inefficiency in general-purpose chips. In: ACM SIGARCH computer architecture news, vol 38, no 3. ACM, pp 37–47
31. Huynh H, Mitra T (2007) Instruction-set customization for real-time embedded systems. In: Proceedings of the conference on design, automation and test in Europe. EDA Consortium, pp 1472–1477
32. Huynh HP, Mitra T (2009) Runtime adaptive extensible embedded processors–a survey. In: International workshop on embedded computer systems. Springer, Berlin/Heidelberg, pp 215–225
33. Huynh HP, Sim JE, Mitra T (2007) An efficient framework for dynamic reconfiguration of instruction-set customization. In: Proceedings of the 2007 international conference on compilers, architecture, and synthesis for embedded systems. ACM, pp 135–144
34. Ienne P, Leupers R (2006) Customizable embedded processors: design technologies and applications. Academic Press
35. Jacob JA, Chow P (1999) Memory interfacing and instruction specification for reconfigurable processors. In: Proceedings of the 1999 ACM/SIGDA seventh international symposium on field programmable gate arrays. ACM, pp 145–154
36. Jayaseelan R, Liu H, Mitra T (2006) Exploiting forwarding to improve data bandwidth of instruction-set extensions. In: Proceedings of the 43rd annual design automation conference. ACM, pp 43–48
37. Kastner R, Kaplan A, Memik SO, Bozorgzadeh E (2002) Instruction generation for hybrid reconfigurable systems. ACM Trans Des Autom Electron Syst (TODAES) 7(4):605–627
38. Kathail V, Aditya S, Schreiber R, Rau BR, Cronquist DC, Sivaraman M (2002) Pico: automatically designing custom computers. Computer 35(9):39–47
39. Leibson S (2006) Designing SOCs with configured cores: unleashing the tensilica Xtensa and diamond cores. Academic Press
40. Li T, Sun Z, Jigang W, Lu X (2009) Fast enumeration of maximal valid subgraphs for custom-instruction identification. In: Proceedings of the 2009 international conference on compilers, architecture, and synthesis for embedded systems. ACM, pp 29–36

41. Lodi A, Toma M, Campi F, Cappelli A, Canegallo R, Guerrieri R (2003) A VLIW processor with reconfigurable instruction set for embedded applications. IEEE J Solid-State Circuits 38(11):1876–1886
42. Lysecky R, Stitt G, Vahid F (2004) Warp processors. In: ACM transactions on design automation of electronic systems (TODAES), vol 11, no 3. ACM, pp 659–681
43. Merritt R (2009) ARM CTO: power surge could create 'dark silicon'. EE Times, Oct 2009.
44. Mitra T (2015) Heterogeneous multi-core architectures. Inf Media Technol 10(3):383–394
45. Mitra T, Yu P (2005) Satisfying real-time constraints with custom instructions. In: Third IEEE/ACM/IFIP international conference on hardware/software codesign and system synthesis, CODES+ ISSS'05. IEEE, pp 166–171
46. Moon JW, Moser L (1965) On cliques in graphs. Israel J Math 3(1):23–28
47. Moore GE et al (1965) Cramming more components onto integrated circuits
48. Mudge T (2000) Power: a first class design constraint for future architectures. In: International conference on high-performance computing. Springer, pp 215–224
49. Nios I (2009) Processor reference handbook
50. Palacharla S, Jouppi NP, Smith JE (1997) Complexity-effective superscalar processors. In: Proceedings of the 24th annual international symposium on computer architecture (ISCA'97), Denver. ACM, New York, pp 206–218. doi: 10.1145/264107.264201
51. Palermo G, Silvano C, Zaccaria V (2005) Multi-objective design space exploration of embedded systems. J Embed Comput 1(3):305–316
52. Pan Y (2008) Design methodologies for instruction-set extensible processors. Ph.D. thesis, National University of Singapore
53. Patterson D, Hennessy JL (2012) Computer architecture: a quantitative approach. Elsevier
54. Pothineni N, Kumar A, Paul K (2007) Application specific datapath extension with distributed i/o functional units. In: Proceedings of the 20th international conference on VLSI design, Bangalore
55. Pozzi L, Atasu K, Ienne P (2006) Exact and approximate algorithms for the extension of embedded processor instruction sets. IEEE Trans Comput-Aided Des Integr Circuits Syst 25(7):1209–1229
56. Pozzi L, Ienne P (2005) Exploiting pipelining to relax register-file port constraints of instruction-set extensions. In: Proceedings of the 2005 international conference on compilers, architectures and synthesis for embedded systems. ACM, pp 2–10
57. Razdan R (1994) Prisc: programmable reduced instruction set computers. Ph.D. thesis, Harvard University Cambridge
58. Reddington J, Atasu K (2012) Complexity of computing convex subgraphs in custom instruction synthesis. IEEE Trans Very Large Scale Integr (VLSI) Syst 20(12): 2337–2341
59. Reddington J, Gutin G, Johnstone A, Scott E, Yeo A (2009) Better than optimal: fast identification of custom instruction candidates. In: International conference on computational science and engineering, CSE'09. vol 2. IEEE, pp 17–24
60. Rosinger HP (2004) Connecting customized ip to the microblaze soft processor using the fast simplex link (fsl) channel. Xilinx Application Note
61. Shafique M, Garg S, Mitra T, Parameswaran S, Henkel J (2014) Dark silicon as a challenge for hardware/software co-design. In: Conference on hardware/software codesign and system synthesis (CODES)
62. Shalf JM, Leland R (2015) Computing beyond moore's law. Computer 48(12):14–23
63. Tan C, Kulkarni A, Venkataramani V, Karunaratne M, Mitra T, Peh LS (2016) Locus: low-power customizable many-core architecture for wearables. In: Proceedings of the international conference on compilers, architecture, and synthesis for embedded systems (CASES)
64. Vassiliadis S, Wong S, Gaydadjiev G, Bertels K, Kuzmanov G, Panainte EM (2004) The molen polymorphic processor. IEEE Trans Comput 53(11):1363–1375
65. Venkatesh G, Sampson J, Goulding-Hotta N, Venkata SK, Taylor MB, Swanson S (2011) Qscores: trading dark silicon for scalable energy efficiency with quasi-specific cores. In: Proceedings of the 44th annual IEEE/ACM international symposium on microarchitecture. ACM, pp 163–174

66. Verma AK, Brisk P, Ienne P (2007) Rethinking custom ise identification: a new processor-agnostic method. In: Proceedings of the 2007 international conference on compilers, architecture, and synthesis for embedded systems. ACM, pp 125–134
67. Wall DW (1991) Limits of instruction-level parallelism. In: Proceedings of the fourth international conference on architectural support for programming languages and operating systems (ASPLOS IV), Santa Clara. ACM, New York, pp 176–188. doi: 10.1145/106972.106991
68. Wirthlin MJ, Hutchings BL (1995) A dynamic instruction set computer. In: IEEE symposium on FPGAs for custom computing machines. Proceedings. IEEE, pp 99–107
69. Wulf WA, McKee SA (1995) Hitting the memory wall: implications of the obvious. ACM SIGARCH Comput Archit News 23(1):20–24
70. Ye ZA, Moshovos A, Hauck S, Banerjee P (2000) CHIMAERA: a high-performance architecture with a tightly-coupled reconfigurable functional unit. In: ACM SIGARCH computer architecture news, vol 28, no 2. ACM, pp. 225–235
71. Yu P, Mitra T (2004) Characterizing embedded applications for instruction-set extensible processors. In: Proceedings of the 41st annual design automation conference. ACM, pp 723–728
72. Yu P, Mitra T (2004) Scalable custom instructions identification for instruction-set extensible processors. In: Proceedings of the 2004 international conference on compilers, architecture, and synthesis for embedded systems. ACM, pp 69–78
73. Yu P, Mitra T (2007) Disjoint pattern enumeration for custom instructions identification. In: International conference on field programmable logic and applications, FPL 2007. IEEE, pp 273–278