Universiteit Gent
Faculteit Ingenieurswetenschappen
en Architectuur
Vakgroep Elektronica en Informatiesystemen

# Automated Design of Domain-Specific Custom Instructions

Geautomatiseerd ontwerp van domeinspecifieke gespecialiseerde instructies

---

Cecilia González-Álvarez

Promotoren: Prof. Dr. Ir. Lieven Eeckhout
Prof. Dr. Daniel Jiménez-González
Prof. Dr. Carlos Álvarez

Proefschrift tot het behalen van de graad van
Doctor in de Ingenieurswetenschappen: Computerwetenschappen
Academiejaar 2015-2016

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
**Departament d'Arquitectura de Computadors**

# Automated Design of Domain-Specific Custom Instructions

Diseño automatizado de instrucciones especializadas para un dominio específico

---

Cecilia González-Álvarez

Directores: Prof. Dr. Lieven Eeckhout
Prof. Dr. Daniel Jiménez-González
Prof. Dr. Carlos Álvarez

Tesis presentada para obtener el título de
Doctora por la Universitat Politècnica de Catalunya
Programa de Doctorado: Arquitectura de Computadores
Año académico 2015-2016

# Acknowledgements

I would like to express my sincere gratitude to my advisors: Daniel Jiménez-González and Carlos Álvarez at UPC, and Lieven Eeckhout at UGent. They have guided me with expertise and understanding, and without their support this thesis would have never been possible.

I would also like to thank people I met at each step of my PhD journey. First, I thank my supervisor at BSC, Xavier Martorell for his guidance and support, and my colleagues at BSC and UPC for all the great moments shared. Also, a very special thanks goes to Hironori Kasahara, professor at Waseda University, for all the encouragement during the year and a half that I spent in Japan. At last, I also would like to thank my colleagues at UGent, with special thanks to Jennifer Sartor for her help and support.

Of course, I would like to thank my family and friends for all the best moments shared that lighted up the darkest times. Special thanks go to my parents and Klaas for their loving encouragement.

Finally, this research would not have been possible without the financial assistance of the Severo Ochoa program (SEV-2011-00067), the Spanish Ministry of Science and Technology (TIN2012-34557), the Generalitat de Catalunya (MPEXPAR, 2009-SGR-980), HiPEAC3 Network of Excellence (FP7/ICT 287759), the European Research Council under the European Community's Seventh Framework Programme (FP7/2007-2013) / ERC Grant agreement no. 259295, the Xilinx University Program, and the Japanese Ministry of Education. I express my gratitude to those agencies.

*Ghent, 24/09/2015*
*Cecilia González Álvarez*

# Summary

In the last years, hardware specialization has received renewed attention as chips approach a utilization wall. Specialized accelerators can take advantage of underutilized transistors implementing custom hardware that complements the main processor. However, specialization adds complexity to the design process and limits reutilization. Application-Specific Instruction Processors (ASIPs) balance performance and reusability, extending a general-purpose processor with custom instructions (CIs) specific for an application, implemented in Specialized Functional Units (SFUs). Still, time-to-market is a major issue with application-specific designs because, if CIs are not frequently executed, the acceleration benefits will not compensate for the overall design cost. Domain-specific acceleration increases the applicability of ASIPs, as it targets several applications that run on the same hardware. Also, reconfigurable SFUs and the automation of the CIs design can solve the aforementioned problems. In this dissertation, we explore different automated approaches to the design of CIs that extend a baseline processor for domain-specific acceleration to improve both performance and energy efficiency.

First, we develop automated techniques to identify code sequences within a domain to create CI candidates. Due to the disparity among coding styles of different programs, it is difficult to find patterns that are represented by a unique CI across applications. Therefore, we propose an analysis at the basic block level that identifies equivalent CIs within and across different programs. We use the Taylor Expansion Diagram (TED) canonical representation to find not only structurally equivalent CIs, but also functionally similar ones, as opposed to the commonly applied directed acyclic graph (DAG) isomorphism detection. We combine both methods into a new Hybrid DAG/TED technique to identify more patterns across applications that map to the same CI. Then, we select a subset of the CI candidates that fits in the available SFU area. Because of the complexity of the problem, we propose four scoring heuristics to reduce the design space and smooth the potential performance speedup across applications. We include these me-

thods in the FuSInG framework, and we estimate performance with hardware models on a set of media benchmarks. Results show that, when limiting core area devoted to specialization, the SFU customization with the largest speedups includes a mix of application and domain-specific custom instructions.

If we target larger CIs to obtain higher speedups, reusability across applications becomes critical; without enough equivalences, CIs cannot be generalized for a domain. We aim to share partially common operations among CIs to accelerate more code, especially across basic blocks, and to reduce the hardware area needed for specialization. Hence, we create a new canonical representation across basic blocks, the Merging Diagram, to facilitate similarity detection and improve code coverage. We also introduce clustering-based partial matching to identify partially-similar domain-specific CIs, which generally leads to better performance than application-specific ones. Yet, at small areas, merging two CIs induces a high additional overhead that might penalize energy-efficiency. Thus, we also detect fragments of CIs and we join them with the existing merged clusters resulting in minimal extra overhead. Also, using speedup as the deciding factor for CI selection may not be optimal for devices with limited power budgets. For that reason, we propose a linear programming-based selection that balances performance and energy consumption. We implement these techniques in the MInGLE framework and evaluate them with media benchmarks. The selected CIs significantly improve the energy-delay product and performance, demonstrating that we can accelerate a domain covering more code while reducing the needed area for the CI implementation.

# Resumen

La especialización de hardware ha recibido renovado interés debido al utilization wall, ya que transistores infrautilizados pueden implementar hardware a medida que complemente el procesador principal. Sin embargo, el proceso de diseño se complica y se limita la reutilización. Procesadores de instrucciones para aplicaciones específicas (ASIPs) equilibran rendimiento y reuso, extendiendo un procesador con instruciones especializadas (custom instructions – CIs) para una aplicación, implementadas en unidades funcionales especializadas (SFUs). No obstante, los plazos de comercialización suponen un obstáculo en diseños específicos ya que, si las CIs no se ejecutan con frecuencia, los beneficios de la aceleración no compensan los costes de diseño. La aceleración de un dominio específico incrementa la aplicabilidad de los ASIPs,acelerando diferentes aplicaciones en el mismo hardware, mientras que una SFU reconfigurable y un diseño automatizado pueden resolver los problemas mencionados. En esta tesis, exploramos diferentes alternativas al diseño de CIs que extienden un procesador para acelerar un dominio, mejorando el rendimiento y la eficiencia energética.

Proponemos primero técnicas automatizadas para identificar código acelerable en un dominio. Sin embargo, la identificación se ve dificultada por la diversidad de estilos entre diferentes programas. Por tanto, proponemos identificar en el bloque básico CIs equivalentes utilizando la representación canónica Taylor Expansion Diagram (TED). Con TEDs encontramos no sólo código estructuralmente equivalente, sino también con similitud funcional, en contraposición a la detección isomórfica de grafos acíclicos dirigidos (DAG). Combinamos ambos métodos en una nueva técnica híbrida DAG/TED, que identifica en varias aplicaciones más secuencias representadas por la misma CI. Tras esto, seleccionamos un subconjunto de CIs que puede ser contenido en el área de la SFU. Por la complejidad del problema, proponemos cuatro heurísticas de selección para reducir el espacio de búsqueda y homogeneizar el rendimiento de las aplicaciones. Incluimos estas técnicas en la infraestructura FuSInG y estimamos el rendimiento para un conjunto de benchmarks multimedia. Los resultados muestran que, al

limitar el área de especialización, la configuración de la SFU con las mayores ganancias incluye una mezcla de CIs específicas tanto para una aplicación como para todo el dominio.

Si nos centramos en CIs más grandes para obtener mayores ganancias, la reutilización se vuelve crítica; sin suficientes equivalencias las CIs no pueden ser generalizadas. Nuestro objetivo es que las CIs compartan parcialmente operaciones, especialmente a través de bloques básicos, y reducir el área de especialización. Por ello, creamos una representación canónica de CIs que cubre varios bloques básicos, Merging Diagram, para mejorar el alcance de la aceleración y facilitar la detección de similitud. Además, proponemos una búsqueda de coincidencias parciales basadas en clustering para identificar CIs de dominio específico parcialmente similares, las cuales derivan generalmente mejor rendimiento. Pero en áreas reducidas, la fusión de CIs induce un coste adicional que penalizaría la eficiencia energética. Así, detectamos fragmentos de CIs y los unimos con grupos de CIs previamente fusionadas con un coste extra mínimo. Usar el rendimiento como el factor decisivo en la selección puede no ser óptimo para disposivos con consumo de energía limitado. Por eso, proponemos un mecanismo de selección basado en programación lineal que equilibra rendimiento y consumo energético. Implementamos estas técnicas en la infraestructura MInGLE y las evaluamos con benchmarks multimedia. Las CIs seleccionadas mejoran notablemente la eficiencia energética y el rendimiento, demostrando que podemos acelerar un dominio cubriendo más código a la vez que reducimos el área de implementación.

# Samenvatting

In de afgelopen jaren heeft hardwarespecialisatie opnieuw aandacht gekregen omdat chips de *utilization wall* naderen, door de vertraging van het schalen van de voedingsspanning. Gespecialiseerde acceleratoren kunnen profiteren van onderbenutte energiezuinige transistors door de implementatie van aangepaste hardware die de hoofdprocessor aanvullen. Echter, specialisatie verhoogt de complexiteit van het ontwerpproces en beperkt de flexibiliteit wat betreft circuit hergebruik. Applicatie-Specifieke Instructieset Processors (ASIPs) houden rekening met zowel prestaties als flexibiliteit. Ze breiden een "general-purpose processor" uit met aangepaste instructies voor specifieke toepassingen, geïmplementeerd in Specialized Functional Units (SFUs). Het ontwerpproces kan vereenvoudigd worden met geautomatiseerde technieken die gespecialiseerde instructies (Eng: Custom Instructions – CIs) identificeren, selecteren en implementeren. De "time-to-market" is echter een groot probleem bij applicatie-specifieke ontwerpen. Indien CIs niet vaak worden uitgevoerd wegen de voordelen van versnelling niet op tegen de totale ontwerpkosten. Domeinspecifieke versnelling verhoogt de toepasbaarheid van ASIPs, als ze zich kunnen richten op verscheidene toepassingen die op dezelfde hardware in dezelfde tijdsperiode uitgevoerd worden. In dit proefschrift onderzoeken we verschillende benaderingen wat betreft het ontwerp van de CIs die een *baseline processor* voor domeinspecifieke acceleratie uitbreiden om de herbruikbaarheid te verhogen en om zowel de prestatie als energie-efficiëntie te verbeteren.

Allereerst ontwikkelen we geautomatiseerde technieken om codesequenties die versneld kunnen worden tussen verschillende toepassingen binnen een domein te identificeren. Iedere onafhankelijke sequentie is en nieuwe CI kandidaat die op een SFU kan uitgevoerd worden. CIs die zich richten op een volledig domein zijn een veelbelovende optie. Het verschil in codeerstijlen van verschillende programma's maakt het echter moeilijk om patronen in code te identificeren die door een unieke CI kunnen worden vervangen. Bijgevolg stellen we een analyse op het basisblok voor die gelijkwaardige CIs binnen hetzelfde programma en over de verschillende programma's

heen herkent. We gebruiken hiervoor het Taylor Expansion Diagram (TED) om niet alleen structureel maar ook functioneel gelijkwaardige stukken code te vinden, in tegenstelling tot de vaak toegepaste direct acyclische grafiek (DAG) isomorfisme detectie. We combineren ook beide methodes in een nieuwe hybride DAG/TED-techniek. Dit helpt ons meer sequenties te identificeren uit meerdere toepassingen die op dezelfde CI gemapt kunnen worden. Hierdoor kunnen hogere versnellingen bereikt worden in een kleinere chip area.

Vervolgens, om grotere versnellingen te verkrijgen, richten we ons op CIs die grotere codesequenties dan een basisblok versnellen. Maar als de beschikbare oppervlakte voor de implementatie beperkt is, wordt herbruikbaarheid van de hardware een kritieke factor. Echter, als we niet genoeg overeenkomsten over toepassingen vinden, kunnen CIs niet gegeneraliseerd worden voor een domein. Wij streven ernaar, specifiek tussen basisblokken, om gedeeltelijk gemeenschappelijke operaties te delen tussen CIs om de code die versneld kan worden uit te breiden en om de hardware ruimte die nodig is voor specialisatie te verminderen. Daarom creëren we een nieuwe canonieke representatie van CIs doorheen basisblokken: het *Merging Diagram*. Deze nieuwe representatie verbetert gelijkheidsdetectie en codedekking ten opzichte van eerdere methoden. Ook introduceren we *clustering-based matching* die gecombineerd word in samengevoegde CIs om oppervlakte voor implementatie te besparen. Deels samengevoegde domein-specifieke CIs leiden over het algemeen tot betere prestaties dan applicatie-specifieke CIs. Toch zorgt op kleine oppervlaktes het samenvoegen van twee extra CIs voor hoge overheadkosten die energie-efficiëntie kunnen verminderen. Zodus sporen we ook fragmenten van CIs op die slechts gedeelten van eerder gegenereerde en/of samengevoegde CIs bevatten. Wij verbinden de fragmenten van CIs met de bestaande samengevoegde clusters, wat resulteert in minder oppervlakte overhead dan volledige CI samenvoegen. Deze technieken zorgen voor het uitbreiden van de versnellingsmogelijkheden van CIs, omdat er meer code wordt versneld terwijl de ruimte die nodig is voor de CI-implementatie verkleind wordt.

Bij de laatste stap van het CI-ontwerp, wordt een selectie gemaakt van subsets van CIs die passen in de beschikbare SFU-ruimte. De complexiteit van het probleem en de grote CI ontwerpruimte voor een volledig toepassingsdomein maakt de selectie rekenkundig uitdagend. Wij stellen vier score heuristieken voor om snel en efficiënt de ontwerpruimte van CI kandidaten te doorzoeken. De heuristieken rekenen het verwacht gezamenlijk gebruik van CIs binnen en over verschillende programma's uit, met als doel de po-

tentiële speedup doorheen applicaties te bereiken, waardoor ze geschikt zijn voor domeinspecifieke versnelling. Het gebruiken van de versnelling als de beslissende factor voor CI selectie kan echter niet optimaal zijn voor toestellen met een beperkte vermogensvoorraad. Daarom introduceren we ook energie-efficiëntie tot een nieuwe parameter om rekening mee te houden. We stellen een op geheeltallig programmering gebaseerd selectie mechanisme voor dat streeft naar een evenwicht tussen versnelling en energieverbruik om een hoog-performante en energiezuinige CI-configuratie te selecteren.

Wij bouwen het exploratie raamwerk FuSInG om codesequenties te extraheren en te analyseren met DAG, TED en Hybrid identificatiemethoden, en om hen te ordenen met score heuristieken. De algemene prestatie wordt geschat op basis van hardware-modellen doorheen het gehele spectrum van applicatie-specifieke en domein-specifieke versnelling in hardware, gebruik makend van een aantal multimedia benchmarks. We evalueren onze canonieke representaties voor het ontwerp van domein-specifieke CIs, en we tonen dat zij essentieel zijn om meer gelijkstellingen dan structurele voorstellingen zoals DAGs over toepassingen te vinden. We onderzoeken ook de tradeoffs van verschillende SFU configuraties om de prestatie van het volledige systeem met beperkt oppervlakte over toepassingen te optimaliseren. Resultaten tonen aan dat, wanneer de beschikbare chip oppervlakte voor specialisatie beperkt is, de SFU aanpassing met de grootste speedups een mix bevat van applicatie- en domeinspecifieke instructies. Daarnaast hebben we de resultaten cross-valideerd om te tonen dat de geïdentificeerde CIs effectief zijn voor ongeziene toepassingen binnen hetzelfde domein, waarmee specialisatie meer algemeen toepasbaar maakt.

Ook creëren we MInGLE, een geautomatiseerd raamwerk dat voorlopige CIs herkent, en deze tot *Merging Diagrams* transformeert. Dit framework voegt, afhankelijk van hun gelijkaardigheidsscore, CIs samen. Het framework selecteert, op basis van de geheeltallig programmering techniek, CI configuraties die efficiënt de beschikbare ruimte voor specialisatie benutten. Experimentele resultaten met een set van media benchmarks tonen aan dat, gemiddeld, versnelling over basisblokken betere speedup en *energie-delay product* (EDP) verbetering bereikt dan over een enkele basisblok (speedup van 1.98× versus 1.48×, EDP verbetering van 3.35× versus 1.67×). Ook over basisblokken, gedeeltelijke *matching* bereikt betere speedup en EDP verbetering in vergelijking met exacte matching, gegeven dezelfde oppervlakte (bijv., voor 1.8% van de oppervlakte, speedup van 1.88× versus 1.73× en EDP verbetering van 3.04× versus 2.53× ). Bovendien, het uit-

gebreide MInGLE+ raamwerk identificeert, extraheert en selecteert fragmenten van CIs. Matching met fragmenten gebruikt oppervlakte effectiever dan de gedeeltelijke matching (bijv. voor 1% van de oppervlakte, 2× versus 1.63× en EDP verbetering van 3.65× versus 2.35×). De geselecteerde CIs verbeteren de prestatie en energie-efficiëntie van toepassingen uit het multimedia domein aanzienlijk. Dit toont aan dat we een applicatiedomein efficiënt kunnen versnellen met gedeeltelijk gekoppelde CIs.

# Table of Contents

# List of Figures

# List of Tables

# List of Acronyms

| | |
|---|---|
| ARM | Advanced RISC Machines |
| ASIP | Application-Specific Instruction Processor |
| ASIC | Application-Specific Integrated Circuit |
| BDD | Binary Decision Diagram |
| CFG | Control Flow Graph |
| CI | Custom Instruction |
| CMOS | Complementary Metal-Oxide-Semiconductor |
| DAG | Directed Acyclic Graph |
| DFG | Data Flow Graph |
| DMA | Direct Memory Access |
| DSFU | Domain-Specific Functional Unit |
| EDP | Energy-Delay Product |
| FPGA | Field Programmable Gate Array |
| GPU | Graphics Processing Unit |
| HLS | High Level Synthesis |
| INF | If-then-else Normal Form |
| IR | Intermediate Representation |
| ISA | Instruction Set Architecture |

| | |
|---|---|
| ISEF | Instruction Set Extension Fabric |
| ITE | If Then Else (operator) |
| MILP | Mixed Integer Linear Programming |
| MISO | Multiple Input Single Output |
| MIMO | Multiple Input Multiple Output |
| RISC | Reduced Instruction Set Computing |
| ROBDD | Reduced Ordered Binary Decision Diagram |
| SFU | Specialized Functional Unit |
| SMT | Simultaneous Multithreading |
| SRAM | Static Random-Access Memory |
| SSA | Static Single Assignment |
| TED | Taylor Expansion Diagram |
| VLIW | Very Long Instruction Word |

# 1 Introduction

## 1.1. Motivation and Context

The steady increase of processor speed that Moore's law had been dictating since 1965 [1] was jeopardized when the limits of dimensional scaling started to raise concern among chip makers. The doubling in number of transistors per chip every one and a half to two years that Moore forecasted, and that yields to higher performing circuits, was conceptualized by Dennard [2]. His scaling law establishes that, keeping the electrical field constant, if the chip dimensions scale down, the integration density of transistors on a chip increases, causing circuits to be faster and to reduce power consumption. The constant field scaling paradigm governed the microelectronics industry due to its continuous delivery of higher performance with lower power consumption at lower costs in each semiconductor process generation. However, since the beginning of this century, there has been a slowdown of the energy per transistor switch scaling due to technology limitations. This has been marked as the end of Dennard scaling: voltage scaling cannot keep up with transistor scaling. Now, at each new process generation, integration density increases, but so does the static power leakage. All the transistors on a chip are not powered at the same time to avoid thermal runaway, resulting in an under-utilization of the chip [3–5], also known as *dark silicon*. Seeing that the times of Dennard scaling are over, we cannot count anymore on power improvements based on traditional technology

advances.

Meanwhile, nowadays market trends demand, more than ever, low-power processors that do not sacrifice performance. Societal needs shape a technological future with energy-efficient intelligent systems integrated in any conceivable gadget. Devices will become increasingly sophisticated, with richer functionalities compared to existing ones, and demanding more optimized solutions. By way of example, smartphone production overtook that of client PCs in 2011; in that year, there were 73 million more smartphones shipped than PCs and tablets together. Also, in May 2015 Google announced that searches on mobile devices surpassed PCs in the US for the first time, and in the UK, mobile handsets now account for 56% of time spent on the Internet. In view of this unstoppable takeover of size-constrained machines, we ask whether the market can keep up with the demand of high functionality with low-power budgets in the advent of *dark silicon*.

Ultimately, we should look for a down-scaled technology that can be efficiently used in modern application domains. The benefits of Moore's shrinking rates could continue without Dennard's rules; with equivalent scaling [6] we use other means than dimensional scaling to maintain improvements in speed and energy. For instance, we can propose new architectures that provide special-purpose functionalities and that are heterogeneously integrated with current processors. Specialized or custom computers are not novel; since the first specialized computer, over 50 years ago [7], the implementation of specific computing units has been extensively studied. But it is now when their benefits over traditional, general-purpose architectures, are becoming more popular. Specialization is seen as a way to cope with the *dark silicon* problem, by increasing the energy efficiency that a low-power budget imposes.

Consequently, custom computing as off-core accelerators, GPUs or in-core functional units is a hot topic. Current generation of mobile processors already integrate heterogeneous chips combined with accelerators, which are also becoming more common in server and desktops. Even supercomputer engineers pay now attention to accelerator-based systems; the first three supercomputers in the June 2015 Green500 list[1] rely on special-purpose acceleration with their PEZY-SC cores to provide high energy efficiency. Also, customized computing is in the spotlight of European public funds for research and innovation. The European Commission, under the Horizon 2020 program[2] and the Joint Technology Initiative on Electronic Compo-

[1] http://www.green500.org/lists/green201506

[2] http://ec.europa.eu/research/participants/portal/desktop/en/opportunities/h2020/topics/9080-ict-04-2015.html

nents and Systems[3], has granted more than €100 million in 2015 for the development of the next generation of CPUs together with customized and low-power computing. New custom architectures across several application domains are key targets for the European Commission to reinforce Europe's technological competitiveness.

Approaches to custom computing are diverse, therefore accelerating systems may vary depending on the target application domain. Focusing on mobile devices, we still find a plethora of different types of applications that demand performance within constrained power budgets, from common multimedia tasks to voice or facial recognition. Embedded systems equipped with specialized hardware can increase performance and reduce energy consumption, but the implementation details of the customized section are nonetheless a hard choice. For instance, we can obtain high performance accelerating critical parts of an application with Application-Specific Integrated Circuits (ASICs); however, their design cost is expensive, as they lack the flexibility of being programmable. Options that allow hardware reutilization, such as reconfigurable technologies, still need a considerable effort from the designer, which may compromise the total cost and time-to-market of the final product.

Ideally, specialization in embedded systems should yield performance and energy gains close to those of an ASIC, while being flexible and reusable with minimum overhead. New micro-architectural approaches that use pre-built building blocks are our proposal: extensible processors, also known as Application-Specific Instruction Processors (ASIPs) [8], balance performance and flexibility, and yet maintain the energy efficiency gains of specialization. As they reuse a pre-verified and pre-optimized base processor, the design process is less complex and time-to-market is shorter. The classical ASIP design process augments a general-purpose processor with specialized functional units (SFUs) that execute instructions customized for a particular application. This design process can be automated to identify, select and implement those **custom instructions (CIs)**, and the focus of this dissertation lays, precisely, in those automated methods for CI design.

## 1.2. Custom Instruction Design

Extending processors with SFUs not only reduces the design costs of acceleration. Other benefits of using CIs include the minimization of the cycles spent in the prediction, fetch, decode, scheduling and commit stages of a processor. For each data and instruction fetch, they can perform from

[3] `http://www.ecsel-ju.eu/web/index.php`

tens to hundreds of operations, cutting down the processor's energy waste. Also, the deployment of CIs is more effective than specializing a complete processor as they are easier to program than bigger off-core accelerators.

SFU customization is the process of discovering new CIs that accelerate the target applications. Within a small and controlled application domain, CIs can be manually detected by studying limited benchmark code, combining frequently executed bundles of operations into one CI. However, with real applications, manual exploration is not an option; the benefits of CIs would not compensate time and effort in design. Therefore, SFU customization is normally an automated process integrated as an alternative path of an application's compilation flow. Automatic CI discovery has attracted extensive attention as research topic since it is far from being a trivial process.

The design of CIs can be broken down into three different phases: the discovery or generation of CIs, the implementation of CIs, and the substitution of generic code by CIs. Out of those three, CI generation is the most important and difficult phase and hence it is our focus here; it is the focus as well for most of the related work. CI generation examines the application code's data-flow graph (DFG) and identifies subgraphs of operations as special instructions. Typically within a single basic block, these subgraphs join tens of operations into a single CI to maximize the overall speedup. Their reusability is commonly very limited; a CI is found at a concrete point of a single application, making them essentially application-specific. CI generation is done in two steps: candidate identification and final selection. First, during candidate identification, subgraphs are identified under architectural constraints. This exploration can take exponential time complexity, thus algorithms to convey this problem are a recurring topic in the literature. Then, the final selection finds the best set of CIs that maximizes the performance in a limited area. How this problem is attacked is also relevant, since it is NP-complete. Therefore, the complete CI design process has substantial research interest, as the acceleration benefits of SFUs can be hard to obtain.

## 1.3. Key Challenges

To accelerate a processor with SFUs, we are compelled to provide simple and fast design methods of CIs that improve performance. However, the general adoption of such a customizing technology depends on a variety of factors: how applicable is the CI generation in a broad context, how energy side-effects are taken into consideration, or how an efficient use of the area is ensured. These points challenge the CI design process and must be adequately addressed.

First, note that the increasing market demand in consumer electronics

imposes strict time-to-market constraints. Also, systems with accelerators are in constant change, since the software that runs on them is regularly modified; therefore, users may want to accelerate applications that were not considered at design time. ASIPs allow programmability to a certain extent to amortize chip design costs, but the common trend is to design CIs for an individual application. As application-specific CIs are not highly reusable, they are sentenced to a short life span at the expense of high investments. In addition, if CIs are not frequently executed, the acceleration benefits will not compensate for the overall energy consumption. Thus, extending the CIs usage to a whole domain of applications can increase the suitability of ASIPs for acceleration. We can then find similar CIs in different programs that can be implemented as only one instruction, improving reusability and economizing hardware resources. But although applications within the same domain often perform similar computations that require the same hardware, we are confronted with two issues: first, exploring several applications at once results in a design space explosion, and second, non-uniform programming styles of different codes may hide their underneath similarities.

Over the past few years, advances in automatic high-level synthesis (HLS) have enabled rapid prototyping of accelerators, which results into another important issue: varying the optimization options, we can obtain many configurations of the same CI. We can have again a design space explosion. Exploration of the different architectural configurations then becomes a sensible task, and automated ways for curbing the design space are key to be able to find a configuration solution that delivers good performance.

While performance speedup is the key metric that traditionally drives CI design, if we only look at performance we may downplay other equally important aspects. Embedded processors have a constrained power budget to watch; therefore, CI design should attempt to balance performance gains with energy efficiency. Furthermore, albeit the SFU hardware area is limited, the expected speedup partially depends on the code coverage of the CI; bigger CIs may deliver better speedups, but all may not fit in the available area. Thus, when several applications are competing for the same resources, we need to be able to prioritize CIs without penalizing overall efficiency. Also, code coverage is subject to the kind of operations executed on the SFU; if branch or predication instructions are left out, as is commonly the case, CIs can only span within a basic block. Small CIs can be counter-productive for performance, but they also have a strong advantage over bigger ones: their reusing degree across applications may be higher and, on average, they yield a more balanced global improvement. All these

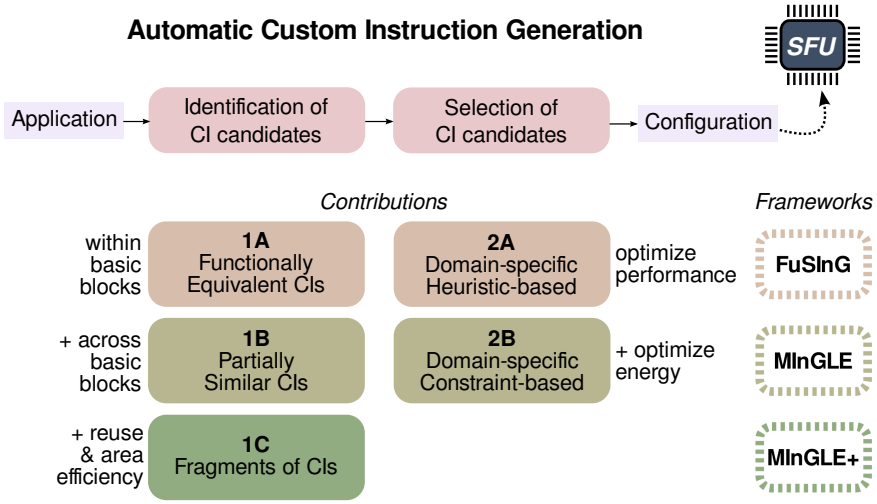


*Figure 1.1: The automated process for CI generation and the key contributions of this dissertation.*

trade-offs have to be carefully weighed to achieve full efficiency in every sense of the word.

## 1.4. Key Contributions

In this dissertation, we explore different approaches for CI generation, focusing on domain-specific acceleration. Figure 1.1 shows, on the top, a high-level schema of the CI generation process; an application is analyzed to identify CI candidates, and a subset of the best candidates is selected to configure the SFU. The figure also shows, on the bottom, our key contributions; we expand and improve the identification in contributions 1-A, B and C, and we provide alternatives to the candidate selection problem in contributions 2-A and B. The contributions are implemented within these frameworks: FuSInG (Functionally Similar Instructions Generator), which implements 1-A and 2-A; MInGLE (Merged Instructions Generator for Large Efficiency), which implements 1-B and 2-B, and MInGLE+, expanded with 1-C.

### Contribution 1-A: Identification of functionally equivalent custom instructions

Limiting ASIPs to application-specific acceleration makes them feasible for only big-volume markets with high returns. Therefore, creating CIs that target a whole domain is an economically viable solution. However, the disparity among coding styles of different programs makes it difficult to identify code patterns that can be represented by a unique CI across applications.

We therefore propose an analysis at the basic block level, complementary to CI candidate identification, that identifies equivalent CIs within the same program and across different programs. We use the Taylor Expansion Diagram (TED) canonical representation to identify common sections of code that can be accelerated by specialized hardware. As TEDs are canonical, we can find not only structurally equivalent pieces of code, but also functionally similar ones. Functional equivalence reveals if CIs perform the same mathematical function, which cannot be guessed with a DFG-based representation. We compare them to a straightforward technique of directed acyclic graph (DAG) isomorphism detection, which essentially reveals whether graphs are similar in shape. We also introduce a new Hybrid DAG/TED technique that combines the best of the traditional graph isomorphism with TEDs. With any of the three techniques, we can identify CI candidates that are specific for either an application or a domain. We find that, with the canonical representation, we can identify more sequences across applications that are mapped to the same CI, thus achieving higher speedups for smaller chip area than the traditionally used DAGs.

### Contribution 1-B: Identification of partially similar custom instructions

Hardware reusability across applications is a critical factor to achieve high and balanced speedups with CIs, yet if there are not enough equivalences, CIs cannot be generalized for a domain. This issue arises more frequently when we target larger code sections to obtain higher performance.

Consequently, we introduce a new canonical representation of CIs across basic blocks, the Merging Diagram, to facilitate similarity detection and improve the code coverage of our CIs. It builds upon the previous Hybrid DAG/TEG representation to provide a more compact representation with predication, spanning CIs across basic blocks. We also propose clustering-based partial matching of code sequences to identify not only those CIs that are functionally equivalent, but also those with partial similarities. Clustered CIs are merged and we quantify their potential improvement of cove-

ring more code while reducing the needed area for the CI implementation. These techniques expand the opportunity for CIs with a limited area budget inside simple processors to accelerate numerous applications from a domain, improving the system's energy efficiency.

### Contribution 1-C: Identification of custom instruction fragments

We have observed that partially-similar domain-specific CIs outperform application-specific ones when the area for implementation is over a given threshold. However, at small areas, we have to rely on application-specific CIs, since the potential gains of merging two CIs do not compensate for the involved overhead.

To solve this, we extend CI merging with an analysis step that detects parts of CIs that can use the existing merged clusters with minimal extra overhead. We call CI fragments to those parts of CIs, since they do not include the full original CI as in the partially similar CIs, but only sections of it. With CI fragments we can improve reutilization of hardware at the most limited areas, because we partially reuse an already merged CI cluster, with minimum additional overhead.

### Contribution 2-A: Domain-specific heuristic-based selection

At the last step of any CI generation process, the final selection tries to choose a subset of CIs that fits within the available area. The objective of getting an optimal CI group is an NP-complete problem, and thus there exists no known fast solution for it. Also, the CI design space of a whole application domain is big enough to make the task computationally challenging. Typically, approximate algorithms or heuristics are used to solve the selection fast. The main drawback of existing methods is that they target speedup for individual applications. Therefore, their suitability for a whole domain is rather limited, since the gains must be balanced to be fair across applications.

To bridge this gap, we propose four scoring heuristics to quickly and effectively cull the huge CI design space. These heuristics rank potential CIs under the premise that we aim to smoothen the obtained gains across applications. We evaluate them and give insight about their suitability for domain-specific acceleration.

### Contribution 2-B: Domain-specific energy-efficient selection

Having proved that the CI selection is appropriate for a domain of applications, we go a step further by introducing energy efficiency into the equation. Speedup is an excellent metric to select a set of CIs that accelerate the applications, hence it has been extensively used in the literature. However, for devices with limited power budgets, focusing only on performance can be detrimental for the design, since we may be introducing power-hungry CIs as well.

This last contribution proposes a constraint-based selection mechanism that, with a novel objective function, balances speedup and energy consumption to fulfill the goal of a energy-efficient design with good performance. We then solve the problem of choosing an energy-efficient set of CIs to fit in limited area while accelerating a domain harmoniously.

## 1.5. Key Results

We summarize here the key results obtained with the frameworks that implement the contributions: FuSInG, MInGLE and MInGLE+.

### FuSInG

We combine the techniques for functional equivalence identification and the scoring heuristics in our automated framework FuSInG, which also estimates performance and area of new acceleration designs. With the framework, we explore the trade-offs between application-specific and domain-specific hardware specialization. Results expose the following insights:

- TED and Hybrid DAG/TED representations identify more hardware acceleration opportunities across applications that are mapped to the same CI, which results in higher speedups for lower area than the traditionally used DAGs.

- While using only application-specific CIs results in the highest possible speedups at unbounded core areas, it is ineffective at small areas. Instead, including domain-specific CIs in the configuration produces the highest possible speedup at small, more realistic core areas, which underlines the importance of identifying CIs that can be shared across applications.

- New applications inside a domain can also benefit from CIs already designed for that domain. This suggests that processors with domain-specific functional units can extend their lifetime, making specialization more generally applicable.

### MInGLE

Partial matching of Merging Diagrams expands the acceleration opportunities for domain-specific CIs with a limited area budget, improving the system's energy efficiency. We implement these techniques in our automated framework MInGLE, and we evaluate them with applications from the media domain. We obtain the following key results:

- CIs that cover code beyond the basic block level expand the acceleration opportunities, achieving a maximum average speedup of 1.98× and an energy-delay product (EDP) improvement of 3.35×, a significant gain over CIs within a single basic block (speedup of 1.48× and EDP improvement of 1.67×).

- Partial matching and merging of CIs is crucial for achieving larger speedup and EDP improvement for a limited hardware area. For instance, for 1.8% of the area, the EDP improvement of partial matching reaches 3.04×, higher than a exact matching CI configuration (2.53×).

### MInGLE+

We extend the analysis in MInGLE+ to detect fragments of CIs that can use the existing merged clusters with minimal area overhead. These are the outcomes of this extension:

- CI fragments increase the share-out of the circuit components on an SFU at better rate than partial matching, which results in more implementation area available. This means that we achieve a particular energy efficiency at greatly reduced hardware area.

- CI fragments are key to get high performance and energy efficiency at the smallest areas. For example, for 1% of the area, CIs with fragments achieve, on average, a speedup of 2× and an EDP improvement of 3.6×, significantly higher than results for partially matched CIs (speedup of 1.6× and EDP improvement of 3.6×)

## 1.6. Publications

The above contributions and results are gathered in several international journals and conference proceedings.

Contributions 1-A and 2-A, with focus on the identification and selection of functionally equivalent CIs for an application domain, were published in:

> C. González-Álvarez, J. B. Sartor, C. Álvarez, D. Jiménez-González, and L. Eeckhout. "Accelerating an application domain with specialized functional units". *ACM Transactions on Architecture and Code Optimization (TACO)*, Vol 10, No 4, January 2014.

Contributions 1-B and 2-B, that go a step further identifying and selecting domain-specific partially-similar energy-efficient CIs, were published in:

> C. González-Álvarez, J. B. Sartor, C. Álvarez, D. Jiménez-González, and L. Eeckhout. "Automatic Design of Domain-Specific Instructions for Low-Power Processors". *Proceedings of the 26th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2015. Best student paper award.

Contribution 1-C, that extends previous contributions to allow maximum efficiency at small hardware areas, will be published in:

> C. González-Álvarez, J. B. Sartor, C. Álvarez, D. Jiménez-González, and L. Eeckhout. "MInGLE: An Efficient Framework for Domain Acceleration using Low-Power Specialized Functional Units". *ACM Transactions on Architecture and Code Optimization (TACO)*, 2015. Under review.

### 1.6.1. Other research activities

In addition to the publications above, we list here other research activities that are not included in this dissertation. These include studies of automatic parallelization for heterogeneous multicores, as well as automatically generated accelerators integration in multicore systems, using the OSCAR source-to-source compiler and runtime. We refer to the original articles for more information:

> C. González-Álvarez, Y. Kanehagi, K. Takemoto, Y. Kishimoto, K. Muto, H. Mikami, A. Hayashi, K. Kimura, H. Kasahara. "Automatic

parallelization with OSCAR API Analyzer: a cross-platform performance evaluation". *IPSJ SIG Notes*, Dec 2012. Information Processing Society of Japan (IPSJ).

C. González-Álvarez, H. Ishikawa, A. Hayashi, D. Jiménez-González, C. Álvarez, K. Kimura and H. Kasahara. "Automatic design exploration framework for multicores with reconfigurable accelerators". *7th HiPEAC Workshop on Reconfigurable Computing (WRC 2013)*.

K. Kimura, C. González-Álvarez, A. Hayashi, H. Mikami, M. Shimaoka, J. Shirako, H. Kasahara, "OSCAR API v2.1: Extensions for an advanced accelerator control scheme to a low-power multicore API", *17th Workshop on Compilers for Parallel Computing (CPC2013)*, Lyon, France, Jul. 2013.

## 1.7. Overview

This dissertation is organized as follows.

In Chapter 2, we present the necessary background information on hardware acceleration. We give an overview of relevant design techniques and accelerator architectures found in the literature, narrowing the research focus to processors with specialized functional units that execute CIs. We provide the design details of an extended processor that we use in our experiments, and we close the chapter with additional background information on code representations.

We present two key contributions in Chapter 3. First, we introduce the techniques to identify equivalent CIs that can be clustered together. Then, we explain a set of heuristics developed for domain-specific selection. Lastly, we evaluate the presented methodologies for a concrete application domain and present insights on the trade-off of application-specific and domain-specific acceleration.

In Chapter 4, we introduce two more key contributions. We firstly describe the methodology behind the identification of partially-similar CIs, and secondly, the selection mechanism to choose the most efficient configuration of CIs.

We introduce in Chapter 5 our last contribution. We explain the concept of fragments of CIs, presenting the techniques to extract and implement them, and we demonstrate their effectiveness comparing the results against previously proposed methods.

Finally, we present the conclusions of this dissertation in Chapter 6 and we discuss possible future research directions.

# 2
# Background

## 2.1. Introduction

Hardware acceleration, in its many forms, has emerged as a solution to the demands of high performance and low power in the embedded market. A cost-effective approach is to extend a baseline processor with specialized hardware and its ISA with new custom instructions. The hardware is augmented as functional units, tightly-coupled to the processor's datapath, or coprocessors, intra or outer-core, working as a slave of the main processor.

Datapath specialization can be approached from different perspectives, depending on the focus of the problem. First, from the top-down point of view of creating hardware starting from a set of target applications, we can consider automated design CIs, also known as instruction set extensions. Generally, methods explore the target applications' critical code and translate them to hardware under some constraints. We review those exploration techniques in Section 2.2. From another angle, a bottom-up process involves creating the configurable accelerator design based on architectural expertise. The focus is on creating specialized hardware, possibly configurable, in a less automated way, and then map applications on the hardware to accelerate them. Those architectural proposals are surveyed in Section 2.3, with a specific focus on tightly-coupled accelerators.

Independently from the perspective, hardware acceleration is a complex process that involves more than exploration and architectural design. Once

the design is ready, the actual circuit implementation is a step necessary to prove the suitability of the accelerator. In the latest years, high level synthesis (HLS) programs, such as Vivado HLS [9], have contributed to speed up the implementation step. Different compilation phases are also involved, such as instruction selection once the hardware is created. We do not cover that step in this dissertation, assuming that we can annotate code to substitute lines of code by the accelerating CI. However, Section 2.4 includes background information on compilers' intermediate representations (IRs) and other coding models that are relevant for the upcoming chapters.

## 2.2. Custom Instruction Design

There are many techniques for CI design from the ISA extension perspective that target different objectives and architectures. We provide here a comprehensive survey of evolution of the field through the most important works found in the literature.

The first work on the topic [10] proposes to partition the main problem of instruction generation into *regularity extraction* and *template matching*, which we have previously introduced as generation of CIs, and the substitution of generic code by CIs. They do not provide any implementation of their ideas neither in hardware nor in simulation, but estimate quantitatively the gains depending on the instruction types. In the first description [11] of CI operations types MISO (Multiple Input Single Output) and MIMO (Multiple Input Multiple Output), the authors discuss architectural constraints, such as register ports, in their identification of new instructions within the VLIW compiler Trimaran. Later work [12, 13] formally establish the analysis using Data Flow Graphs (DFG), and the importance of preserving graph convexity. The research focus shifts to the design process problem and clearly separates **identification** and **selection** phases.

Reducing the algorithmic complexity of the design methods is a priority to make the program tractable. Some works rely on heuristics [14] to predict a CI's gain as a function of the instruction's frequency of execution and latency, and on dynamic programming to optimize area usage. Pipelining techniques [15] allow CIs with more inputs and outputs than ports in the register file. A later work by the same authors [16] couple the identification and selection phases, which results in relaxing the constraints and opens up the possibility of approximate techniques and genetic algorithms that are computationally less expensive. Their CIs have any number of outputs, and are evaluated with a software latency model using the hardware measurements of CMOS operators. Others [17] assume that the core processor must be a RISC, which also relaxes constraints. This implies a limited number

of inputs and outputs, which prunes the results, in order to minimize the number of registers used.

A different work for the application-specific embedded market [18] assumes additional storage inside an ARM-based ASIP that allows DMA for some vectors. They select CIs with a merit function based on a latency estimation of memory accesses as a model of SRAM. In their experiments with decoder, filter and encryption applications the CIs are simulated in Simplescalar.

Breaking with the application-specific CI design trends, the generation of domain-specific CIs [19] involves a pattern-matching approach on the data-flow graph using heuristics. They define guide functions for a greedy search that prune the exploration space, using the criticality of the datapath, latency and area as metrics. Most notably, they combine instruction candidates to generalize the accelerators for a simulated VLIW architecture within Trimaran. Also integrated in Trimaran, ASIP extensions for multimedia and cryptography applications based on identification of maximal convex subgraphs within a basic block [20], could be adapted for domain-specific acceleration, since they group graphs that can be implemented with the same hardware and estimate their gain to choose the most promising one. The concept of maximal convex subgraph, or the maximum code coverage that we can get in a basic block without violating any constraint, is further studied [21]. They propose a fast CI identification algorithm [21] based on binary search, that we adapt in our domain-specific framework described in Chapter 3.

Other approaches to solve the CI design problem include applying integer linear programming [22, 23] or constraint programming [24] at only the CI identification, or at any of the CI design steps. Alternatively, other authors [25] apply a predefined set of rules, in a specific order, to obtain a DAG representation of code functionality instead of focusing on the structure of CI subgraphs, which is related to the techniques we see in this dissertation.

## 2.3. Datapath Accelerators

A common accelerator classification [26] categorizes architectures according to their size and proximity to the CPU. On one hand, **loosely-coupled systems**, or **coprocessors**, accelerate coarse-grained tasks with low interaction with the rest of the program, and imply a manual approach to hardware/software partitioning. An example of such a loosely-coupled system is GARP [27], in which a MIPS processor invokes special instructions that run on a custom coprocessor, outside the main core. On the other hand, **tightly-coupled systems**, or **specialized functional units** (SFUs),

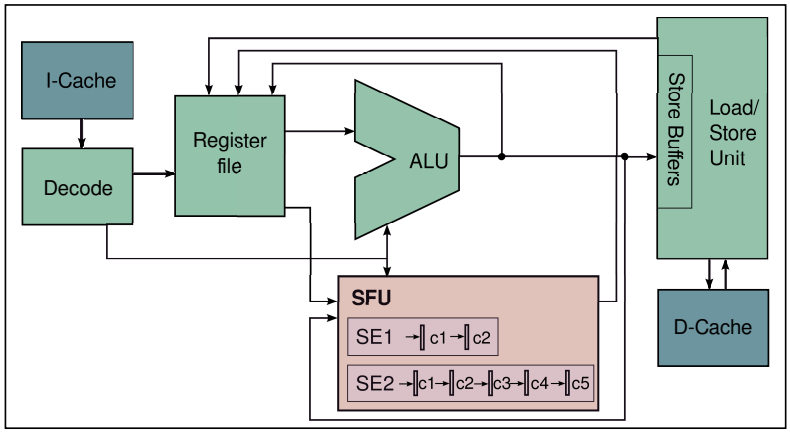


*Figure 2.1: Target architecture. The specialized functional unit (SFU) is part of the execution pipeline of an in-order processor core.*

accelerate finer-grained tasks and interact directly with the processor flow. Their programming approach is more automated and target a wider range of applications. Our generic architecture falls in the latter category, as it executes the CIs on an SFU that is tightly integrated in the datapath of an in-order general-purpose processor (see Figure 2.1). Our target architecture is a single-issue in-order processor with a configurable pipeline to execute CIs. Each CI runs inside one specialized execution (SE) pipeline of the SFU and takes a variable number of cycles ($c$). The SFU is multi-cycle and reads and writes data from and to the register file of the core. We do not consider parallel execution of the SFU with the processor's functional units because it has been proven that the performance improvement is not significant enough [28]. Benefits of such a design include a system that maintains precise interrupts, the reduction of instructions in the execution pipeline of the processor core, and the increment of operational and data-level parallelism in the SFU. Those benefits have been studied in the literature, and there are numerous works that implement tightly-coupled accelerating systems.

We find the first proposals of customizable processors with tightly-coupled accelerators [28–30] at the end of the $20^{\text{th}}$ century. PRISC [29] is a RISC processor extended with a programmable functional unit. In this architecture, the specialized unit is placed as an additional functional unit in the RISC pipeline and performs combinatorial operations using the processor register file for data transfers. The hardware is responsible for updating the configuration of the programmable unit when a CI requests it. CIs are implemented using a preamble of the RISC instruction format. OneChip [28, 30] also proposes an integrated reconfigurable architecture on a MIPS-like

processor. It extends the PRISC concept to allow pipelining in the programmable functional unit. The first OneChip version [30] is implemented on a prototyping board to test the feasibility of the design. A later work [28] extends OneChip as a RISC superscalar processor allowing dynamic scheduling and reconfiguration, and it is simulated. Chimaera [31] is another example of a tightly-coupled reconfigurable unit that extends a superscalar processor. It is able to perform 9-input 1-output integer operations with the support of a compilation chain that identifies groups of instructions that can run in their reconfigurable functional unit. They provide also subword-parallelism as an attempt to introduce data-level parallelism to their system.

Tensilica's Xtensa processor [32], based on a single-issue RISC and from the late 90's, fills the gap of commercial customizable processors. Although designers could choose different configurations adding new instructions, functional units, register files, peripherals and memory interfaces, any customization had to be done before manufacturing. Xtensa is used as the implementation base for other customizable architectures, such as Stretch's software-configurable processor [33]. It combines a RISC core with an instruction-set extension fabric (ISEF) that interchanges operands through the register file. To program it, the compiler identifies functions that are annotated with pragmas and generates the code to load and execute predefined bitstreams for the ISEF. Nios II from Altera [34] is another example of a commercial customizable processor. It is a soft processor that allows up to 256 custom instructions [35] and virtually unlimited hardware accelerators. While custom instructions are integrated within the processor pipeline, hardware accelerators work as coprocessors.

XiRisc [26] is a load/store architecture with a pipelined run-time configurable datapath called PiCoGa. The PiCoGa is integrated in the processor pipeline, and is connected to the register file, with the possibility of 4-input 2-output registers. The configuration of the PiCoGa can be dynamically scheduled at run time, and some configurations may be already stored inside to avoid configuration overhead times. Another project with a tightly-coupled RISC-based processor is CUSTARD [36]. It features a customizable multithreaded processor with different parameterizations beyond CIs, such as the number of threads, the threading type or the datapath bitwidth. They provide a cycle-accurate simulator to evaluate the application-specific optimizations applied with their compiler CoSy.

DySER [37] accelerates applications by extracting computation that is then run on an accelerating functional unit network, tightly coupled with the processor of choice, such as OpenSPARC [38]. They aim to improve both performance and energy efficiency specializing for concrete applications, providing basic control flow inside the special units, and applying

vectorization.

Broadening the definition of tightly-coupled systems, Beret [39], although not completely integrated in the datapath of an ARM processor, presents an execution engine that is still inside the core. Their Trimaran-derived toolchain extracts execution pipelines from small loop bodies based on application trace analysis. They aim to accelerate wider code sections than the above works, and focus on reducing energy consumption in general-purpose computing.

Despite that there exist CIs with memory support [40], the SFU of our target architecture is connected to the processor's register file to simplify the design and to not increase energy consumption beyond the processor's baseline, as many other works also do [26, 29, 30, 36, 38]. In this thesis, we also do no consider runtime configuration issues [41, 42], since the main problem we solve is centered in the CI design process for an application domain. We do not consider other adaptable parts in the microarchitecture [36] except the SFUs. Instead of annotating the code to be accelerated with pragmas [28, 33], we use, as other works do [29–31, 36, 38], a compilation chain that automatically identifies the CIs that run on the SFU. In our case, this exploration is focused on accelerating an application domain, while keeping the power consumption of the design low. In the rest of this section, we explain the details of a specific implementation of the target architecture that we have just introduced.

### 2.3.1. The Case for Domain Acceleration

We have seen that the design of an ASIP involves augmenting a general-purpose processor with instructions customized for a particular application. However, if CIs are not frequently executed, the acceleration benefits will not compensate the overall cost and energy consumption of adding new hardware. Domain-specific acceleration increases the applicability of ASIPs, as they can target accelerating several applications that run on the same hardware close in time. Therefore, we present in this section an accelerator model that is reusable across a domain to increase its utilization, thereby improving both performance and energy efficiency.

We focus on the embedded market, where both performance and energy consumption are important factors. Thus, the baseline processor is in-order and low-power. The accelerator, or Domain-Specific Functional Unit (DSFU), is tightly coupled within the general-purpose processor pipeline. This would be technically feasible with the last generation of FPGAs, connecting a processor core to a reconfigurable array seamlessly [42]. We extend the basic ISA with CIs, such as in the traditional ASIP design. These instructions acce-

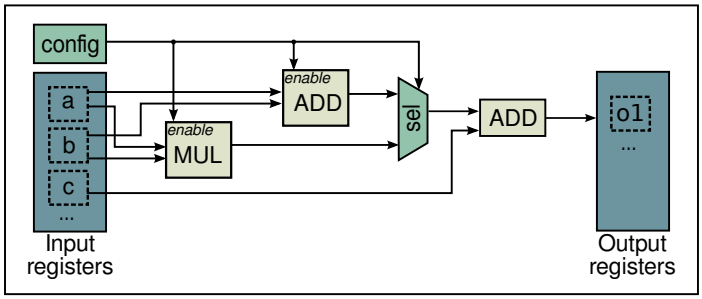


*Figure 2.2: Implementation of a merged CI that executes on a DSFU.*

lerate the programs by executing a bundle of predicated arithmetic operations in the DSFU.

The rest of this section presents the specification of the DSFU design and its integration in an Intel Atom processor's pipeline.

#### 2.3.1.1. DSFU Design

The DSFU processes CIs that execute intermittently at different points of varied programs. We consider a loop body, made up of one or several basic blocks, to be the basic portion of code that defines our CIs. They use few inputs, not necessarily consecutive in memory, to produce few outputs. As they access data through the processor's register files, input and output data is always within some established limits. Typically, CIs are calculation intensive, branch speculative. They exploit sub-word parallelism as SIMD instructions do, also executing operations of different kinds in parallel (instruction-level parallelism).

The direct benefit of a tightly-coupled, loop-body based DSFU is the performance speedup expected from more parallelism. Additionally, we can obtain a reduction of resource contention in different pipeline stages, due to collapsing several instructions into just one, including branch instructions that may fall into expensive branch mispredictions, or a reduction of instruction cache misses.

In the context of domain-specific acceleration, reusing hardware is critical for an efficient design. Of course, we can completely reuse CIs used in different programs if they are computing exactly the same functionality. However, in the case of unequal instructions, instead of maintaining separate hardware for each one, they can share the common parts of the circuit, taking up less hardware area. Thus, we introduce partial merges of CIs as in Figure 2.2. Consider the polynomials $F1 = a + b + c$ and $F2 = a * b + c$. They can be collapsed into the represented circuit, where a *config* signal activates

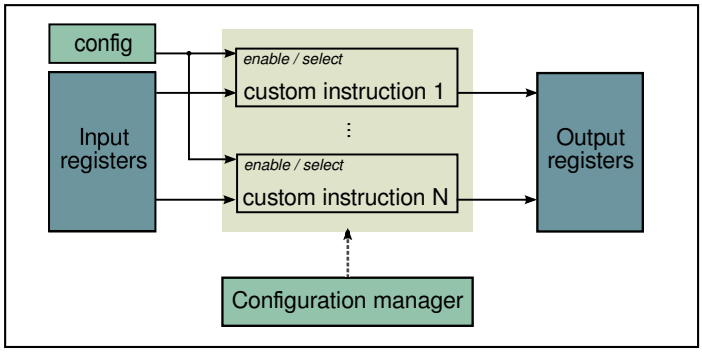


*Figure 2.3: DSFU with a configuration manager to assist in the reprogramming of the implemented CIs.*

the non-common part of the circuit and selects the operands of the common part. The *config* signal is encoded in the instruction coding.

Figure 2.3 shows the DSFU architecture template with several CIs that share two arrays of input and output registers, private to the DSFU, that are disjoint in order to overlap load and store operations. The CIs' connections to those registers are also controlled by the *config* signal. This architecture template can be adapted to different configurations with the configuration manager shown in the figure. Although we can still have a part of the design fixed at design time, such as the size of the input and output registers, we can reconfigure the CIs' implementation area at boot-up and/or run time.

Thus, CIs executing on the DSFU and their connections to the input and output registers are controlled by the *config* signal, encoded in the instruction coding. The configuration manager is connected to memory, where it can read a new configuration with a different implementation of CIs and modify the whole reconfigurable area (shadowed in the figure).

#### 2.3.1.2. Base Processor Integration

The DSFU is integrated in an in-order processor pipeline and augments the processor's functionality, neither replacing nor duplicating any existing functional units. A DSFU reads data using the processor's register files, and writes data back to them, with the premise of writing it always back to memory. If the number of inputs exceeds the amount of the register file's outputs, the execution of the CI requires a pre/post execute stage for extra data transfers. We count on the same data bandwidth as for other processor instructions using the processor's memory hierarchy.

We take the Intel Atom processor [43] as a concrete implementation

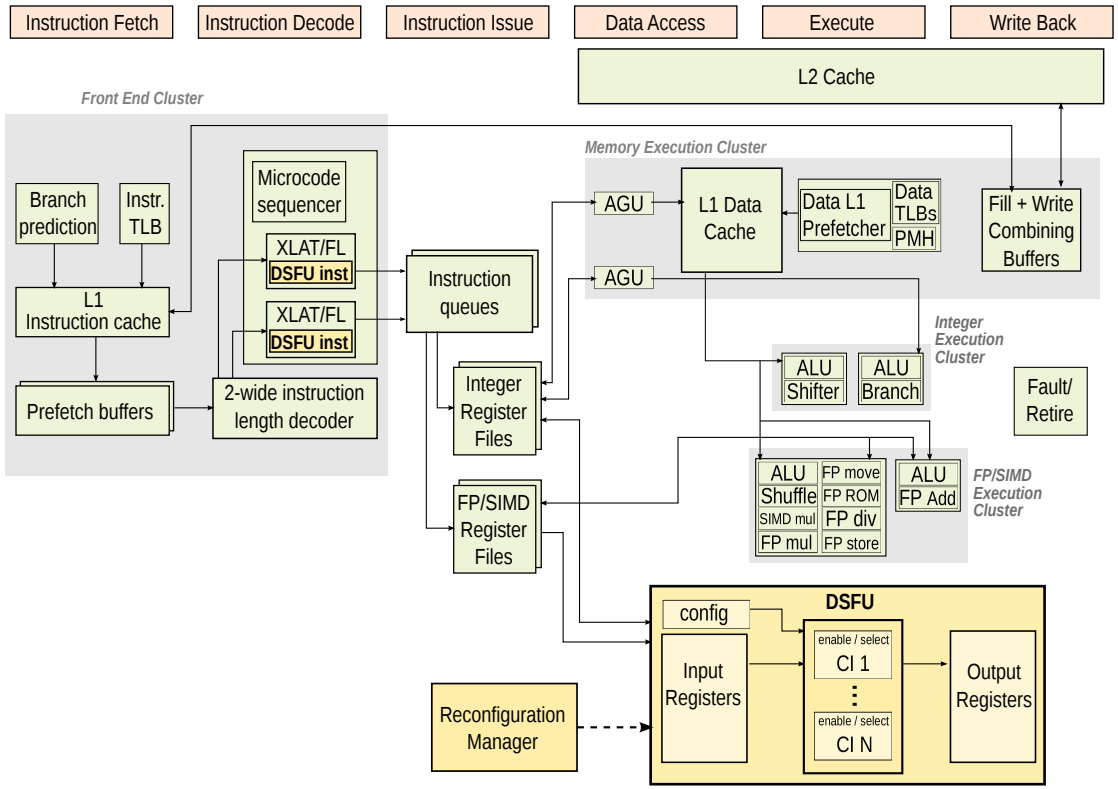


*Figure 2.4: Block diagram of a modified Intel Atom processor pipeline that includes a DSFU.*

example for the DSFU integration. The Atom processor is in-order and low-power, with one or two cores, each one running up to two threads. For simplicity, we do not consider its simultaneous multithreading (SMT) features at the moment. Although Atom processors implement the x86 instruction set, some versions support the x86-64 instruction set as well, which we take as the baseline ISA here.

### Microarchitecture details

Figure 2.4 shows a diagram of an Atom processor extended with an embedded DSFU. The pipeline stages are also depicted at the top of the Figure. Concretely, the pipeline has sixteen stages: three for instruction fetch, three for decoding, three for issue, three for data access, one for execution and three for exception handling and write-back.

We modify the instruction decode stage of the baseline processor to support CIs. The lookup table for the translation of instructions (XLAT) is extended with the additional instructions needed for the DSFU.

The DSFU's 16 128-bit input registers are directly connected for reading from both the integer (64-bit) and SIMD/XMM (128-bit) register files. A load from memory to an integer register (cache read) takes 1 or 3 cycles, while a load to a XMM register takes 4 cycles. Depending on the situation, we will read from one or another register file to minimize latency. The 32 64-bit output registers are only connected to the integer register file, since a store always takes 1 cycle, while a XMM store is set in 5 cycles.

DSFU's input and output registers operate at the same clock frequency as the main processor. However, the reconfigurable part of the DSFU where the CIs execute may operate at a different frequency than the main processor, depending on the reconfigurable hardware technology. Thus, there is a DSFU clock that is isochronous to the core clock; in other words, DSFU's period is a multiple of the core's period (for instance, there is a 1:4 ratio for an Atom core operating at 1.6 GHz and an FPGA set at 400 MHz). To keep the clocks synchronized, at the beginning of a DSFU execution, the decoding stage sends a reset signal to the DSFU clock before executing the requested CI.

The configuration manager, responsible for reprogramming the DSFU CIs, is directly connected to main memory to load the configuration data.

### ISA extensions

We extend the x86-64 based ISA with the instructions in Table 2.1, whose encodings are compatible with those of the Atom processor. The

| Instruction | Operands | Latency | Description |
|---|---|---|---|
| RF2DSFU_i | *dsfu_in, r* | 1 | Moves 64-bit of data from an integer register to a DSFU input register. |
| RF2DSFU_x | *dsfu_in, x* | 1 | Moves 128-bit of data from an XMM register to a DSFU input register. |
| DSFU2RF | *r, dsfu_out* | 1 | Moves 64-bit of data from a DSFU output register to an integer register. |
| DSFU_exec | *config, i/r/x* | $4 \times C$ | Resets DSFU internal clock and starts execution of the CI determined by config. The second operand may be used to transfer data to a determined input register. |
| DSFU_config | *m, i* | $F(i)$ | Reads $i$ number of bytes from $m$ memory location and reconfigures the CI implementation space of the DSFU with that configuration. |

*Table 2.1: Extensions to the base ISA to operate the DSFU. Conventions: i = immediate data, r = integer register, x = 128 bit xmm register, dsfu_in = DSFU's input register, dsfu_out = DSFU's output register, C = number of DSFU internal cycles. Operands are determined in Intel syntax (destination, source).*

first two instructions, RF2DSFU_i and RF2DSFU_x, move data from the register files to the DSFU input registers. When we need a memory access of at least 128-bit data whose address was recently calculated, it is better to use an XMM register (4 cycles versus $3 \times 2$ cycles to load). In all other cases we prefer to use the integer registers. The third instruction sends the data from the output registers to the integer register file, ready for write back.

A single instruction DSFU_exec is added to execute CIs. The *config* signal that chooses the CI configuration is passed as the first operand, allowing up to 256 different configurations. The second operand is data needed in the

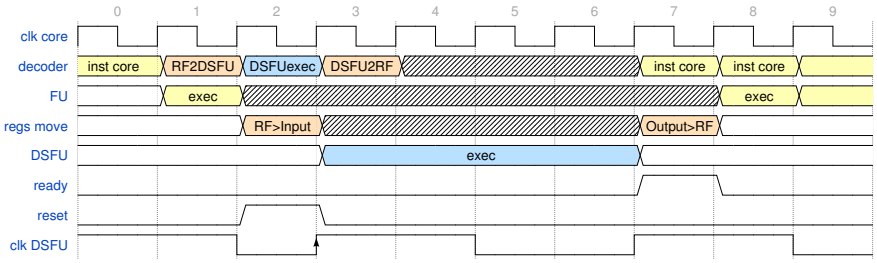


*Figure 2.5: Chronogram of a sequence of instructions executing on a pipeline with an integrated DSFU.*

DSFU execution, saving in the best case one cycle of a transfer instruction. When we try to execute a non-configured CI, an exception is triggered and the equivalent code in the non-extended ISA will be executed. The latency is determined for a DSFU that runs 4× slower than the Atom core. Therefore, the latency of the DSFU_exec instruction is the number of internal cycles $C$ that the CI takes in the DSFU, scaled to the core clock.

Finally, the DSFU_config instruction loads from the specified memory address a whole new configuration of the specified size. The latency of this instruction depends on the number of bytes to read and the memory bandwidth.

### Chronogram

Figure 2.5 shows an example of the execution of a CI, displayed as a chronogram with different pipeline stages and signals involved.

We have two clocks: the core clock in the first row, and the DSFU clock in the last one. Cycles in the upper part are counted as part of the core clock. After the core clock signal, we show the instructions decoded (*decode*), execution in normal functional units (*FU*), data moves between register files and DSFU registers (*regs move*), execution on the DSFU (*DSFU*), and the ready and reset signals. For simplicity, we assume that the instruction decode stage takes 1 cycle, though in Atom it would take 3 cycles. Also, the gap between decoding and execution would be wider in a real setting.

In cycle 1, the decoder processes a data transfer instruction that results in a register move in cycle 2. In cycle 2, the instruction that starts the DSFU execution is decoded, setting the reset signal until the end of the cycle, which triggers a reset of the DSFU clock. In cycle 3, the execution of the CI on the DSFU starts, while the decoder processes the instruction that reads the DSFU output, and then stalls. At the end of the DSFU execution the ready signal is set, triggering the moves from the output registers to the

register file and resuming the stalled pipeline.

## 2.4. Intermediate Code Representations

In this section, we examine the most relevant intermediate representations (IR) for this dissertation. These code representations are created for transforming an application's high-level code to an abstract description, in which significant features are highlighted. First, we discuss in Section 2.4.1 those models that represent the code's structure and flow, and we then review the canonical representation alternatives in Section 2.4.2.

### 2.4.1. Structural Representations

A data flow graph (DFG) is a well-known and widely-used representation that exposes the data dependences of operations within a basic block. It is built as a directed acyclic graph (DAG) $G(V, E)$, with $V$ the nodes that represent the operations, and $E$ the edges that stand for the data dependencies between nodes. Most of the CI design algorithms in Section 2.2 start from this representation to solve the candidate identification problem. The main advantage of this representation is that we can expose structural constraints naturally as the DAG imposes a strict topological order to the operations. In CI design, not all the operations of a basic block are included to be accelerated; the most common case is to avoid memory and branch operations. The resulting subgraph $G'$ after suppressing those *invalid* nodes may contain a structural hazard that make instruction scheduling infeasible. Therefore, CI design algorithms have to ensure that $G'$ is convex; that is, there does not exist any path in $G$ from a node $u \in G'$ to another node $v \in G'$ which involves a node $w \notin G'$.

Although DFGs have proved their validity for application-specific CI design, we question the use of DAG-formed basic blocks for domain-specific CI generation in Chapter 3.

#### 2.4.1.1. IR and SelectionDAG in LLVM

LLVM [44] is our compiler of choice at several phases of the frameworks we develop. Thus, we describe briefly here two representations within LLVM that are relevant for our tools, LLVM-IR and the SelectionDAG.

LLVM-IR, produced after the first compiler's frontend passes is the common code representation used for analysis and optimizations in LLVM. It is based on Static Single Assignment (SSA) and it is microarchitecture independent, though still representing type safety and low-level operations. The

IR is distinctly used in three forms: as in-memory C++ classes, as on-disk bitcode, and as assembly-like human-readable language. The assembly is a strongly typed RISC instruction set which abstracts away details of the target, for instance, using unlimited virtual registers. Basic block limits are clearly marked, as well as control flow through SSA's $\phi$ (*Phi*) functions. If auto-vectorization is activated in the frontend, the IR is also able to represent vector operations. This last form is the most common used one, and we work with it at several points of our frameworks: for profiling in FuSInG (Chapter 3), and for both profiling and HLS in MInGLE (Chapters 4 and 5).

The SelectionDAG is a code abstraction found in several backend phases of LLVM. It passes through different transformations, starting from an IR-close form to a more final microarchitecture specific representation. This representation facilitates the compiler's instruction selection phase that uses pattern matching techniques, as well as low-level target-independent optimizations. The SelectionDAG is a directed acyclic graph whose nodes are operation codes – the operation performed and the operands involved. It has two different kinds of edges: those to represent data flow dependencies, and those to represent control flow ones. We isolate and work on the DFG of the SelectionDAG of individual basic blocks just before the instruction selection phase in LLVM's backend. We use it for the *DFG Explorer* step that extracts the CI candidates in the FuSInG framework of Chapter 3.

### 2.4.2. Canonical Diagrams

A canonical representation presents a unique description for a given mathematical object, such as a graph, as a mathematical expression. With a canonical form we can test, for instance, if two graphs are equal. This is different to the graph isomorphism commonly applied to DFGs where we check the structural equivalence of two graphs. In the case of canonical representations, we aim to test the semantic equality of the mathematical expressions that the graphs represent, or in other words, if they are functionally equivalent.

Although there is vast work in graph-based canonical representations, here we focus on the relevant ones for this dissertation: Binary Decision Diagrams (BDDs), that represent binary functions, in Section 2.4.2.1, and Taylor Expansion Diagrams (TEDs), that can express work-level functions, in Section 2.4.2.2.

#### 2.4.2.1. Binary Decision Diagrams

A binary decision diagram (BDD) [45] represents a binary function as a rooted graph, based on a recursive Shannon decomposition, which com-

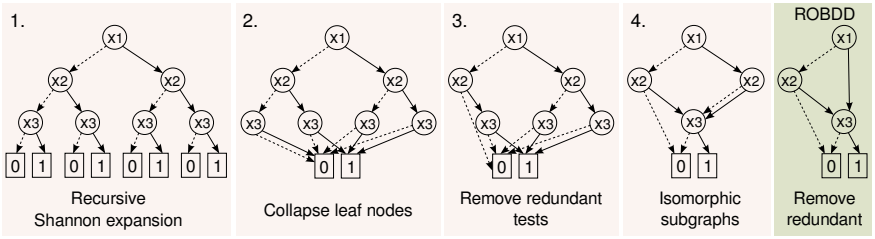


*Figure 2.6: Example of a reduced and ordered BDD construction for Boolean functions $f_1 = (x_1 \wedge x_3) \vee (x_2 \wedge x_3)$ and $f_2 = (x_1 \vee x_2) \wedge x_3$, with variable order $[x_1, x_2, x_3]$.*

bined with a set of reduction rules, makes the BDD minimal and canonical for a given ordering of variables. BDDs have been applied in formal verification problems, such as correctness check of circuit implementations.

BDDs represent Boolean functions that determine Boolean values from logical calculations on Boolean inputs. The representation is a rooted DAG with several decision and terminal nodes, all connected by decision edges. The initial node stands for the top Boolean formula. Non-terminal, decision, or internal nodes are labeled as a Boolean variable $w$ and have one out-edge 0 (*then*) and one out-edge 1 (*else*), Each non-terminal node represents the Boolean function corresponding to its 1 edge if $w$ evaluates to 1, or to its 0 edge if $w$ evaluates to 0, building up the top formula down to the terminals. Terminal nodes are labeled as 0 or 1, representing the Boolean functions 0 and 1. Figure 2.6 shows and example of BDD construction, explained in detail later. The rightmost graph is the final BDD of the Boolean function $f_1 = (x_1 \wedge x_3) \vee (x_2 \wedge x_3)$. Each node represents one of the Boolean variables $\langle x_1, x_2, x_3 \rangle$, and in this graphical representation, edges labeled with 0 are dashed and with 1 are solid.

The if-then-else normal form (INF) represents a Boolean function using the if-then-else (ITE) operator. For inputs $\{x, y, z\}$, ITE computes *if x then y else z*, which is equivalent to:

$$ITE(x, y, z) = (x \wedge y) \vee (\bar{x} \wedge z), \tag{2.1}$$

with variable $x$ evaluating to 1 or to 0.

Multivariate boolean functions can be expressed by recursive decomposition as a Shannon expansion using the ITE operator:

$$f(x_0, \ldots, x_n) = ITE(x_n, f_1(x_0, \ldots, x_{n-1}), f_0(x_0, \ldots, x_{n-1}). \tag{2.2}$$

Thus, the BDD is built performing the ITE logical function at each node, operating in time proportional to the size of the resulting function graph.

A BDD composed of INF expressions, with all equal ITE nodes shared, and with variables appearing in the same order and at most once in any path from root to leaf, is called reduced ordered binary decision diagram (ROBDD). Such BDDs are canonical since each derivation of a particular Boolean function leads to the same representation.

The variables in the ROBDD must have a specific order definition used to build the diagram recursively. Therefore, in ordered BDDs different variables appear in the same order from root to leaf in each expression path. For an ordered list of variables $L = [v_1, \ldots, v_n]$ without duplicates, a BDD $B$ has an ordering $[v_1, \ldots, v_n]$ if all the variable labels of $B$ occur in $[v_1, \ldots, v_n]$, and if $v_j$ follows $v_i$ on a path in $B$, then $j > i$. The orderings of two BDDs $B$ and $B'$ are compatible if there are no variables $\langle v, w \rangle$ such that $v$ is before $w$ in the ordering for $B$, and $w$ is before $v$ in the ordering for $B'$. The example of Figure 2.6 follows the variable ordering $[x_1, x_2, x_3]$ to apply the required recursive Shannon expansion in step 1.

To reach a full reduced BDD, reduction operations are applied to simplify the diagram to its maximum. First, duplicated terminals are removed; if there is more than one 0 terminal nodes, all edges pointing to them are redirected to just one of the 0 node. The same applies to 1 terminal nodes. Then, redundant tests are also removed; if the two outgoing edges of a node $v$ point to the same node $w$, $v$ is removed and its incoming edges are redirected $w$. And finally, duplicated non-terminals are removed as well; if two different nodes $v$ and $w$ are the roots of identical sub-BDDs, eliminate $v$ and redirect its incoming edges to $w$. In the example of Figure 2.6, duplicated terminals are removed and collapsed in step 2. Then, a redundant test with variable $x_3$ is also removed in step 3, while step 4 detects and collapses all the identical $x_3$ sub-BDDs. Finally, the reduced and ordered BDD is obtained removing in the last step redundant tests once again.

The reduced and ordered BDD is a canonical form of a logic function. This means that two functions with compatible variable orderings are equivalent if the ROBDD for each function are isomorphic. For instance, the ROBDD of Figure 2.6 represents not only Boolean function $f_1 = (x_1 \wedge x_3) \vee (x_2 \wedge x_3)$, but also the function $f_2 = (x_1 \vee x_2) \wedge x_3$. This property makes it useful in functional equivalence checking of Boolean functions, and we make use of this feature in the MInGLE framework (Chapters 4 and 5).

#### 2.4.2.2. Taylor Expansion Diagrams

A Taylor Expansion Diagram (TED) [46] is a canonical, graph-based representation like BDDs, but whose decomposition is non-binary. Such a representation raises the level of abstraction to allow word-level algebraic

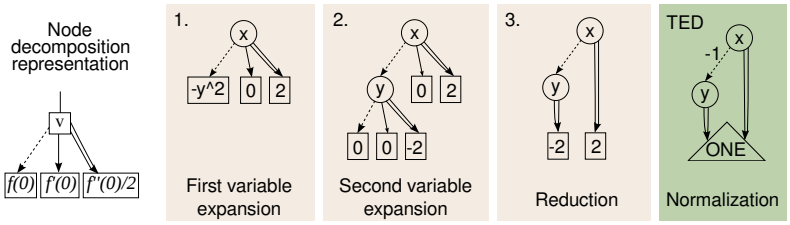


*Figure 2.7: Example of a TED construction for the polynomials $f_1 = (x+y)(x-y)$ and $f_2 = x^2 - y^2$, with variable order $[x, y]$.*

symbols with lower memory requirements than binary-based representations.

TEDs are based on the Taylor series expansion which, for a multivariate algebraic expression $f(x, y, \dots)$, is represented as follows:

$$f(x,y,\dots) = f(0,y,z,\dots) + xf'(0,y,z,\dots) + \frac{1}{2}x^2 f''(0,y,z,\dots) + \dots, \quad (2.3)$$

where the origin is set in $x = 0$ and with $f'(x = 0)$ and $f''(x = 0)$ as the successive derivatives of $f(x = 0)$. The individual terms of the expression are then decomposed with respect to the remaining variables on which they depend ($y, \dots$, etc.),

This decomposition, applied recursively at each algebraic function derived, is stored in a directed acyclic graph, the Taylor Expansion Diagram. A TED is composed of a root $\rho$, a set of nodes $V$, a set of edges $E$, and terminals $T$. The root represents the multivariate polynomial $\phi$ that the TED expresses. Each node $v \in V$ has an index that identifies an input variable and it is related to a specific decomposing order. For a node $v$, the function it represents is determined by the Taylor series expansion of all variables with indexes lower than $index(v)$. An edge $e$ is directed E is directed from $v$ to the derivative of the function with respect to the variable $index(v)$. Graphically, there are three different types of edges: dashed for the constant Taylor expansion, plain lined for the expansion on the first derivative, and double-lined for the expansion on the second derivative. The function computed at the terminals is an integer constant. On the leftmost diagram of Figure 2.7 we can find a key of that representation. In step 1 and 2 of the same figure, we show the Taylor expansions of polynomial $f_1 = (x+y)(x-y)$ with respect to variables $x$ and $y$.

The order in which the variables are expanded affects the size and shape of the final canonical representation. Following certain rules to find an initial variable ordering [47] can help to obtain TEDs optimized in size:

1. Variables that never appear in the same monomial on a single-output TED can be treated as outputs of a temporary multiple output TED.

2. Variables that appear in most terms of the monomial with the same exponent should be placed at the top of the TED.

3. Variables that appear in most terms of the monomial and have several exponents should be placed right after any variable identified in rule 1.

4. In the case of TEDs with multiple outputs: we place in rule 2 the node at the bottom instead of the top, and we put in rule 3 the node before instead of after.

A TED is reduced if it contains no redundant nodes and has no distinct nodes $v$ and $v'$, such that the subgraphs rooted at $v$ and $v'$ are isomorphic. A node in a TED is redundant if all of its non-0 edges are connected to terminal 0. We can reduce redundant nodes by removing them and merging isomorphic subgraphs. Figure 2.7 shows, in step 3, the reduction of a TED with redundant nodes and edges removed.

The normalization of a TED consist of propagating the weights at the terminal edges, or the common factor of all $k$ edges from node $v$ to the terminal node, and storing them as edge weights in the upper edges, enabling the extraction of common subTEDs. By applying this to all terminal nodes, only the terminal node 1, also represented as *ONE*, remains in the graph. Last step of Figure 2.7 shows a reduced and normalized TED.

For any multivariate polynomial *phi* with integer coefficients, there is a unique ordered, reduced and normalized TED denoting *phi*, that is, an ordered, reduced and normalized TED is minimal and canonical.

TEDs can also represent operators for Boolean logic:

- $not : x' = (1 - x)$
- $and : x \wedge y = x \cdot y$
- $or : x \vee y = x + y - x \cdot y$
- $xor : x \oplus y = x + y - 2 \cdot x \cdot y$

with $x$ and $y$ Boolean variables represented by binary variables. The resulting functions are $0, 1$ integer functions.

TEDs are limited to represent only those functions that have a finite Taylor expansion, therefore functions with an infinite Taylor series such as $a^x$ ($a$ is a constant) are excluded from the representation. Also, TEDs cannot

represent relational operators (such as comparators, $A \geq B$) in symbolic form, since relations are characterized by discontinuities over their domain and are not differentiable. For the same reason, modular arithmetic is also restricted.

TEDs have been commonly used for circuit verification. We use the representation for another purpose in this dissertation: to find common parts of the code that cannot be found with pattern matching techniques using DAGs. For instance, the example TED from Figure 2.7 represents two different polynomials that perform the same functionality: $f_1 = (x + y)(x - y)$ and $f_2 = x^2 - y^2$. These ideas are further developed in Chapters 3 to 5.

# 3

# Functionally Similar Domain-Specific Instructions

## 3.1. Introduction

In the introduction of this dissertation, hardware specialization was presented as a promising paradigm to improve performance and energy-efficiency in the *dark silicon* times. We discussed how an application-specific processor, while costly to manufacture, is limited to deliver high performance for a single application. We presented the alternative of customized processors that target an entire application domain may deliver better overall system performance when different applications run on the device, and may be more economically viable by targeting a larger market.

In this chapter, we focus on designing CIs that extend the ISA of a base architecture and accelerate a sequence of operations in different applications. We explore the design space of CIs that are implemented in an SFU in hardware, from those designed for a particular application versus those targeting many applications within a domain. With this in mind, we introduce a new technique to extract common sequences of computations from several applications within a domain, which become CIs. We use the canonical representation TED, traditionally used in the areas of compiler optimization and design verification, to identify common computations, We compare the effectiveness of DAG, TED, and a Hybrid technique at finding common code

sequences to target for acceleration in hardware. Our study shows that a canonical representation is key to identifying sequences that are mapped to the same CI across applications. We also evaluate four new scoring heuristics that prune the huge search space of the potential CIs without a detailed evaluation, selecting those that maximize the speedup of our application domain.

We build the exploration framework FuSInG (Functionally Similar Instructions Generator) to estimate the speedup of new CIs, across the spectrum of application-specific and domain-specific acceleration. We use 9 media benchmarks, and extend the LLVM compiler framework to identify code sequences amenable for acceleration. We extract sets of reusable CIs, both within and across benchmarks, which we subsequently analyze and rank using our scoring heuristics. We then use the Xilinx design software to synthesize a hardware implementation of a potential CI. Given an instruction's hardware datapath, we use estimation models to approximate its core area and number of cycles, and thus speedup. We show that while DAG, TED and Hybrid perform similarly when finding CIs for a particular application, using the TED and Hybrid techniques to identify CIs across a domain leads to much higher speedups than when using the DAG technique alone. Our analysis reveals that when the SFU occupies a small, realistic core area, it obtains the highest speedups when including both CIs designed across all applications in a domain and some specific to one application. We study a few machine design points in detail: given a particular area, we present the characteristics of the SFU that obtains the highest speedup. Finally, we study how well CIs identified for a set of benchmarks perform for other, previously unseen workloads.

This chapter is organized as follows. Section 3.2 sets the context and main challenges. Section 3.3 gives a high-level description of the developed framework, which details are elaborated in Sections 3.4 to 3.7. The experimental setup is explained in Section 3.8 and the results presented in Section 3.9. We close this chapter with the summary in Section 3.10.

## 3.2. Context

We assume that the CIs execute on an SFU that is tightly integrated within the datapath of a general-purpose processor, as presented in Chapter 2, Section 2.3. Figure 2.1 shows a high-level diagram of such an architecture. Specialized execution pipelines that execute the CIs can be configured at system boot time. When analyzing code sequences to identify CIs, we disallow control or memory operations, since the SFU reads and writes data from and to the processor's register file. Therefore, we do not focus on

creating a new specialized processor, but on accelerating a general-purpose processor using a small additional amount of chip area.

Previous research has used automated tools to identify repeated patterns of instructions and propose them as extensions to the ISA. Initial developments established the grounds for the field using exhaustive identification of patterns [20] and approximate techniques [16]. Other works [19, 48] have used pattern matching-based approaches on the data flow of programs, represented as directed acyclic graphs (DAG), to identify CIs across a domain. However, pattern matching cannot always find similarities between sequences of code in order to map different functionality to the same CI, inherently limiting specialized hardware opportunities.

Consequently, in this chapter we explore the trade-off between application-specific versus domain-specific hardware specialization. Given a defined set of applications, our main objective is to design the hardware to maximize the platform's efficiency. We focus on maximizing speedup, or boosting system performance and application execution time, given a particular core area dedicated to the SFU. Exploring the application-specific versus domain-specific specialization trade-off involves a number of challenges. For one, we need a framework to identify code sequences within and across applications that are amenable to hardware acceleration. Finding common code sequences across applications is particularly challenging because of the huge search space, i.e., one needs to keep track of all code sequences of all applications to be able to find commonalities, and one needs to find the best way to represent these code sequences to maximize the likelihood of finding commonalities both within and across applications. Further, to be able to quickly explore the CI design space and keep exploration time reasonable, we need heuristics to rank the effectiveness of potential specialized hardware without relying on detailed evaluation of each possible CI. We have to use tools to estimate the speedup an application would achieve when using a particular set of CIs, and optimize not only for speedup across the domain of applications, but also for minimizing the SFU's area. In order to perform this study, we have built an accelerator exploration framework, which we describe next and which includes several novel contributions over prior work to identify and rank potential specialized functional units that accelerate computation.

## 3.3. FuSInG Automatic Framework

Figure 3.1 shows an outline of our CI selection and evaluation framework FuSInG (Functionally Similar Instructions Generator), which we detail in the following sections. We first analyze application code to identify po-

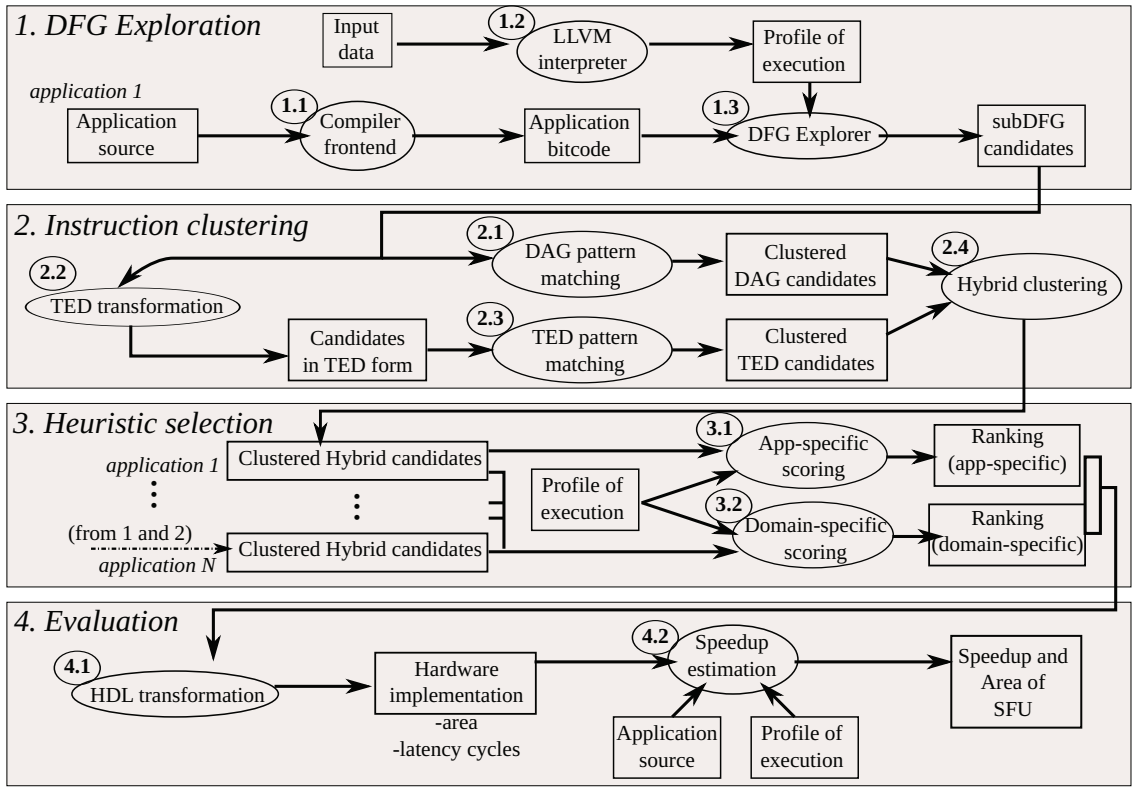


*Figure 3.1: Schematic overview of our CI selection and evaluation framework* FuSInG.

tential code sequences for CI design (Step 1). We then take steps to find commonalities among these identified code sequences, both within and across applications (Step 2), and then evaluate which CIs are most effective using newly proposed scoring heuristics (Step 3). Using these heuristics, we plug our chosen CIs into a low-level model that estimates both the speedup and the area of each (Step 4), so we can evaluate the potential of new computer designs with hardware acceleration.

## 3.4. Identification of CI Candidates with DFG Exploration

Step 1 of Figure 3.1 shows how we identify code sequences amenable for acceleration in hardware. We use the compiler (label 1.1 in the figure) to transform the source code of the application into its IR to expose the DFG and CFG of the program. We use an IR representation close to the assembly language to find sequences of code that could be turned into specific CIs in hardware. Because identifying sequences of code to accelerate could blow up to a huge state space search, we apply certain constraints to lower the space exploration.

Static program analysis, implemented in the DFG Explorer (label 1.3), identifies a list of candidates that could be implemented as CIs. Each candidate must be a maximal convex subgraph [20] of a data flow graph for a given basic block, that is, the biggest disconnected subgraph of a basic block that preserves the convexity constraint [12]. These subDFGs exclude *invalid* instructions that cannot be executed in the SFU. In this chapter, we assume that the SFU executes neither memory nor branch instructions to keep the unit highly integrated in the processor's pipeline. Instead, they are executed in the core's ALU, thus we mark them as *invalid* in the exploration step. However, to support other kinds of acceleration hardware that target code beyond the basic block level, and include memory instructions, we could extend this step of the framework as well as step two, which clusters instructions using TEDs. Therefore, our exploration framework FuSInG was built to be general and broad enough to study a variety of acceleration designs.

The DFG exploration is done with a fast implementation of the algorithm presented by [21] using binary structures. The algorithm performs a binary search for each basic block in the application, first enumerating the *invalid* instructions of the graphs, which turn into the cutting nodes of the subtrees to be explored recursively in the search. The exploration result is a list of candidate code sequences, represented as subDFGs, that satisfy the

criteria above in non-exponential asymptotic time complexity (bounded by the number of *invalid* instructions, as they define the amount of recursive calls).

In order to cut down on the number of candidates, we define a few rules to limit subDFG candidates. Groups of instructions are selected to preserve the consistency of scheduling, which means that all the inputs of the set are ready at issue time. In our exploration, we allow unlimited inputs and outputs to the CI, because more complex CIs will potentially achieve a higher speedup. We also limit the exploration space by only considering executed parts of the code, using a previously-gathered execution profile of the application (label 1.2 in Figure 3.1). At the end of Step 1, we have a list of candidates that are then passed to the next step which clusters the potential code sequences to help select CIs.

## 3.5. Instruction Clustering to Discover Equivalences

In Step 2 of Figure 3.1, we analyze the code sequences found in Step 1 in order to group them to propose CIs that apply to several sequences of code. This clustering step can be performed on code sequences identified from the same application (targeting application-specific CIs), and/or sequences from different applications (targeting domain-specific CIs). Clustering serves several functions: to enhance reusability, to minimize implementation area in hardware, and to reduce the search space in the selection step.

In the following sections, we describe three methodologies for the clustering: *DAG*, *TED* and *Hybrid*.

### 3.5.1. Clustering with DAG Isomorphism

The first technique clusters the code sequences using directed acyclic graphs (DAGs). For each pair of subDFGs obtained in Step 1, we perform a one-to-one isomorphism detection (label 2.1 in Figure 3.1). Those graphs that are isomorphically exact are clustered under the same label, to be potentially transformed into a single CI candidate.

Previous works [19, 48] approached the problem by starting from small graphs, building them up to arrive at relatively large-sized accelerators — a bottom-up approach. In our work, we employ a top-down approach and start from maximal subgraphs extracted from a basic block, ideally covering as large code sequences as possible, and exploit as much instruction-level parallelism as possible.

Relatively larger CIs are more likely to yield better overall performance, but the identification of big patterns of functionally identical computation is

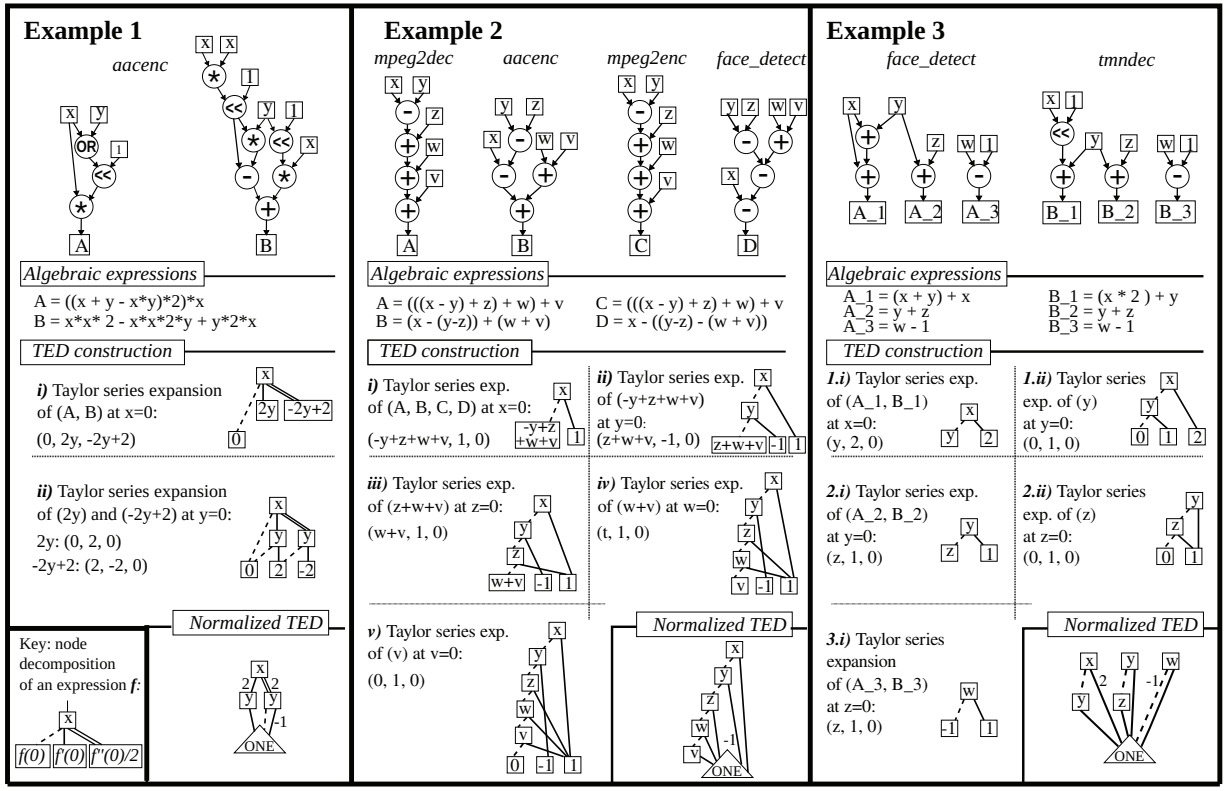


*Figure 3.2: Three examples of the usage of TEDs for instruction clustering. From top to bottom: DAGs, Algebraic expressions, TED construction process and final normalized TEDs.*

a complex problem. Consider the three examples of subDFGs in Figure 3.2, identified in different benchmarks and their equivalent algebraic expressions. Example 1 shows two portions of code of the aacenc application from different basic blocks in their DAG representations. They differ in the number and types of instructions they contain. Simple DAG pattern matching would not cluster these two DAGs, although their algebraic functions are equivalent. In Example 2, we extend the problem to a domain of applications. We show DAGs of basic blocks from different benchmarks (mpeg2dec, aacenc, mpeg2enc and face_detect) that perform the same computation, but with different operators. The DAGs of two of them (mpeg2dec, mpeg2enc) are isomorphically the same, therefore they could be clustered with DAG pattern matching. However, DAG pattern matching is not able to cluster all four of them. In Example 3 we show two DAGs of face_detect and tmndec with multiple outputs. In this case, although we can have a partial match with DAGs for outputs 2 and 3, the full match for identical computation cannot be found. Summarizing, in the three motivational examples, pattern matching using DAGs is missing opportunities to find commonalities among code sequences.

### 3.5.2. Clustering with TED isomorphism

Because of the limitations of using DAG pattern matching, we introduce a second clustering technique based on a canonical representation of portions of the application's code. We gather insights from works on TEDs (see Chapter 2, Section 2.4.2.2), adapting them here to find common parts of the code that cannot be found with a simple pattern matching technique using DAGs. We match code from applications using TEDs at compile time (at an intermediate code level), and thus the shape of a TED does not influence the final implementation of a CI at the circuit-level.

Although TEDs were described in Chapter 2 in great detail, we briefly introduce here the basics of the representation to understand how the TED technique works for cases such as the one depicted in the examples of Figure 3.2. Starting with a multivariate algebraic expression, we apply recursively the Taylor series expansion and we store this decomposition, into a directed acyclic graph, the Taylor Expansion Diagram (label 2.2 in Figure 3.1). Each node of the graph represents an input variable, and three different types of edges can be linked to a node: constant Taylor expansion, the expansion on the first derivative, and the expansion on the second derivative. Following a set of rules, we obtain a normalized and canonical representation of the TED from the starting algebraic expression.

We start with the computations expressed as subDFGs or DAGs from

Step 1 in Figure 3.2. Then, in order to build a TED, we execute the following steps:

1. Convert the subDFG into an algebraic expression. Note that boolean logic can be expressed as an algebraic expression as well: for example, the logical 'or' operation can be represented as $x \vee y = x + y - xy$.
2. Establish the order in which the variables are expanded, as it affects the size and shape final canonical representation.
3. Recursively calculate the values of the Taylor expansion for the constant, first and second derivative for every term in the algebraic expression, following the order from point 2.
4. Apply reduction and normalization rules to ensure that the TED is canonical.

We explain the TED construction with three examples in Figure 3.2. In Example 1, the first step converts the DAGs into the algebraic expressions A and B written under the graphs. Note the expansion of the 'or' operation into its counterpart algebraic expression. In the second step, we decide the ordering of the variables, which is important to arrive at a canonical representation. In this case, the order is $x, y$. In the third step we construct the TED, which will be unique for both A and B, as their Taylor series expansions yield the same values. Step *(i)* in the TED construction builds a partial TED performing the Taylor series expansion first on variable $x$. Then, step *(ii)* expands on variable $y$. The resulting TED, after applying normalization and reduction, leads to the reduced version in the bottom of the example. For Example 2, the four algebraic expressions are expanded in the same way, as shown in steps *(i)* to *(v)*. In Example 3, with multiple output DAGs, we will have an algebraic expression for each one of the outputs. Each expression is transformed into the corresponding TED, with as many steps as input variables. At the end, the generated TEDs, separately, are reduced and normalized, but also merged into a single normalized TED.

Finally, as TEDs are also directed acyclic graphs, we perform a one-to-one isomorphism detection with the normalized TED — like the ones at the bottom of Figure 3.2 – as we do with the DAG representation (label 2.3 in Figure 3.1).

### 3.5.3. Hybrid TED-DAG clustering

The final clustering technique is a hybrid TED-DAG technique. Not all computations in their directed acyclic graphs can be converted to a polynomial expression, and only polynomials with a finite Taylor expansion can be

modeled as TEDs. This excludes modular arithmetic, relational operations, and exponentiation of constants as a base, whereas a DAG can represent all types of computations as they are expressed structurally in the DFG. Due to these restrictions, we propose a hybrid technique that uses the TED representation when it can be created, and otherwise uses the DAG representation of subDFGs to cluster computation (label 2.4 in Figure 3.1). Using this hybrid approach, we should be able to cluster more code sequences to target the same hardware, identifying the most efficient CIs for our set of applications.

## 3.6. Heuristic Selection

After clustering code sequences, we have many CI candidates. In order to select the most promising ones for our applications, we introduce four novel scoring heuristics in Step 3 of Figure 3.1. Our scoring techniques use dynamic execution data from the applications in order to prioritize CIs, either focusing on application-specific or domain-specific CIs, that maximize speedup. They score based on the number of regular instructions covered by each CI, the frequency of execution of the basic blocks that contain the subDFG that maps to that CI, and (for domain-specific) the number of applications that can use each CI.

### 3.6.1. Application-Specific Scoring

We first focus on a scoring heuristic that prioritizes CIs targeted at just one application (label 3.1 in Figure 3.1). Our heuristic ranks CIs based on the potential speedup they can offer, using the following terms: $K$ is a CI for which $n$ code sequences are found in an application, i.e., $n$ code sequences can be accelerated using CI $K$. $ninst_i$ is the number of regular instructions and $freq_i$ is the frequency of execution of the code sequence amenable to the CI. The latter is gathered through profiling (label 1.2 in Figure 3.1).

Our application-specific scoring heuristic for CI $K$ is then defined as:

$$scoring_K = \sum_{i=1}^{n} ninst_i \times freq_i, \tag{3.1}$$

and essentially weights all code sequences with their instruction counts and execution frequencies to have a measure of the speedup of the application as a whole.

### 3.6.2. Domain-Specific Scoring

To identify CIs that are most efficient across a domain of applications, we must use different heuristics that take into account the reusability of the hardware (label 3.2 in Figure 3.1). We still take into account a CI's execution frequency, however with a slight change. Because we are considering different applications, we must normalize the execution frequencies to the application's total dynamic instruction count. For any given application, the normalization is done by scaling the frequency of execution to the percentage of the application's total number of instructions executed.

We first define the following variables:

- $K$ is a CI with $n$ code sequences found across all applications ($1 \leq n$).
- $ninst$ is the number of regular instructions of a given code sequence amenable to the given CI.
- $nfreq$ is the normalized frequency of execution of the given code sequence.
- $napp$ is the number of applications that can use the CI.
- Each of these $napp$ applications can use the CI at $m$ different points in the code ($1 \leq m \leq n$), and thus ($n = \sum_{i=1}^{napp} m_i$).

We now detail four new scoring heuristics that each prioritize CIs differently, and we compare them later in the experimental results section.

#### 3.6.2.1. Scoring 1: Normalized Application-Specific

$$scoring_K = \sum_{i=1}^{n} ninst_i \times nfreq_i \tag{3.2}$$

Equation 3.2 shows the first scoring, which is similar to the application-specific scoring, though it uses normalized frequency values. It maximizes the ranking of frequently used CIs targeting high numbers of instructions. A CI's sharing across applications is not taking into account with this scoring heuristic.

#### 3.6.2.2. Scoring 2: Scaled by Sharing

$$scoring_K = (\sum_{i=1}^{n} ninst_i \times nfreq_i) \times napp \tag{3.3}$$

Our second scoring technique, in Equation 3.3, does take into consideration a CI's ability to be reused or shared across applications. The *napp* factor prioritizes CIs that have a high sharing factor, when the scoring has to discriminate among CIs with similar numbers of normalized dynamic instructions. Application-specific CIs that are very frequently used are still highly ranked, since $nfreq_i \gg napp$.

#### 3.6.2.3. Scoring 3: Geometric Mean of Sharing

$$scoring_K = \sqrt[napp]{\prod_{i=1}^{napp}(\sum_{j=1}^{m_i} ninst_j \times nfreq_j)} \tag{3.4}$$

Equation 3.4 shows our third scoring heuristic that calculates the geometric mean of the $m_i$ application-specific scores, where $i$ an index that iterates over the applications involved. Since application-specific scores for a given CI can vary by several orders of magnitude, we propose this scoring to smooth out the spikes in the scores due to a single application (when $napp > 1$). CIs that benefit many applications but get a high score from only one application, are penalized. This heuristic thus introduces fairness for CIs targeting several applications. However, CIs used by one application are not penalized.

#### 3.6.2.4. Scoring 4: Random-Scaled Sharing

$$scoring_K = \sum_{i=0}^{napp-1}(\sum_{j=1}^{m_i} ninst_j \times nfreq_j) \times \frac{napp}{napp - i} \tag{3.5}$$

In the final scoring heuristic, in Equation 3.5, we introduce a randomness factor controlled by the number of applications that the CI targets. The application-specific scoring is weighted by $\frac{napp}{napp-i}$. The assignment of $i$ is random, but *napp* still influences the final result, thus the higher the sharing factor, the higher the score. Note that the value of $i$ assigned to a particular application is non-deterministic, so the applications are weighted differently for each code sequence. The reason for introducing some controlled randomness is to distribute scores in a more flexible way, since there are other factors that we do not consider in our current heuristics, such as the area that a CI occupies.

## 3.7. Estimating Performance and Area

Finally, in Step 4 from Figure 3.1, we evaluate the effectiveness of the CIs identified by the previous three steps. Informed by the ranking of CIs produced by the scoring heuristics in Step 3, we feed top CIs into a hardware description language conversion tool that creates a preliminary hardware implementation (label 4.1 in Figure 3.1). This implementation verifies that the identified sequences of code can be implemented as hardware structures, and double-checks the scoring techniques. The hardware implementation, using information from the application profile, is fed into a model that estimates the achievable speedup and area occupied by each CI (label 4.2 in Figure 3.1). Area estimates are obtained through hardware synthesis as we will explain in Section 3.8.

We estimate the speedup each CI can achieve for each identified sequence of code as follows. Consider a CI that would be invoked at $n$ different locations in the code of a particular application, that covers *ninst* normal instructions, and is executed *nfreq* times at a particular location. Further, assume that hardware synthesis estimates the CI to take *hw_cycles* to execute. Consider also a cost of *Cin* cycles to move input data from the register file to the SFU before the CI starts and a *Cout* cost to move outputs back to the register file at the end of the accelerated execution. Both costs depend on the number of input and output parameters of a particular CI and the available register ports in the baseline processor. We first estimate the execution time in cycles of all uses of the CI (on the SFU) as:

$$T_{w/\,ci} = \sum_{i=1}^{n} nfreq_i \times (hw\_cycles + Cin_i + Cout_i), \tag{3.6}$$

or the number of times the CI is invoked multiplied by its execution time in cycles. Then, we estimate the number of cycles that the same sequences of code would take on the non-customized processor (without using the CI):

$$T_{w/o\,ci} = \sum_{i=1}^{n} ninst_i \times nfreq_i \times CPI, \tag{3.7}$$

with *CPI* as the cycles per instruction of the application on the target processor.

We define $T$ as the total application execution time in cycles on the target processor (without using the CI). We then can find the difference between the number of cycles our candidate sequences take on the non-customized processor versus using CIs, and subtract this from $T$ to approximate the accelerated performance. Formally, the estimated total application

time when using CIs is:

$$T - (T_{w/o\ ci} - T_{w/\ ci}). \tag{3.8}$$

We then divide $T$ by that estimated time to calculate the SFU's achievable speedup. This is a conservative estimate since we do not take into account the potential instruction-level parallelism between regular and CI execution, which would result in higher speedups.

With this evaluation step, we are able to compare the potential performance improvements that a set of CIs, whether including just application-specific CIs, domain-specific, or both, can provide to an application or set of applications.

## 3.8. Experimental Setup

We describe the implementation details of our specialized functional unit design exploration framework FuSInG, including the software and hardware tools used, and our benchmarks.

We use the LLVM compiler infrastructure [44] as the front-end to our CI design exploration framework FuSInG. We modify the LLVM code generation module to find maximum valid subDFGs for DFG exploration (Step 1 in FuSInG). Using the NetworkX library [49], we perform graph isomorphism detection, and construct the TED representations using the symbolic algebra and calculus part of Sage [50]. We obtain an execution profile for each of our applications using the LLVM binary interpreter. The profile indicates the frequency of execution for each basic block, and is used in Steps 2 to 4 of FuSInG.

We assume that the target architecture has spare core area tightly-coupled to the processor core to implement the configurable SFU, as shown in Figure 2.1 (Chapter 2). We consider a single-core single-thread OpenSPARC T1 as the baseline architecture, which has been adapted previously for research on embedded applications [51]. The register file that both the ALU and the SFU access consists of 32 64-bit registers with three read, two write and one transport ports. The instruction encoding allows moving two input operands to the SFU with no additional cost. Any extra inputs are sent in groups of three, with a cost of one cycle per transfer, before the CI execution starts. When the instruction ends, outputs are packed together in groups of two and moved back to the register file, with a cost of one cycle per transfer.

To evaluate the selected CIs, we first translate their functionality to C code. For a given application, CIs that are functionally equivalent are translated to one common piece of code. Across applications, for a given set of

| Benchmark | Description | Input |
|---|---|---|
| aacenc | AAC audio compression encoder | 33.9 MB WAV |
| cjpeg | JPEG image format compressor | 1.2 MB PPM (Mediabench) |
| djpeg | JPEG image format decoder | 12.8 kB JPEG (Mediabench) |
| face | Face detection on bitmap files | 734.5 kB bitmap |
| tmndec | H263 video format decoder (TMN) | 114 kB H263 (Mediabench) |
| tmnenc | H263 video format encoder (TMN) | 5.5 MB YUV (Mediabench) |
| mpeg2dec | MPEG2 video format decoder | 34.9 kB (Mediabench) |
| mpeg2enc | MPEG2 video format encoder | 506.9 kB (Mediabench) |
| optflow | Optical flow for motion estimation | 884 kB images |

*Table 3.1: Description of the evaluated application benchmarks and their input files.*

sections of code identified as functionally equivalent, we provide an implementation of the CI execution path for each application involved. Later, we choose the best among them for the performance model. We use the Vivado HLS suite to perform C to HDL conversion on those C code segments. For feasibility reasons, our automated toolchain uses the default optimizations of Vivado HLS [9]. Any further improvements to the hardware implementation with specifically-set optimizations would result in better overall speedups. The Xilinx ISE tool performs the synthesis of the design, using the Virtex 5 FPGA as a target, which estimates the new hardware's area (per CI) as a number of look-up tables (LUTs) and slices. We report area estimates relative to the OpenSPARC T1 core area, which is also mapped onto a Xilinx Virtex 5 FPGA for apples-to-apples comparison. We also use the Xilinx ISE reports to estimate the number of cycles per CI, which we use to estimate performance speedup through acceleration as previously explained.

Table 3.1 shows the list of benchmarks that we use for our experiments, with their descriptions and input files. All the applications belong to the media domain. The optical flow kernel and the face detection benchmark are part of the OpenCV library [52]. The AAC (audio compression) encoder is based on a program provided by Renesas Technology and Hitachi Ltd. The rest of the applications and their input files belong to the Mediabench II benchmark suite [53].

| Benchmark | Num. code sequences | | | Num. CI | | | % dynamic instr. | | |
|---|---|---|---|---|---|---|---|---|---|
| | *DAG* | *TED* | *Hybrid* | *DAG* | *TED* | *Hybrid* | *DAG* | *TED* | *Hybrid* |
| aacenc | 81 | 73 | 72 | 29 | 32 | 27 | 10.5 | 6.1 | 4.9 |
| cjpeg | 126 | 138 | 140 | 53 | 41 | 41 | 3.5 | 10.8 | 10.9 |
| djpeg | 115 | 119 | 119 | 52 | 43 | 43 | 2.0 | 16.9 | 16.9 |
| face | 165 | 211 | 211 | 45 | 66 | 66 | 0.9 | 9.3 | 9.4 |
| tmnenc | 89 | 116 | 121 | 29 | 37 | 38 | 0.5 | 0.9 | 0.8 |
| tmndec | 51 | 68 | 70 | 31 | 43 | 45 | 2.8 | 6.6 | 6.6 |
| mpeg2dec | 75 | 83 | 86 | 44 | 40 | 43 | 24.1 | 16.6 | 21.2 |
| mpeg2enc | 106 | 164 | 172 | 51 | 68 | 72 | 2.1 | 9.0 | 9.7 |
| optflow | 1 | 7 | 7 | 1 | 6 | 6 | 0.0 | 27.2 | 27.2 |

*Table 3.2: Number of code sequences and CIs found in each application with DAG, TED and Hybrid methods, and the percentage of dynamic instructions covered by them. These results use the random-scaled sharing heuristic, and are for unlimited core area.*

## 3.9. Results

In this section we present the experimental results obtained using the FuSInG framework presented in Section 3.3. We first compare the speedup that we can achieve using the DAG, TED, and Hybrid clustering techniques described in Section 3.5, showing in Section 3.9.1 that TED and Hybrid techniques by far out-perform DAG for identifying CIs across a domain. We then show differences between our four new scoring heuristics (from Section 3.6) across benchmarks, demonstrating in Section 3.9.2 that on average the random-scaled sharing heuristic works best for our applications. In contrast to Sections 3.9.1 and 3.9.2, focusing only on domain-specific CIs, we then evaluate the differences in speedup that can be achieved using only domain-specific, only application-specific, or a mix of both kinds of CIs in Section 3.9.3. With the whole core area at our disposal, application-specific CIs achieve the highest speedup; however, at lower core areas, domain-specific CIs perform well, but always benefit from the addition of application-specific CIs. Using both kinds of CIs, we achieve the highest speedups. In Section 3.9.4, we perform a detailed analysis of the CIs included at particular percentages of core area for application-specific, domain-specific, and mixed configurations. We reveal insights about the number of small, medium, and large CIs, the average number of inputs and outputs, and the number of applications each configuration can target. Finally, in Section 3.9.5, we evaluate a more realistic setting using cross-validation, evaluating how a set of CIs identified as useful for a group of applications perform for another, previously unseen, application.

### 3.9.1. DAG vs TED vs Hybrid

We first evaluate the effectiveness of using a directed-acyclic graph to guide pattern matching between code sequences (DAG), versus using a canonical approach to cluster code sequences (TED). We compare their effectiveness considering all applications from the domain. Table 3.2 compares the three techniques for each benchmark in the number of code sequences they identified, number of CIs selected, and percent of total dynamic instructions that can be converted to CIs. These numbers were gathered using the random-scaled sharing heuristic to rank candidates, and devoting an unlimited core area to the SFU. We select a CI if it can accelerate two or more code sequences from different benchmarks. For all but one benchmark (aacenc), the TED and Hybrid techniques find a larger number of code sequences than DAG. For all but two benchmarks (cjpeg and djpeg), TED and Hybrid also select about the same or a larger number of CIs. Even with cjpeg

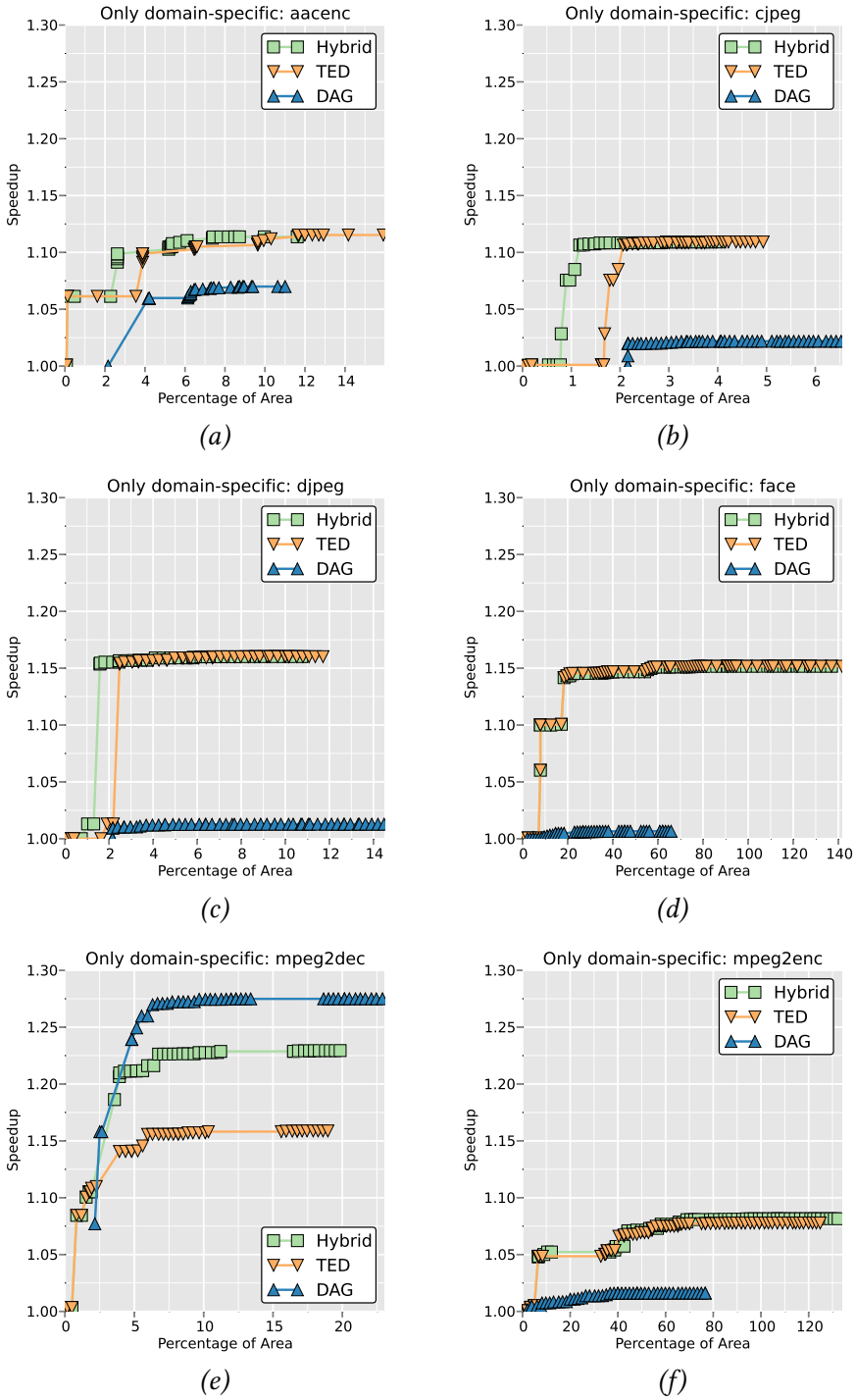


*Figure 3.3: Results of benchmark speedup versus CI area for DAG, TED and Hybrid methods, with domain-specific CIs using random-scaled sharing scoring (part 1/2).*

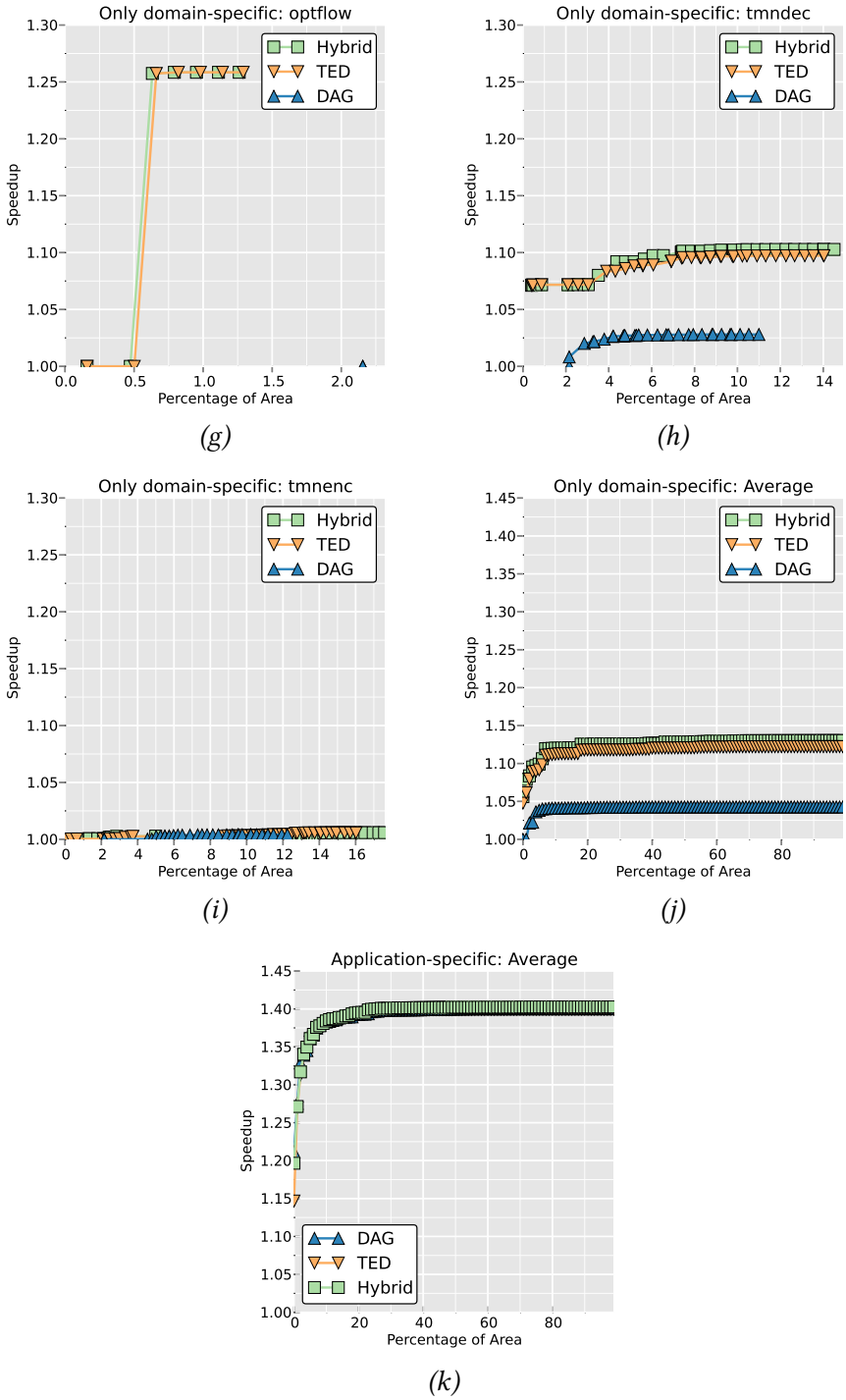


*Figure 3.4: Results of benchmark speedup versus CI area for DAG, TED and Hybrid methods, with domain-specific CIs using random-scaled sharing scoring (part 2/2). Graphs (j) and (k) show averages.*

and djpeg, TED and Hybrid cover significantly more dynamic instructions than DAG, which is also the case for all other benchmarks except aacenc and mpeg2dec. As the heuristic discards instructions that might cover more execution time, TED and Hybrid perform slightly worse for aacenc and mpeg2dec.

Figures 3.3 and 3.4 present graphs with a range of core areas dedicated to the SFU on the x-axis, and speedup on the y-axis. Individual benchmarks are shown from *(a)* to *(i)*. These results use the best performing scoring heuristic (random-scaled sharing), which we discuss in detail in the next section. Each point on the graph represents a group of domain-specific CIs that can be used by that benchmark and that fit inside that core area (x-axis), which together can achieve that speedup (y-axis) for a given benchmark. Note that each benchmark has a different x-axis scale because these are the area percentages used per benchmark, not for the whole SFU. In all following sections, we consider the entire SFU design when discussing area. The average of all applications (using total SFU area) is shown in *(j)*. The last graph *(k)* shows the average area and speedup numbers for the three clustering techniques when we only include application-specific CI.

On average, the Hybrid technique, which uses the TED representation when it is able and otherwise uses DAG, is the most effective technique at finding domain-specific CIs (Figure 3.4 *(j)*). The Hybrid technique achieves higher speedups at smaller areas (left hand side on the graphs in Figures 3.3 and 3.4), always increasing the speedup faster than the other two techniques. All but two benchmarks show the best speedups with TED and Hybrid techniques regardless of area, and for tmnenc *(i)*, DAG performs best only between 6% and 12% core area. When given unbounded core area, only one benchmark, mpeg2dec *(e)*, performs better with the DAG clustering technique than with Hybrid. This happens because the Hybrid technique first tries to identify CIs using TED, and when it cannot find any more, it complements with DAG. If part of an application's code is represented by TEDs, and creates a less efficient CI than a DAG design would, then the Hybrid technique would not be able to take advantage of the better DAG implementation. We also see that for most benchmarks, Hybrid and TED techniques perform very similarly. However, for mpeg2dec, which reveals a large opportunity with the DAG technique, Hybrid can achieve higher speedups than the TED technique alone because it can benefit from the code sequences that can only be represented in a DAG.

Figure 3.4 *(j)* shows that on average across our benchmarks, TED and Hybrid achieve around 12% and 13% speedup, respectively, when using only 20% of the core area for domain-specific CIs, while DAG obtains only 4% speedup. We contrast this with Figure 3.4 *(k)*, which shows the ave-

rage when we only include application-specific CIs. While TED's canonical representation does not make a large difference when clustering code sequences within the same application, we see that it is very important to achieve higher speedups when generating domain-specific CIs. The key insight here is that individual applications are coded following the same style, so the benefit of a canonical representation is not so clear. However, as we move across applications we find different code styles and a canonical representation is key to identifying acceleration opportunities.

### 3.9.2. Domain-Specific Scoring

We next compare the four new scoring heuristics that we explain in Section 3.6. Figures 3.5 and 3.6 from *(a)* to *(i)* present a graph for each benchmark of the speedup that each heuristic predicts for a given SFU area. For these graphs, we use the Hybrid clustering technique, and include only domain-specific CIs. Note that in these and all following sections, we consider the entire SFU design and its area, not only those CIs useful per application. Thus, area always ranges between 0 and 100% of the core. The average across all benchmarks is presented in Figure 3.6 *(j)* for 100% of the area and on Figure 3.6 *(k)* we zoom in on smaller, more realistic areas of 0 to 20%.

Across all benchmarks, we see that the fourth scoring technique, or random-scaled sharing, performs best on average. In Figure 3.6 *(j)* and *(k)*, it achieves higher speedups quicker at lower areas, and at unlimited area, it performs the best. At 20% area, shown in *(k)*, this technique achieves similar speedups as scaled-by-sharing. There are some variations across benchmarks in Figures 3.5 and 3.6. For face *(d)*, the geometric mean scoring takes more area to achieve similar speedups, probably because it dampens the importance of a domain-specific CI that only performs well for one application. For djpeg *(c)*, the geometric scoring heuristic cannot achieve the speedups the other three techniques achieve, and for tmndec *(h)*, we see random-scaled sharing more than doubling the speedup of any other heuristic at any given area. For mpeg2dec *(e)*, and to a lesser extent, mpeg2enc *(f)* and tmnenc *(i)*, the geometric mean heuristic that averages the benefit each application can receive, does rise to higher speedups at lower areas. Only for mpeg2dec does the geometric mean technique get larger speedups than the random-scaled sharing heuristic at high areas. In this particular case, the geometric mean heuristic ranks a pair of CIs with low re-utilization higher compared to the other scoring heuristics. The other heuristics did not rank these CIs as high because of previously identified, partially overlapping CIs. For aacenc *(a)*, random-scaled maximizes the speedup at smaller areas. In

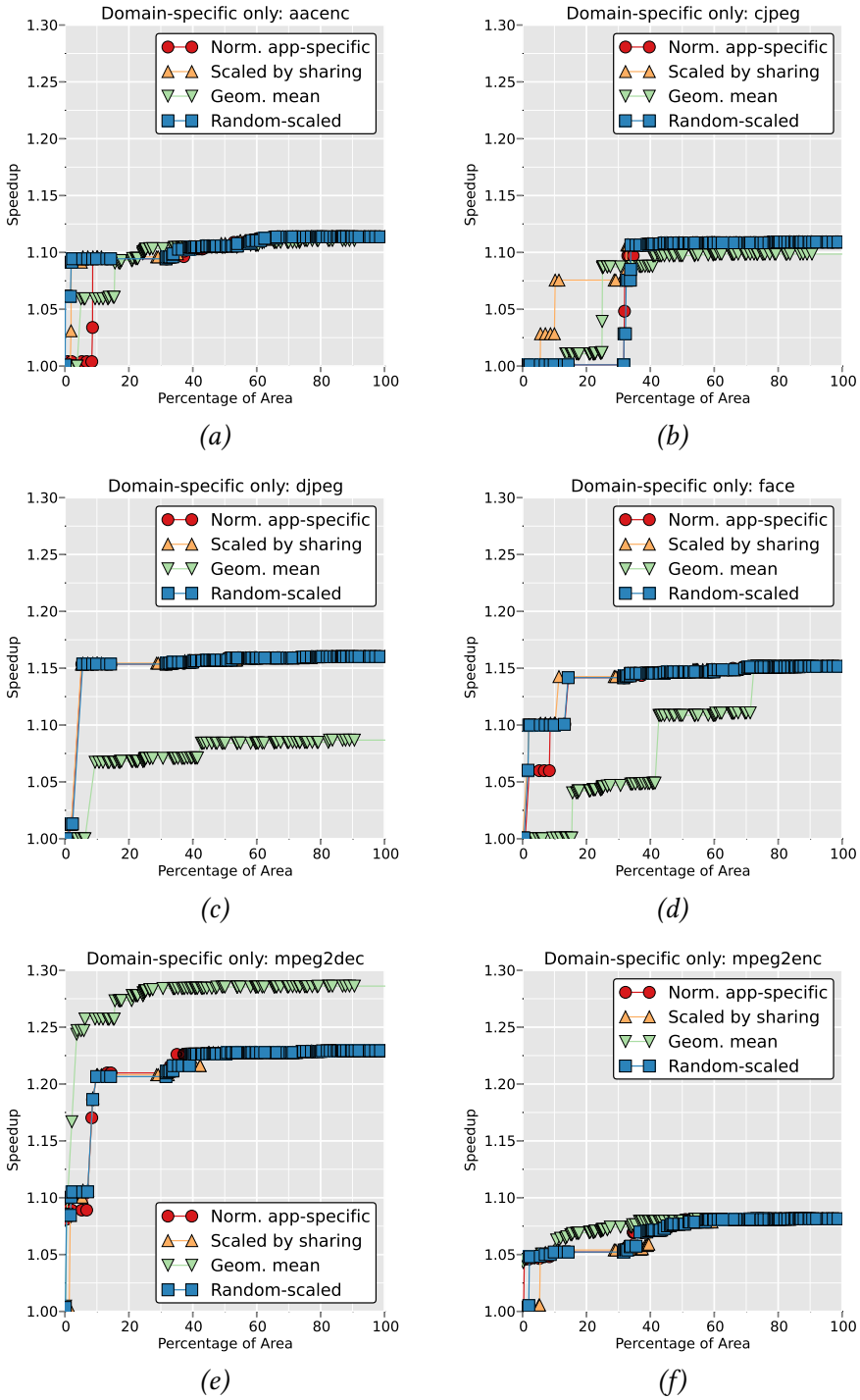


*Figure 3.5: Results of benchmark speedup versus SFU area for scoring techniques, with domain-specific CIs created with the Hybrid technique (part 1/2).*

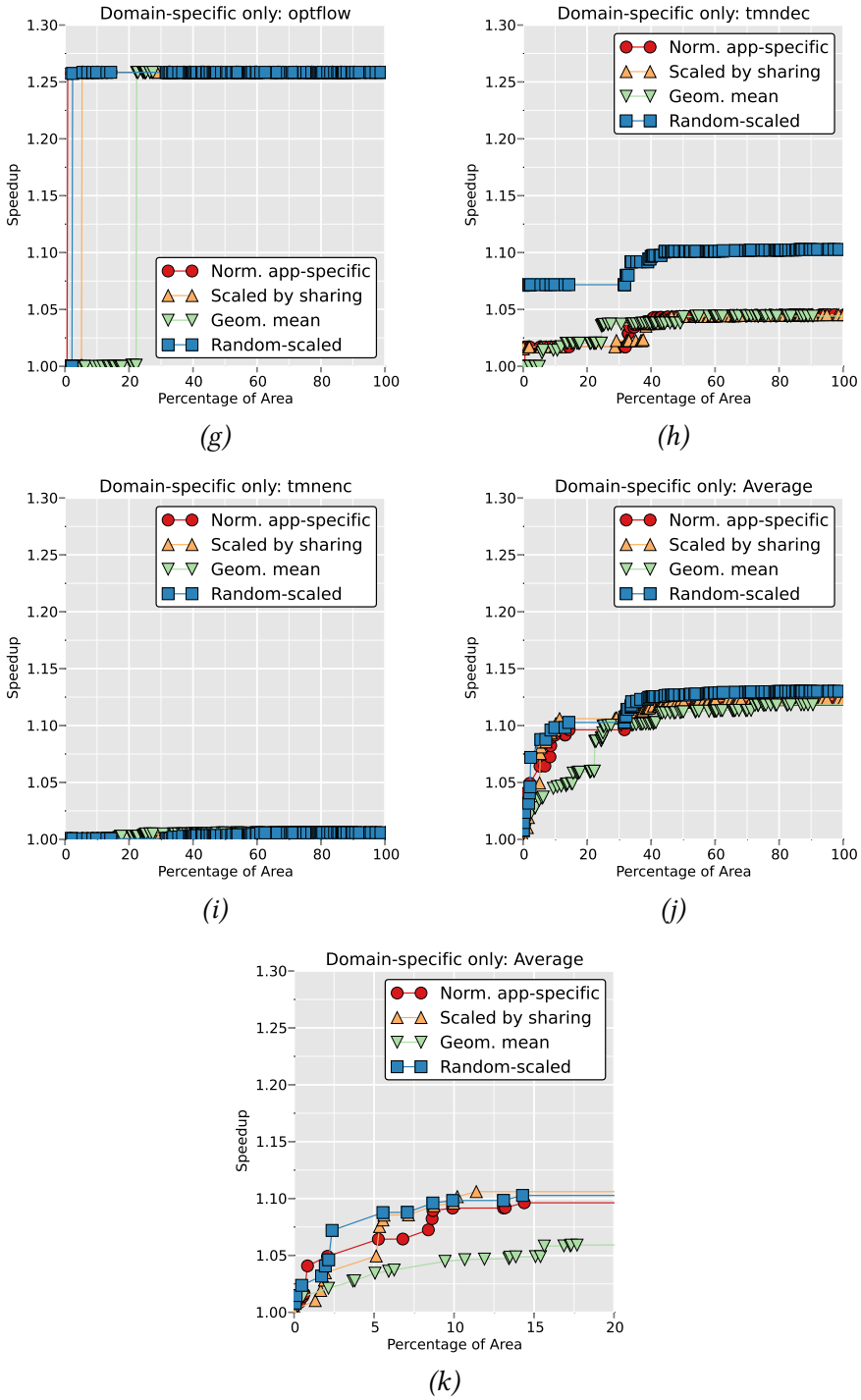


*Figure 3.6: Results of benchmark speedup versus SFU area for scoring techniques, with domain-specific CIs created with the Hybrid technique (part 2/2). Graph (j) shows the average for 100% of the SPARC core area, and (k) zooms the average in for 20% of the area.*

particular, a CI that causes a 6% speedup improvement is selected with that scoring three positions earlier than with scaled-by-sharing. However, for cjpeg *(b)*, the scaled-by-sharing heuristic is the one that raises to high speedup values at lower areas. We find here a counter-example: scaled-by-sharing selects a CI that contributes 5% to the speedup improvement five positions earlier than random-scaled. A closer look at the groups of code sequences that are clustered into those CIs tell us that in both cases the coverage across applications is maximized. However, random-scaled prioritizes less aggressively, and CIs with a medium number of applications but good overall performance will still rank high. Therefore, we use that scoring as our default in the other experiments reported in the chapter.

### 3.9.3. Application-Specific vs Domain-Specific Configurations

Up until now, we have analyzed the potential of only domain-specific CIs. But our framework allows us to compare the performance of potential application-specific CIs as well. In this section, we compare the speedups that can be achieved using a part of the core area dedicated to only application-specific, only domain-specific, or a mixture of both kinds of CIs. Our goal here is to understand how to best configure an SFU to optimize full-system performance across applications subject to area constraints. Or in other words, for a given core area, are we better off choosing application-specific only, domain-specific only, or both application- and domain-specific CIs for the SFU?

Figures 3.7 and 3.8 present the speedup for each benchmark across a range of areas, including only application-specific, only domain-specific, and both kinds of CIs. We analyze performance when the SFU takes zero to 100% of the core area. Figure 3.8 *(j)* and *(k)* show the averages across all benchmarks, using up to 100% of the core's area, and zooming in on small, more realistic areas from zero to 20%. For all of these graphs, we use the Hybrid clustering technique, and we use the application-specific scoring for application-specific CIs, and the random-scaled sharing scoring for domain-specific.

Our results reveal that, if given unlimited area, using only application-specific CIs can achieve the maximum speedup (34% on average) for our benchmarks. However, a potentially surprising result is that using both application- and domain-specific CIs together approaches the performance of using only application-specific CIs (29%), and obtains higher speedups at lower areas as compared to only application-specific. While using only domain-specific CIs limits maximal speedup to around 13%, we see that this technique is more effective than application-specific at obtaining speedups

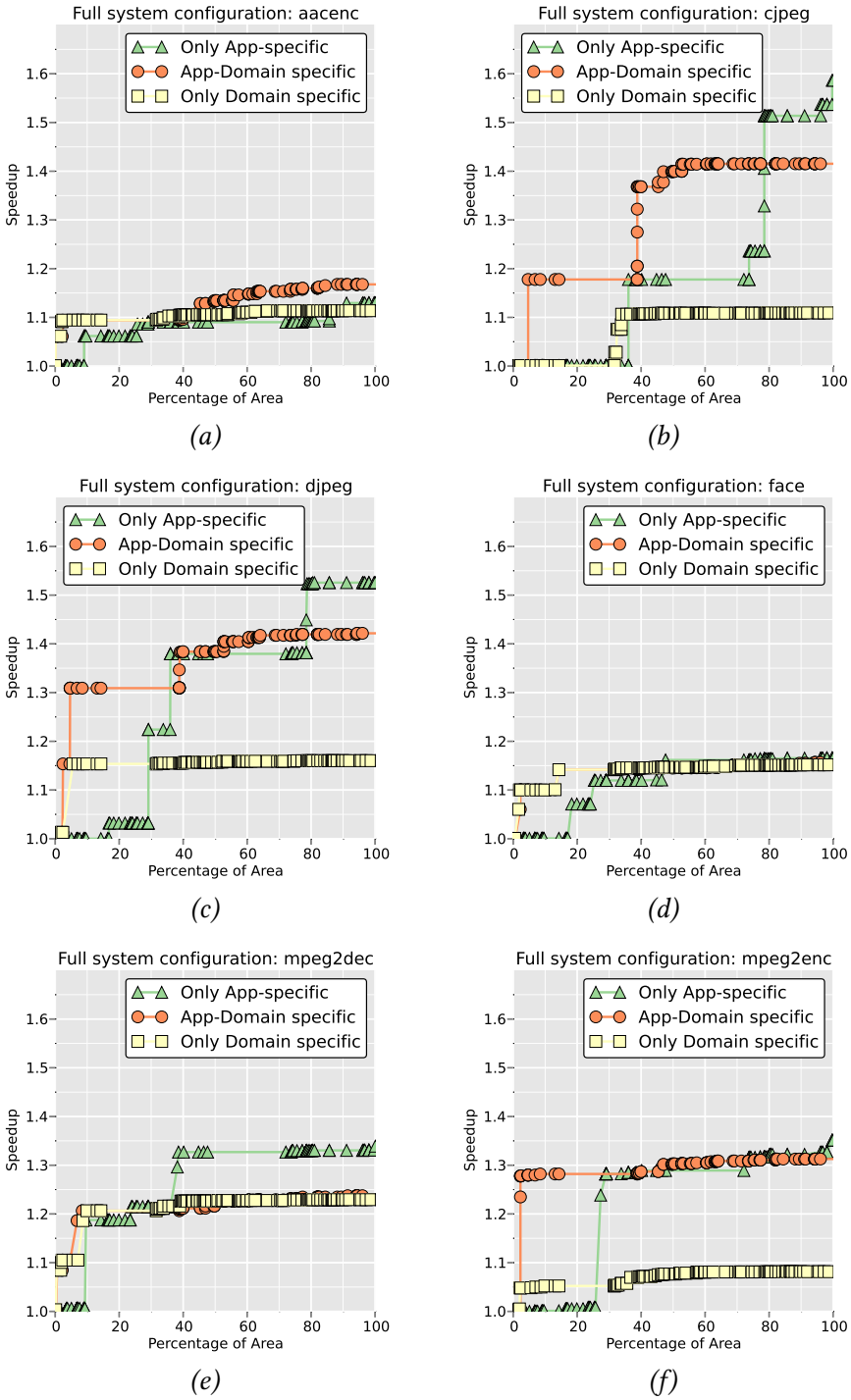


*Figure 3.7: Results of benchmark speedup versus SFU area using only application-specific, application and domain-specific, or only domain-specific CIs (part 1/2). Results gathered using the Hybrid technique.*

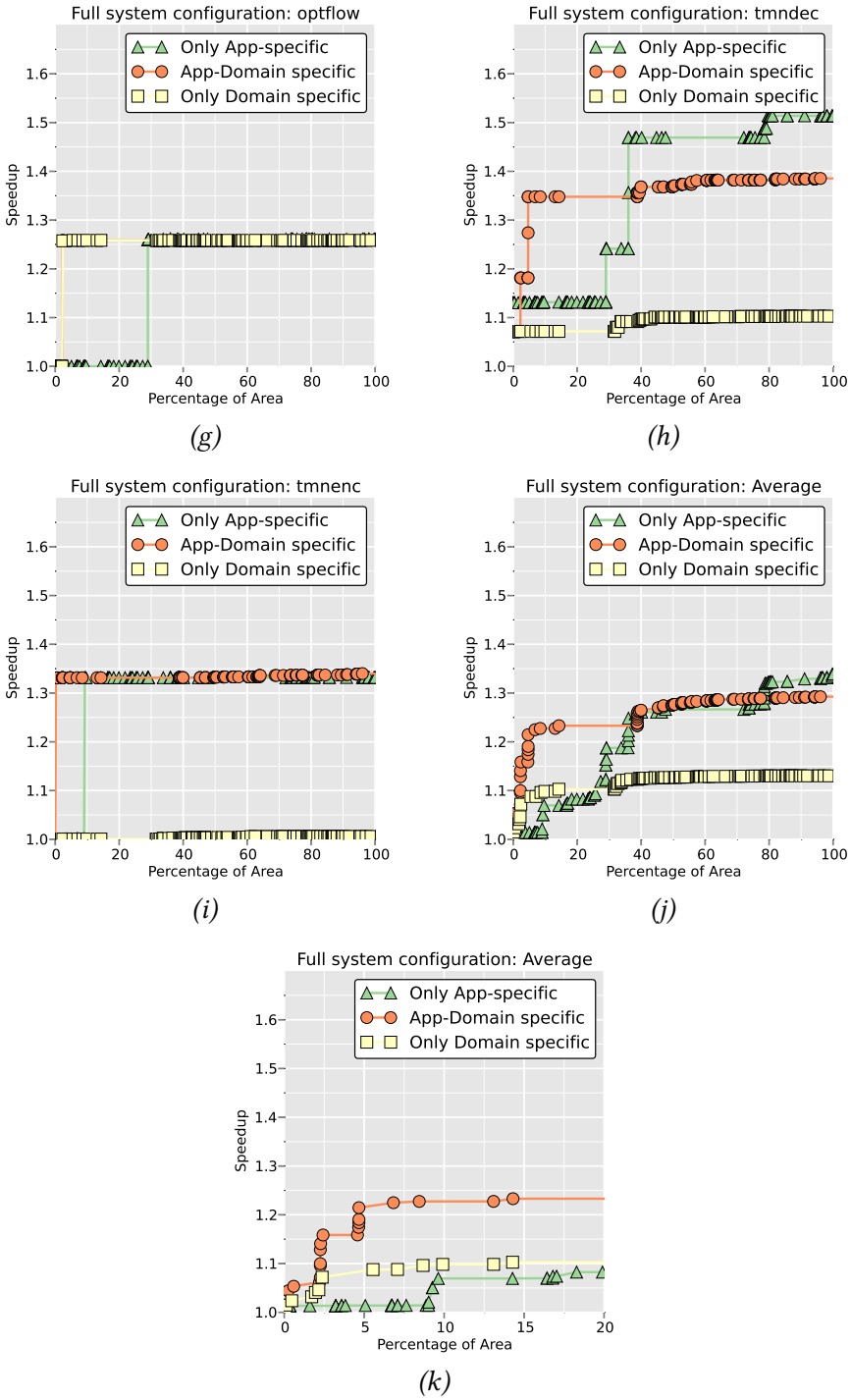


*Figure 3.8: Results of benchmark speedup versus SFU area using only application-specific, application and domain-specific, or only domain-specific CIs (part 2/2). Results gathered using the Hybrid technique. Graph (j) shows the average for 100% of the SPARC core area, and (k) zooms the average in for 20% of the area.*

at very small areas. Given 20% area, application-specific achieves 8% speedup, while domain-specific achieves 10% and both together achieve 23%. Furthermore, for several benchmarks, namely aacenc *(a)*, face *(d)*, mpeg2dec *(e)*, and optflow *(g)*, using only domain-specific CIs performs close to the best of the other two techniques.

The key insight here is that, while using only application-specific CIs results in the highest possible speedups at large or unbounded core areas, considering domain-specific CIs next to application-specific CIs yields the highest possible speedup at realistic, smaller core areas. The reason is that the domain-specific CIs benefit several applications, which is more area-efficient compared to application-specific CIs which benefit a single application only, and therefore have limited contribution to overall system performance. A corollary of this finding is that, in order for hardware acceleration to deliver substantial speedups, some notion of application-specific hardware acceleration is needed (even at small areas). This requires knowing the target domain and its applications at SFU configuration time so that some application-specific CIs can be included. Alternatively, one could devote a fraction of the SFU die area to domain-specific and application-specific CIs that are known to perform well given the applications known at design time.

### 3.9.4. Custom Instruction Analysis

In order to reveal further insight about how to build future specialized computing units, and which CIs offer the most benefit inside an application domain, we present an analysis of the CIs identified as the most effective at a few particular core areas. We compare the details of the SFU for designs with application-specific, domain-specific, and a mixture of both kinds of CIs. We show CI statistics for core area percentages 5, 10 and 15% in Table 3.3, taking the best configurations as shown in Figures 3.7 and 3.8.

Table 3.3 shows three configurations: using only application-specific CIs (*only AS*), using only domain-specific CIs (*only DS*), and using both (*AS/DS*, with the specific AS and DS portions in parentheses). We define three sizes of CIs, depending on the number of instruction primitives that each CI implements. A small-sized CI has 1 to 5 instructions, a medium-sized one has 6 to 15, and a large-sized one has more than 15. We also present the average number of inputs and outputs for each size-class; however, these do not affect the size-class (i.e., small CIs could have many inputs or outputs). Finally, we show the number of applications that each configuration can cover in the second-to-last column, and the speedup it achieves.

We can draw a few interesting conclusions from the best-performing CI configuration statistics. First, using both application and domain-specific

| %area | Config | small-sized | | | medium-sized | | | large-sized | | | #app | Speedup |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | *# CI* | *in* | *out* | *# CI* | *in* | *out* | *# CI* | *in* | *out* | | |
| 5% | only AS | 2 | 2.5 | 2 | 0 | – | – | 2 | 38 | 2.5 | 4 | 1.07× |
| | AS/DS | 6(0/6) | 5.3 | 2.2 | 2(0/2) | 10 | 5 | 6(6/0) | 26.5 | 8.2 | 9 | 1.22× |
| | only DS | 7 | 4.8 | 2 | 1 | 9 | 5 | 0 | – | – | 9 | 1.07× |
| 10% | only AS | 4 | 2.7 | 1.5 | 0 | – | – | 2 | 38 | 2.5 | 6 | 1.07× |
| | AS/DS | 8(0/8) | 5.4 | 2.3 | 4(2/2) | 11.25 | 5.25 | 6(6/0) | 26.5 | 8.2 | 9 | 1.24× |
| | only DS | 11 | 4.6 | 1.8 | 3 | 11.33 | 5.33 | 0 | – | – | 9 | 1.10× |
| 15% | only AS | 15 | 4.9 | 2.3 | 1 | 9 | 5 | 3 | 31.6 | 7 | 9 | 1.13× |
| | AS/DS | 9(0/9) | 4.7 | 1.8 | 4(2/2) | 11.25 | 5.25 | 6(6/0) | 26.5 | 8.2 | 9 | 1.24× |
| | only DS | 13 | 4.8 | 2 | 4 | 12 | 6.5 | 0 | – | – | 9 | 1.10× |

*Table 3.3: Classification of CIs in a full-system configuration of* 5%, 10% *and* 15% *of the SPARC area. (AS = application-specific, DS = domain-specific. Small = 1-5 regular instructions; Medium = 6-15 instructions; Large = >15 instructions.)*

CIs already achieves speedup of 22% using only 5% of the SPARC core's area. At the same area, using only application-specific CIs targets only 4 applications and can get only a speedup of 7%, which raises to 13% when using 15% of the core (while covering all 9 applications). Interestingly, application-specific CI configurations usually include small and large-sized CIs, but few medium-sized ones; in comparison, domain-specific CI configurations include no large-sized CIs, instead prioritizing CIs with fewer than 15 base ISA instructions. Using both kinds of CIs (AS/DS), we find more domain-specific small-sized CIs, but more application-specific ones of the large size. We also see that, though the average input and output sizes are independent of the number of regular instructions per CI, in general, the numbers of inputs and outputs grow as we go from small to medium to large-sized CIs. Interestingly, the mixed application and domain configurations include CIs from each size-class, and achieve the highest speedup. This suggests that the best-performing machine should include both application and domain-specific CIs.

### 3.9.5. Cross-Validation

In all previous experiments, we generated candidate domain-specific CIs from code sequences using the entire set of benchmarks. In this final section, we evaluate a realistic setting where the machine is configured with a set of CIs for a particular application domain, but then an as-yet-unseen application runs upon it and tries to take advantage of the flexibility of the domain-specific CIs (generally known as cross-validation). In Step 3 of our methodology, shown in Figure 3.1, we cluster code sequences from $N-1$ of our benchmarks, prioritizing using our random-scaled scoring heuristic, and then in Step 4, we evaluate the effectiveness of those CIs on a different, the $N$th, application.

Figures 3.9 and 3.10 show our cross-validation results for each benchmark from *(a)* to *(i)*, and the average across benchmarks *(j)*. When given the total core area, all but two benchmarks can reach the maximal speedup (obtained using domain-specific CIs identified over *all* benchmarks, as in Section 3.9.3, when given unlimited area). Benchmarks optflow *(g)* and tmnenc *(h)* cannot achieve their maximum speedup using our cross-validation approach; optflow achieves its speedup when using only one CI; in addition, as shown in Figure 3.8 *(g)*, optflow does achieve its maximum speedup when we include domain-specific CIs identified from all benchmarks, whereas in Figure 3.8 *(h)*, tmnenc can only benefit from application-specific CIs (achieving very limited speedup overall). The other seven benchmarks can take advantage of CIs deemed useful for the domain, and especially aacenc in

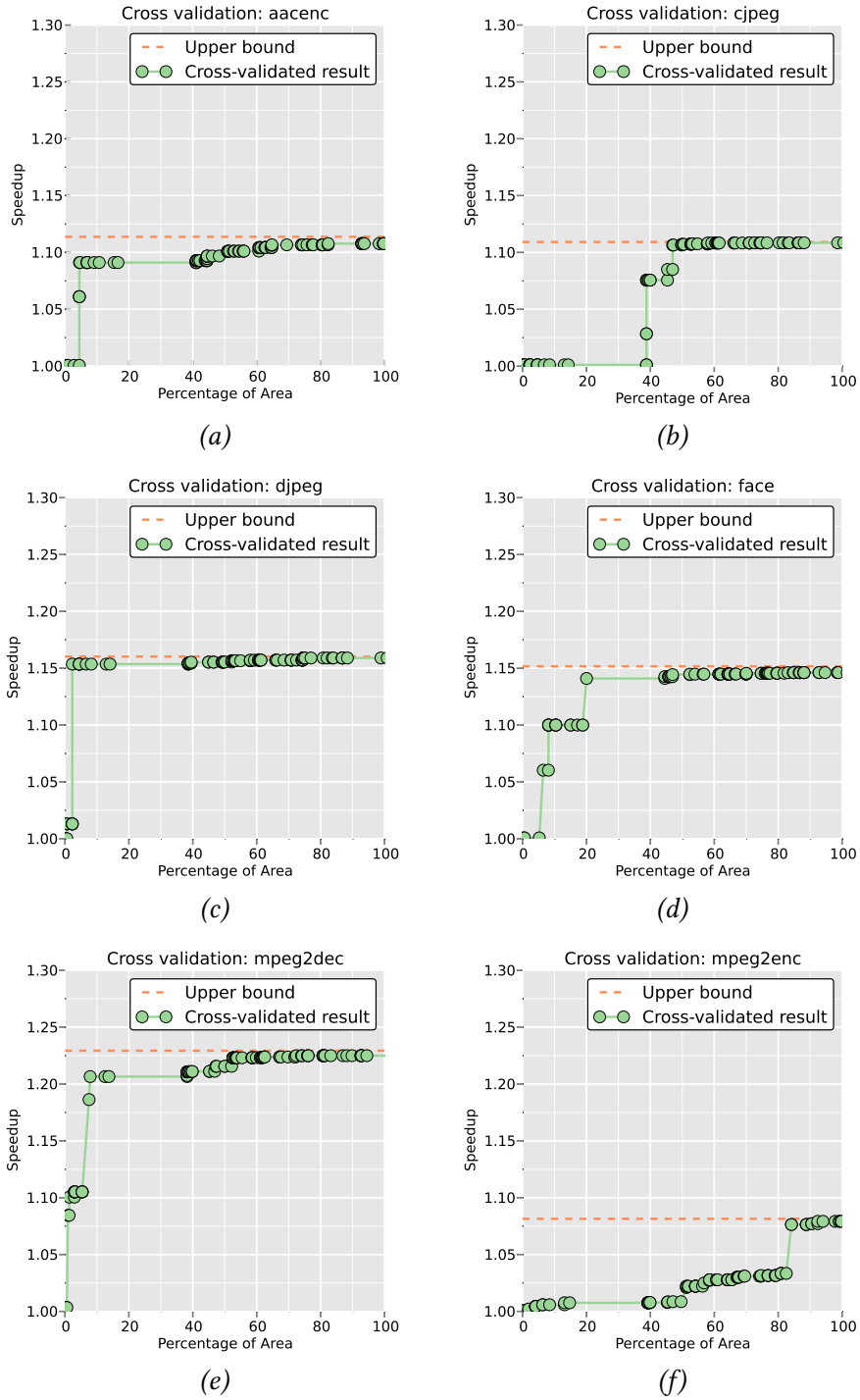


*Figure 3.9: Results of benchmark speedup versus SFU area for cross-validation per application using domain-specific CIs (part 1/2). Results gathered using the random-scaled sharing scoring and the Hybrid technique.*

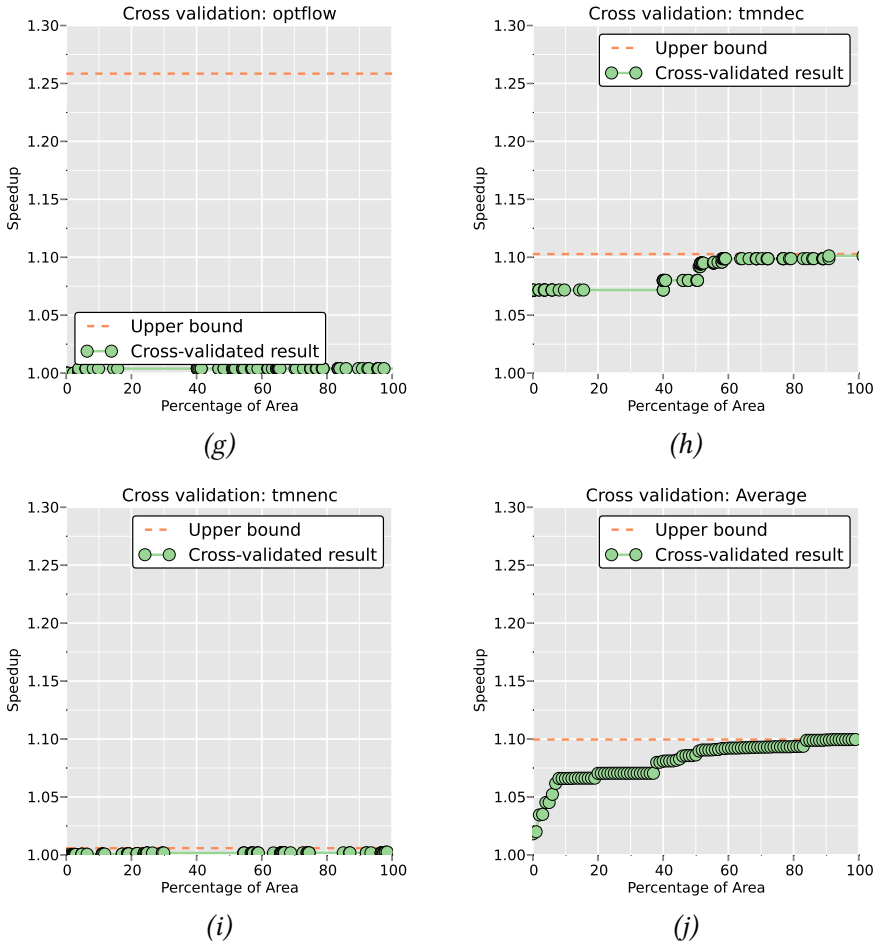


*Figure 3.10: Second part of results of benchmark speedup versus SFU area for cross-validation per application using domain-specific CIs (part 2/2). Results gathered using the Hybrid technique. Graph (j) shows the average for 100% of the SPARC core area.*

Figure 3.9 *(a)*, djpeg *(c)*, face *(d)*, mpeg2dec *(e)* and tmndec in Figure 3.10 *(h)* achieve high speedups at very low core area percentages. On average, at only 20% of the core area shown in Figure 3.10 *(j)*, our applications achieve over 7% speedup, which is a significant percentage of the maximum of 10%.

## 3.10. Summary

This chapter explores the trade-off between application-specific versus domain-specific hardware specialization, contributing with new methods to generate CIs to accelerate an application domain. We propose the use of Taylor Expansion Diagrams (TED), canonical representations of code, to identify CI opportunities. TEDs are more effective at identifying functionally equivalent code sequences across applications than the commonly used Directed Acyclic Graph (DAG) representation. We also propose a new hybrid representation that combines TEDs with DAGs and results being even more effective at finding CIs to accelerate the application domain. To be able to quickly select the potentially best domain-specific CIs during exploration, we propose scoring heuristics that take into account the frequency of CI use both within and across applications. We use the canonical representations and our scoring heuristics in our CI design framework FuSInG (Functionally Similar Instructions Generator), along with performance and area estimations. We find that while application-specific CIs result in the highest possible performance at large or unbounded core areas, including domain-specific CIs yields the highest possible speedup at small, more realistic core areas. This finding underlines the need of domain-specific instructions for practical and flexible hardware specialization, that we explore further in the coming chapters. In addition, we demonstrate that the identified CIs using our exploration framework are effective for previously unseen applications within the same domain, making specialization more generally applicable.

# 4

# Partially Similar Domain-Specific Instructions

## 4.1. Introduction

In the previous chapter, we studied the trade-offs between application-specific and domain-specific CIs. We showed that application-specific CIs had the highest performance at large or unbounded core areas; however, including domain-specific CIs results in the highest possible speedup at small, more realistic core areas. But code sequences analyzed in the previous chapter were limited to a single basic block, which leads to moderate speedups. CI exploration both across applications and beyond the basic block level is challenged by the difficulty of finding exact matches of code sequences. Therefore, in this chapter, we explore specialization for a domain of applications in greater depth, still focusing on identifying CIs that are accelerated in hardware in a domain-specialized functional unit (DSFU) that extends a general-purpose processor, but also extending the acceleration opportunities across basic blocks through partial matching of different implementations of code sequences.

Also, in the previous chapter, CIs are selected using domain-specific heuristics based on performance indicators. While speedup is a well-known metric used in CI design, a configuration that extends a low-power processor must try to balance energy as well. Hence, the selection proposed in this

chapter focus not only on high performance, but also on energy efficiency.

We build the automated framework MInGLE (Merged Instructions Generator for Large Efficiency) to identify fruitful CIs across a set of applications from a domain. While this search space can grow exponentially, we develop steps to tractably generate a set of potential CIs by preferably merging those with high similarity. We first use profiling to extract hot loops from the applications, and then we use high-level synthesis to gather execution time and hardware area measurements for several implementation versions of the potential CIs. Next, MInGLE transforms the sequences into a *Merging Diagram*, a canonical representation to facilitate similarity identification, and merges CIs that could be executed in the same DSFU pipeline to reduce specialized area. We cluster CIs to identify not only those that have exact functional similarity but also those with partial similarities that could cover more code while reducing the needed area for the DSFU. Finally, MInGLE selects a set of CIs that fit into a particular hardware area, maximizing energy efficiency and performance speedup across the applications. We demonstrate the effectiveness of the framework using 11 media benchmarks in the context of a superscalar in-order processor.

The outline of this chapter follows. Section 4.2 illustrates the context and motivation of the work behind the framework, which is presented in Section 4.3. Following sections describe each one of the main parts of the framework: CI identification and implementation (4.4), Merging Diagram definitions and construction (4.5), partial merging of CIs (4.6), and selection (4.7). Computational complexity is discussed in Section 4.8, and results on the evaluation of the framework are in Section 4.9. We close this chapter with Section 4.10.

## 4.2. Context and Motivation

The CIs we target in this chapter are executed on a domain-specialized functional unit (DSFU) to accelerate a domain of applications. The DSFU is integrated within the low-power processor core's datapath, as in the model of Section 2.3.1 in Chapter 2. Figure 2.4 shows a diagram of the processor extension. Deployment of DSFUs is more effective than specializing a complete processor and they are easier to program than bigger off-core accelerators. However, this kind of acceleration presents several challenges in existing design methodologies.

With a limited hardware area for implementation, we want to maximize the CIs' utilization. We can achieve this by targeting regions of code beyond basic blocks, although we must keep the number of data transfers from and to the DSFU limited to avoid high transfer overhead. Going beyond the ba-

| Benchmark | ID | Implementation | % area | % EDP improv. | |
|---|---|---|---|---|---|
| | | | | cjpeg | gsmdec |
| cjpeg | ci1.1 | no unroll | 0.0020 | +5.3 | −1.0 |
| | ci1.2 | unroll 4 | 0.0080 | +7.1 | −1.0 |
| gsmdec | ci2.1 | unroll 4 | 0.0013 | −1.0 | +218.7 |
| | ci2.2 | unroll 8 | 0.0027 | −1.0 | +290.6 |
| cjpeg+gsmdec | mci1 | ci1.1 + ci2.1 | 0.0029 | +4.5 | +217.0 |
| | mci2 | ci1.2 + ci2.2 | 0.0087 | +6.2 | +227.0 |

*Table 4.1: Percentages of area occupancy and EDP improvement for different CI implementations.*

sic block level is key to improve performance and justify the design effort of CIs, especially if the prototyping platform is an FPGA, which is reported to run a circuit implementation up to 4.6x slower than its ASIC equivalent [54].

Identification of CIs for a domain is challenging, because we must find similar code patterns that repeat across applications to improve hardware reusability. We've seen that while commonly used DAGs hold the exact structure of a program, a canonical diagram represents the program's functionality, thus exposing common functions across applications that can become the same CI. In this chapter, we also use a canonical representation, but extending the CI beyond the basic block and adding partial matching to improve reusability.

We also aim to share common operations of sequences of instructions in order to take up less hardware area. For instance, the functions $F1 = a+b+c$ and $F2 = a*b+c$ can be collapsed into a single instruction that shares the circuit of one addition, and selects between an addition and a multiplication. This sharing greatly expands the amount of code that can be accelerated, and greatly reduces the hardware area needed for specialization.

We want to consider several implementations of each CI as part of the CI exploration as well; i.e., several unrolling factors and vectorization, since they offer divergent trade-offs and benefits. Consider, for instance, the CIs listed in Table 4.1. For each CI, we show the benchmark where it was extracted, the ID, implementation details, the percentage of area it takes on a Virtex 7 FPGA and the energy-delay product (EDP) improvement (higher is better) of each application when that CI is implemented in the DSFU. The

first four rows are application-specific CIs, while the last two ones merge the previous CIs into domain-specific ones. By exploring different implementations, we can vary the choice of which one to include depending on the available area and potential EDP gains. Note that different implementations present the additional challenge of a bigger search space. We try to avoid exponential search algorithms, keeping the execution time of the framework linear with the size of the search space.

## 4.3. MInGLE Framework

Figure 4.1 shows a high-level diagram of our automated framework MInGLE, which is throughly explained in the coming sections. Starting with a set of applications from a domain, we first detect and enumerate potential CI candidates based on profile information (Step 1, *Candidates Extraction*). This chapter's contributions are implemented in subsequent steps, with the double objective of generating energy-efficient CIs across a domain of applications, while making the exponential search space tractable. In the next step (*Canonicalization*, we transform CI variants expressed in the compiler's Intermediate Representation (IR) into a canonical representation: Merging Diagrams. Then, in Step 3 *Merged CIs Generation*, we first calculate the pairwise distances used in the identification of similarities between CIs. Because we use a canonical representation and create a global ordering of variables, that step is computed quickly and efficiently. After this, the clustering allows the framework to do both exact and partial matching of CI variants, the latter enhancing the CI reutilization across applications. The *Merging Estimation*, together with the *Performance and Energy models*, quantify the advantages of the generated CIs, estimating the new area, energy and speedup of each clustered group of variants. Finally, Step 4 (*CI Selection*) solves the optimization problem of fitting the best group of candidates, that save the most energy across the domain, into a limited area.

## 4.4. Candidates Extraction: From Application Code to Hardware Acceleration

In the *Candidates Extraction* step of MInGLE (upper side of Figure 4.1), we first profile each of the input applications, identifying their hot loops in step *Profiling* (1.1). We extract those hot loops' bodies as isolated code that we can execute as new CIs in step *Extraction/Slicing* (1.2). As our target CIs operate on data transferred from and to the register file, there is a transfer time before the execution starts and when it ends. Thus, memory operations

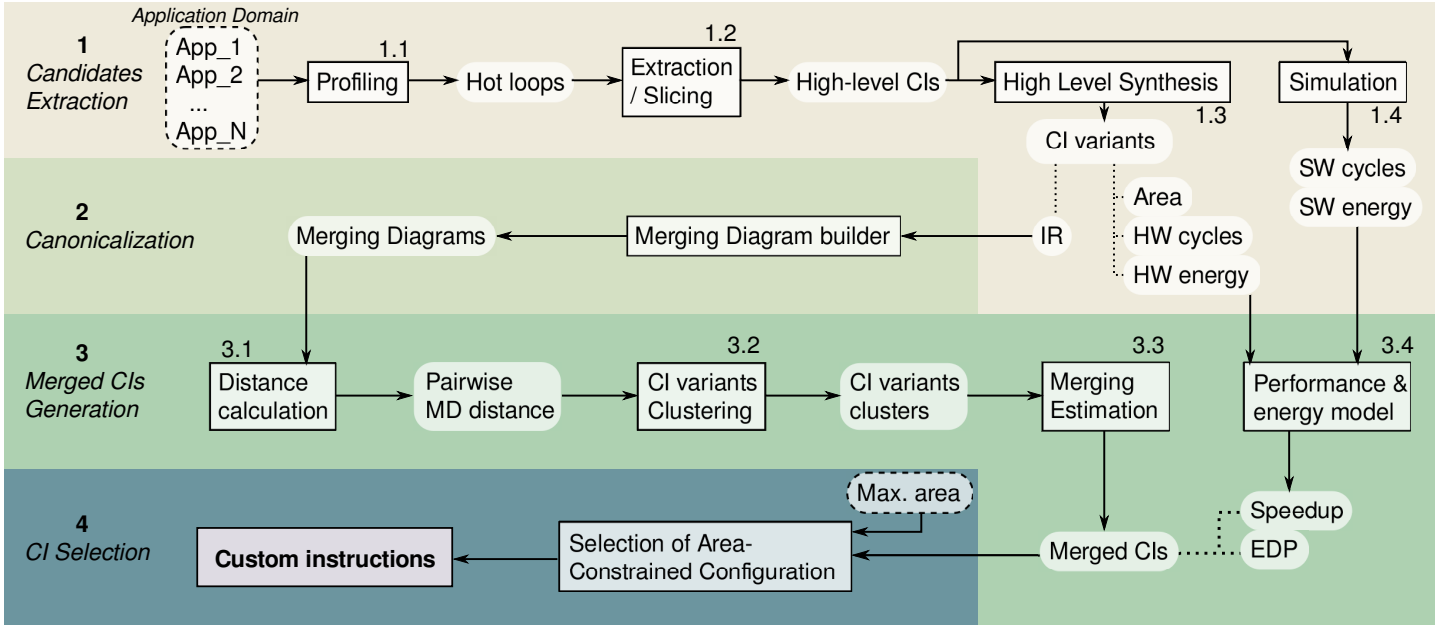


*Figure 4.1:* MInGLE *framework for the implementation and generation of partially-merged CIs.*

are sliced and placed before and after the loop body computation. In step *Simulation* (1.4), we simulate the applications with the identified high-level CIs to measure cycles and energy consumption in the baseline processor. In step *High Level Synthesis* (1.3), we implement CIs in hardware, obtaining their area occupancy, execution cycles and energy, and their Intermediate Representation used in subsequent steps. We apply different unrolling and vectorization factors in the HLS transformation. Therefore, besides the implicit instruction-level parallelism of the CIs, we also have potential data-level parallelism from the HLS optimizations. From now on, we talk about a **CI** as the high-level representation of a loop body or inline function that can be accelerated in hardware, and we talk about **CI variants** or only **variants** to specify distinct implementations of a CI (for example, with different unrolling factors). Thus, depending on the optimizations applied, we can obtain several variants of the same CI. The *Candidates Extraction* step produces application-specific CI variants with their implementation details.

## 4.5. Canonicalization of Custom Instructions using Merging Diagrams

Identifying similarities between CI variants in a non-unified representation is difficult due to the amount of unnecessary information a modern compiler IR includes. Also, a representation such as a DFG, which expresses structural relations between operators, does not expose functional similarities, since different coding styles among applications may hide them. Therefore, in step *Canonicalization* of Figure 4.1, we transform the codes of the CI variants expressed initially in a compiler IR, into an abstract, canonical representation: the *Merging Diagram (MD)*.

The MD represents arithmetic and logic operations (within the basic block), and predicate information (at the loop level), both with unrestricted number of inputs and outputs. Its representation is partially based on Binary Decision Diagrams (BDDs) and Taylor Expansion Diagrams (TEDs), surveyed in Sections 2.4.2.1 and 2.4.2.2, respectively. We have successfully used TEDs for CI similarity detection within a basic block in Chapter 3, but extending CIs beyond the basic block level needs a new representation that includes predication. Also, the codes we process include operations that cannot be expressed as polynomial functions, which are the base of the TED representation. The following definitions explain the details of our new representation, which include both modified versions of TEDs and BDDs.

**Definition 1**. *An Augmented TED (AugTED), is a directed acyclic graph based on linearized and reduced TEDs. It is composed of a labeled set of*

*nodes $V$, a weighted set of edges $E$, and the terminal node* 1. *In normal TEDs, $V$ represents variable names and $E$ are additions/subtractions or multiplications. AugTEDs expand TED nodes to represent any kind of computation, using variable renaming. Here, labels in $V$ can be integer, float or special. Integer and float labels represent variable types, and special labels a function that cannot be represented by a Taylor expansion.*

**Definition 2**. *A Linking BDD (LinBDD) is a directed acyclic graph based on reduced and ordered BDDs. It consists of a labeled set of nodes $V'$, a set of edges $E'$ and terminal nodes* 0 *and* 1. *LinBDDs nodes have BDDs' 0-1 decision edges, and additionally a third edge $Link$ that references an outside diagram, namely an AugTED. A LinBDD is constructed with the Shannon expansion of boolean functions created with the if-then-else (ITE) operator: $ITE(I, T, E) = I \cdot T + \bar{I} \cdot E$.*

**Definition 3**. *A Merging Diagram is a data structure that provides a canonical representation of a predicated code region. It consists of a set $A$ of AugTEDs that represent computations and a set $L$ of LinBDDs that represent control flow execution. Link edges from the nodes in each member of $L$ references a member in $A$.*

Figure 4.2 (d) shows an example of an MD for a given code sequence. The left part of the MD is a LinBDD and its nodes are linked to AugTEDs on the right by *Link* edges. There is a special label ($SA(slt)$) that stands for a relational operator that cannot be expressed by Taylor expansions. Details on the construction of the MD, with explanation of the example of Figure 4.2, follow.

### 4.5.1. Merging Diagram Construction

To build all the canonical MDs of a group of CI variants, we follow the steps of Algorithm 1. We start processing in lines 3 – 7 the set of IRs of all the CI variants' code regions. Figure 4.2 (a) shows an extract of an IR example with arithmetic, relational and conditional selection instructions. For each one of the IRs, we extract the polynomial representation of the computations (*PolyAugTED*) and the branch predication (*PolyLinBDD*) of the code, as illustrated in Figure 4.2 (b). With those base polynomials, we establish a precise renaming of variables that unifies their name space in lines 8 – 22, which facilitates fast similarity identification in Section 4.6.1. We decompose each polynomial into its monomials, and we rename each variable based on the type of monomial where it is found. We find primarily adding and multiplying types of monomials, but also cover floating-point

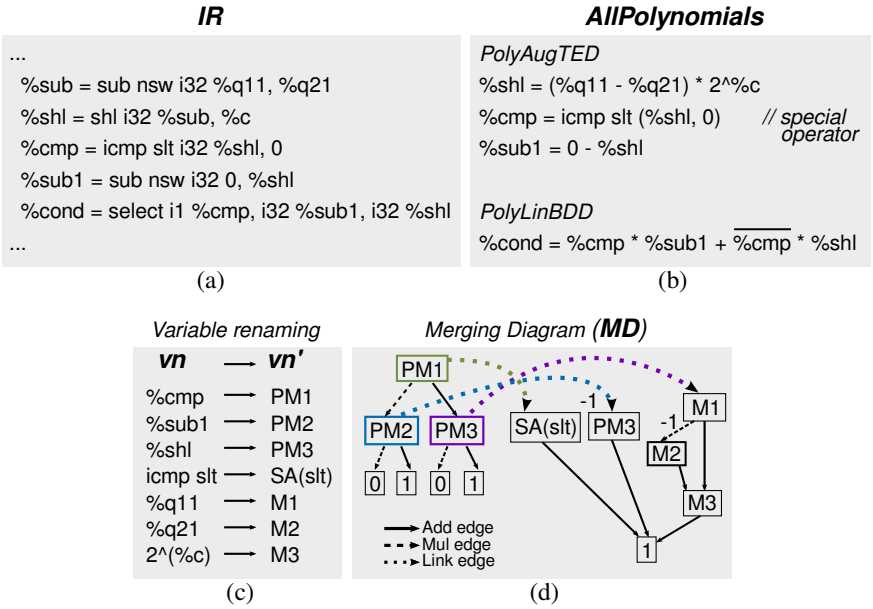


*Figure 4.2: Example of Merging Diagram construction. For the IR on the top left (a), polynomials are extracted (b), variables renamed (c), and an MD is created (d).*

and predicated types. For instance, in Figure 4.2 (c), variables are renamed as $A$ (adding) and $M$ (multiplying) preceded by $P$ (predicated) or $S$ (special).

Then, in lines 23 – 25 we define a strict variable ordering to perform the expansions, common to all variables implicated. As we have multiple polynomials that expand with the same group of variables, we first set variables in ascending order based on the number of times they occur. This ensures that we will have a minimum number of expansions, resulting in a more compacted MD. For the same reason, in the case of a tie in the number of instances between multiplying and adding variables, we prioritize the former ones.

Finally, in lines 27 – 31, for each rewritten polynomial, we create an MD structure with a condensed matrix *Diagram* that contains all the nodes and edges from the AugTEDs and LinBDD; it is thus of size $s \times s$, with $s$ the precomputed size of all the variables involved. *Link* edges are though kept apart in a two dimensional array. Following the variable ordering, we build the MD expanding each term recursively as it is done regularly with TEDs and BDDs. We show in Figure 4.2 (d) the resulting representation, which is still canonical for the assumed variable order, as it is the case for regular TEDs and BDDs.

**Algorithm 1**: Merging Diagram construction

**input** : Array of CIs' IR codes $AllIRs$
**output**: Merging Diagrams $AllMDs$

Array $AllPolynomials, RewrittenPolynomials \longleftarrow \emptyset$
2D array $RenamedMap \longleftarrow \emptyset$
**for** $IR \in AllIRs$ **do**
    $PolyAugTED \leftarrow ComputationPolynomials(IR)$
    $PolyLinBDD \leftarrow PredicationPolynomials(IR)$
    add $(PolyAugTED \cup PolyLinBDD)$ to $AllPolynomials$
**end**
**for** $p \in AllPolynomials$ **do**
    $M \leftarrow GetMonomials(p)$
    **for** $m \in M$ **do**
        $MonomialType \leftarrow GetMonomialType(m)$
        $VarNames \leftarrow GetVariablesNames(m)$
        **for** $vn \in VarNames$ **do**
            **if** $vn \notin RenamedMap$ **then**
                $vn' \leftarrow renameVar(vn, MonomialType)$
                add $< vn, vn' >$ to $RenamedMap$
            **end**
        **end**
    **end**
    $p' \leftarrow replaceVars(p, RenamedMap)$
    add $p'$ to $RewrittenPolynomials$
**end**
$VarsOccurrences \leftarrow$
$countOccurrencesVars(RewrittenPolynomials)$
$OrderedVars \leftarrow ascendingOrderVars(VarsOccurrences)$
$s \leftarrow$ size of $OrderedVars + 1$
Array $AllMDs \longleftarrow \emptyset$
**for** $p' \in RewrittenPolynomials$ **do**
    $MD \leftarrow <$ *Diagram*: $s \times s$ array, *Link*: 2D array>
    $MD.Link \leftarrow linkToAugTEDVars(p', RenamedMap)$
    $diagramExpansions(p', MD.Diagram, OrderedVars)$ add
    $MD$ to $AllMDs$
**end**
**return** *AllMDs*

### 4.5.2. Global diagram of variants

In order to cut down computation costs in later steps it is required to have a diagram that represents the entire design space of CI variants. To do so, we combine all the AugTED and LinBDD polynomials to obtain a global MD unified representation. For each variant, we locally rename its polynomial variables, saving the naming convention and number of instances in a global structure. Then, based on that locally collected information, we produce a global variable ordering that is fixed for the design space. Finally, MDs are produced individually for each variant with the global ordering.

## 4.6. Generation of Merged Custom Instructions

### 4.6.1. Distance Calculation

In step *Distance Calculation* of *Merged CIs Generation* (step 3.1 in Figure 4.1), we need to establish a concrete metric that measures similarities among CIs to guide the subsequent clustering step of the MInGLE framework. We therefore develop a new way to measure how different two CI variants are in terms of their functionality, using the MD.

We perform a distance calculation for pairs of MDs of variants that do not implement the same loop body, $CI_X$ and $CI_Y$. We use the previously built global diagrams to speed up this calculation. If we would not have the global, uniformed variable space that we obtained in Section 4.5.2, we would have to build a pair of diagrams for each pair of CIs being compared, which would be computationally very expensive. Thus, based on the prebuilt global diagrams, we obtain the number of AugTED-operations and LinBDD-branches that in $CI_X$ do not match with those in $CI_Y$, namely $nM_X$, and vice versa, $nM_Y$. An MD node $v_x$ matches another MD node $v_y$ if their labels and out edges also match. The matching information is kept for the merging step explained below in Section 4.6.3. We also count the number of total AugTED and LinBDD nodes that each MD variant has – $Tot_X$ and $Tot_Y$. Then, we compute the distance $\delta$ as:

$$\delta(CI_X, CI_Y) = average\left(\frac{nM_X}{Tot_X}, \frac{nM_Y}{Tot_Y}\right). \tag{4.1}$$

One-to-one distances are saved in a condensed distance matrix.

### 4.6.2. Clustering Custom Instruction Variants

For domain-specific acceleration, merging CIs reduces energy consumption because we need less implementation area; to put it another way, it im-

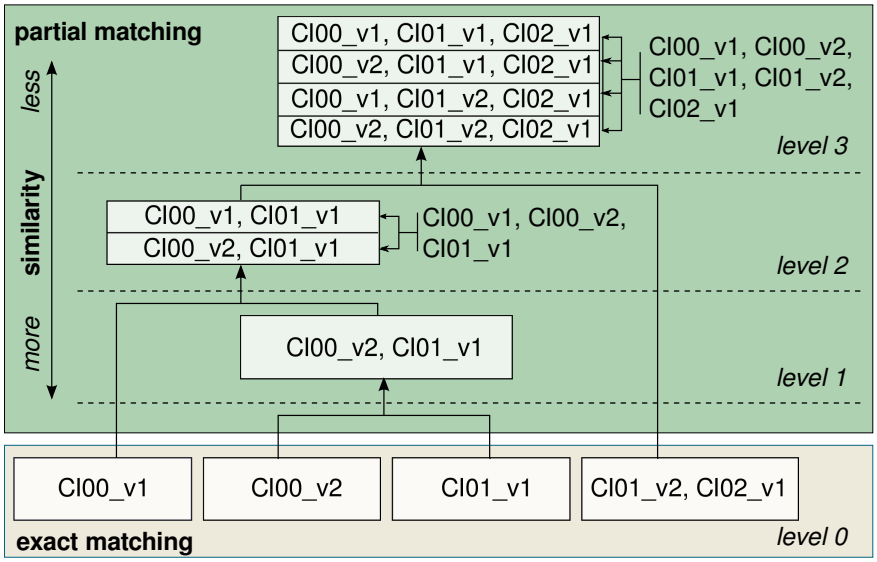


*Figure 4.3: Hierarchical clustering of CIs. Exact matching instructions are found at the bottom, while nodes closer to the root group are increasingly less similar CIs. CIXX_vy: CI with identifier XX and implementation variant y.*

proves performance since we can allocate more CIs in the constrained area. We have to merge circuits of CIs that have more in common to maximize area reduction, as well as minimize the implementation overhead due to circuit multiplexing. However, with the huge set of CI variants that we obtain when we work with multiple applications, it is prohibitive to try all the possible combinations of CIs that could be grouped together. Therefore, in step *CI variants Clustering*, we group CIs based on a hierarchical clustering that organizes groups by more to less functional similarity, cutting down the search space to avoid those groups that are not similar enough to be worth implementing together.

Distances between variants help to quickly decide which ones are better to merge together to reduce energy consumption. Using the distance matrix computed in the previous step, we create clusters of CI variants. We perform hierarchical, agglomerative clustering of CI variants, obtaining a dendrogram, a tree-like structure, as shown in Figure 4.3, where tree leaves represent **exact** matches and internal nodes denote **partial** matches. Starting from the baseline CI variants, we form exact-matching clusters based on the distance matrix (leaves – level 0 in the figure). Then, distances between the newly formed clusters use the *complete* method to determine the agglomerative distance, that is, the maximal distance between any two vari-

ants in the cluster (levels 1 to 3, to the root). From leaves to root, we find different versions of merged variants, ordered from more to less similar.

Some obtained clusters may include variants that target the same CI. In Figure 4.3, level 0 includes two variants of the same CI: *CI00_v1* and *CI00_v2*; a variant of *CI01*, and *{CI01_v2, CI02_v1}*, that is the exact matching of two different implementations of two different CIs. Level 1 has the cluster *{CI00_v2, CI01_v1}*, which has the maximum similarity for partial matching. Variant *CI00_v1* from level 0 is clustered at level 2 with *{CI00_v2, CI01_v1}* from level 1. However, as a merged variant cannot implement a concrete CI more than once, we produce different versions that do not duplicate the loop body (*CI00* or *CI01*) within the clusters where this problem occurs. Thus, at level 2 we generate two solutions: *{CI00_v1, CI01_v1}* and *{CI00_v2, CI01_v1}*. Since the latter already exists at level 1, we will eventually discard it, although its information is still used to generate the cluster at level 3. Note that this can induce an explosion in the number of solution clusters for a given level. In case of many cluster versions, we select a reduced group chosen heuristically based on an approximation of expected EDP improvement.

### 4.6.3. Merging Estimation and Modeling

With the clustering formation, we obtain a bigger set of CI variants, some of which are merged to save area. In step *Merging Estimation*, we estimate the new hardware area occupancy, performance and energy gains of merged variants in order to run the selection step with accurate information.

Based on the information from the distance calculation (Section 4.6.1) of non-common matches between each pair of variants, we obtain the area consumption of operators that are shared (*shared*) and of those that are not (*non_shared*). For sharing logic, we need to introduce multiplexers that will induce an extra area cost, *overhead*.

Therefore, we calculate the area $a_i$ of a merged CI variant $i$ as:

$$a_i = overhead_i + shared_i + \sum_{j=1}^{N} non\_shared_{ij}. \tag{4.2}$$

Then, in step *Performance and Energy Model*, we model first the performance of an accelerated application. We start obtaining the cycles $c\_l\_SW$ that a hot loop iteration takes to execute in the baseline processor, excluding memory operations, from simulation. We also obtain the number of iterations $N\_it$ of that loop for a given execution of the benchmark. From hardware synthesis, we get the number of cycles $c\_HW$ that a CI variant

takes. We calculate the cycles $c\_T$ to transfer data to the DSFU local memory as a function of the input data size.

With the previous data we obtain the cycles we save executing a CI variant as:

$$c\_saved = (c\_l\_SW - (c\_HW + c\_T)) \times N\_it. \tag{4.3}$$

We calculate the new number of application cycles as:

$$App\_cycles = c\_total\_SW - c\_saved, \tag{4.4}$$

with $c\_total\_SW$ as the application cycles without CIs.

Finally, the modeled energy consumption of an application that uses CIs is calculated as:

$$E_{app} = E_{baseline} + E_{CI}, \tag{4.5}$$

with $E_{baseline}$ the baseline processor's energy model and $E_{CI}$ the CI energy consumption.

The latter is modeled as the sum of its dynamic and static components:

$$E_{CI} = P_{dynamic} \times T_{CI} + P_{static} \times T_{total}, \tag{4.6}$$

where $P_{dynamic}$ and $P_{static}$ are, respectively, the dynamic and static power of the hardware components that implement the CI variant, $T_{CI}$ is the time that the CI is active, and $T_{total}$ is the execution time of the application calculated from $App\_cycles$.

## 4.7. Custom Instruction Selection for an Area Constrained Configuration

Implementation area is an expensive commodity in our low-power target that largely influences the energy consumption of the final design. However, performance gains also play an important role, because a faster running application would generally consume less energy. Therefore, in the last step of the framework (*CI Selection*), we address the performance and energy trade-off when choosing the best fitting set of CI variants for a given hardware area. We model this optimization as a Knapsack problem, in which we try to fit a subset $S$ of a collection of objects $C$ – each object $o_i$ with an intrinsic value $v_i$ and weight $w_i$ – within limited mass $M$ so that the sum of the values of the final subset is maximized and the sum of the weights does not exceed $M$. In our case, we try to fit the $n$ CI variants, merged and not merged, within a limited hardware area $A$. Each $c_i$

candidate has a value $v_i$ that we describe later, and a hardware occupancy, $hw_i$. We have an additional requirement in our problem: as each CI can be selected only once, though it can be implemented by different variants – with distinct unrolling factors, or merged with other instructions – once we select one CI variant, all other variants of the same CI are invalidated for the following selection steps.

We model our problem with Mixed Integer Linear Programming (MILP). We define the following constraints:

$$\sum_{i=0}^{n} c_i \times hw_i \leq A, \tag{4.7}$$

$$\sum_{i=0}^{n} lb_i \leq 1, \tag{4.8}$$

with $lb_i$ a loop body that can be implemented by ($n$) CI variants. Therefore, for a given loop body, only one of its CI variants will be selected.

As our main goal is to accelerate execution and save energy, our objective function tries to maximize the EDP improvement. However, the total EDP value changes depending on the area occupancy, and thus, it cannot be deterministically precomputed before the selection starts. Though, for each potential CI we can calculate an approximated value of the EDP difference with respect to the baseline processor without the CI. Also, the static energy component of the EDP is subject to the known value of the maximum area $A$, which is an approximation for the value that we want to maximize. Therefore, we define the objective function as:

$$\sum_{i=1}^{n} c_i \times \sigma\_EDP_i \rightarrow max. \tag{4.9}$$

The metric $\sigma\_EDP_i$ of a concrete CI variant is the value $v_i$ in the original Knapsack problem and we calculate it as:

$$\sigma\_EDP_i = \sum_{j}^{B} \|\sigma\_EDP_{ij}\| \times (1 + \sigma\_A_i \times A_i), \tag{4.10}$$

where $B$ is the number of applications that the current variant targets; $\|\sigma\_EDP_{ij}\|$ is the original application $j$'s EDP minus the EDP with the variant, normalized to the observed maximum for that application; $\sigma\_A_i$ is applicable only to merged variants, since it is the percentage of area we save by merging and $A_i$ is the percentage of the total area that the variant

takes. We find that this metric selects more medium-sized variants that help to save area occupancy, and have lower overhead and lower static power than larger variants. From experimentation, we confirm that this objective gives stable results and maximizes EDP fairly among all applications.

## 4.8. Complexity

While the overall complexity of the framework varies in each step, our methodology reduces the search space to keep the exploration tractable and fast. We establish bounds based on the number of total CI variants. Selection is the most critical step and could be exponential in the worst-case. Therefore, we try to always keep a reduced number of CI variants candidates, while maintaining energy and performance efficiency.

For each input application from the set of $B$ benchmarks we have a number of CIs $C$, and each CI is implemented as a variant $numVariants$ times. The total number of variants $CV$ processed to build MDs by Algorithm 1 is determined as:

$$CV = \sum_{i=1}^{B} \sum_{j=1}^{C_i} numVariants_j. \tag{4.11}$$

The complexity $K_{dis}$ of calculating distances between pairs of MDs (Section 4.6.1) is:

$$K_{dis} = O(CV \times (CV - C - 1)). \tag{4.12}$$

However, the key design decision here is to have a global MD, which obviates the need for a new MD to be computed to compare each pair of variants, speeding up the calculation. Finally, by performing the hierarchical clustering step explained in Section 4.6.2, and using a heuristic to limit the number of cluster versions per level, the final number of generated solutions that the selection of Section 4.7 processes is within the bounds of $O(CV)$. We thus retain the most promising CI candidates, in terms of area, performance and energy efficiency, while making sure the selection step's complexity does not explode exponentially.

## 4.9. Evaluation

### 4.9.1. Experimental Setup

We now describe the setup and experimental evaluation of our automated exploration framework MInGLE.

| Benchmark | Suite | Max. bits input | Max. bits output |
|---|---|---|---|
| cjpeg | MediaBench II | 2048 | 2048 |
| djpeg | MediaBench II | 1168 | 192 |
| gsmdec | MiBench | 176 | 160 |
| gsmenc | MiBench | 1312 | 128 |
| mpeg2enc | MediaBench II | 256 | 128 |
| optflow | OpenCV | 512 | 128 |
| rawcaudio | MiBench | 256 | 192 |
| rawdaudio | MiBench | 192 | 256 |
| susan | MiBench | 192 | 64 |
| tmndec | MediaBench II | 368 | 192 |
| tmnenc | MediaBench II | 2048 | 256 |

*Table 4.2: List of the evaluated applications and benchmarks suites, with the maximum size needed for the DSFU's input and output registers.*

We evaluate the framework with eleven applications from the media domain, listed in Table 4.2. The applications are extracted from the benchmark suites OpenCV [52], Mediabench II [53] and MiBench [55]. The two rightmost columns list the maximum bits needed for input and output data.

The target architecture is an in-order Intel Atom with a tightly-coupled DSFU, as described in Section 2.3.1 (Chapter 2). The DSFU accesses both the integer and SIMD/XMM register files with a latency that depends on the access type and operation. This specialized unit also has private registers: 16 128-bit for input data and 32 64-bit for output. We determine the size of the register files with the maximum size needed among the benchmarks in Table 4.2, which in this case is 2048 bits for both input and output data. Before starting any CI computation, data is moved into the input registers from the core's register files, and once the computation is completed the results are written back. Note that, for any CI, the extra cycles required to reading and writing its data are considered as part of the total latency for calculating speedup values.

In the *Candidates Extraction* step of the framework, we first identify hot regions of code with the LLVM profiler [44] and extract the CI functionalities in C code. We synthesize high-level CI descriptions with Vivado HLS

2013.3 [9] to obtain the circuit design cycles and area consumption. To be able to examine the area and speedup trade-offs illustrated in Table 4.1, we apply different unrolling factors to the CIs: none, 2, 4, and 8. The target FPGA is a Xilinx Virtex 7 (XC7VX690T) that runs at 400 MHz. DSFU power estimations are obtained with the Xilinx Power Estimator (XPE). We compile the target applications with LLVM-Clang with an unrolling factor of 8, automatic vectorization, and optimization $-O2$ as the baseline. Software cycles are measured with the Sniper simulator [56], with changes to accurately simulate an Intel Atom processor running at 1.6 GHz. Thus, the DSFU runs 4× slower than the baseline processor. Power measurements on Sniper are obtained with McPAT [57]. We run two different versions of the code on Sniper: the original application for baseline comparison, and the application with the code accelerated by the CIs marked in assembler for functional simulation. Unrolled, non-vectorized code sequences in the LLVM IR are analyzed in step *Canonicalization* to generate the polynomials for the Merging Diagrams, which are built with support of the symbolic algebra and calculus part of Sage [50]. In step *Merge CIs Generation*, we use the Fastcluster library [58] for hierarchical clustering, and fed cycles and power data into the models of Section 4.6.3 to obtain results. The interface for the CPLEX optimizer [59] in the *Selection* step is OpenOpt [60].

### 4.9.2. Results and Discussion

We now present experiments and results to assess how well our framework can identify CIs to be accelerated by a DSFU in hardware, measuring both speedup and improvement in EDP across various areas.

Figure 4.4 presents a comparison of different configurations of our framework, with DSFU area on the x-axis expressed as a percentage of the Virtex 7's area, and the average performance speedup across the domain on the y-axis. Figure 4.5 shows the same comparison, but this time with average EDP improvement on the y-axis. Dashed lines show improvements achieved when we use CIs targeting code within basic blocks. At the larger areas, performance improvement reaches a maximum of 1.48× and EDP improvement goes up to 1.67× the baseline. We compare this to the solid lines in the figures, which target code regions across basic blocks. In this case, speedup reaches a maximum of 1.98× and EDP improvement goes up to 3.35×. Considering regions with multiple basic blocks gives us a significant boost in both performance and energy efficiency, because we are able to accelerate 31% more statically counted body loops than with one basic block. Also, CIs across basic blocks cover 41% more dynamic instructions on average. Exploring CIs across basic blocks covers more code, expands

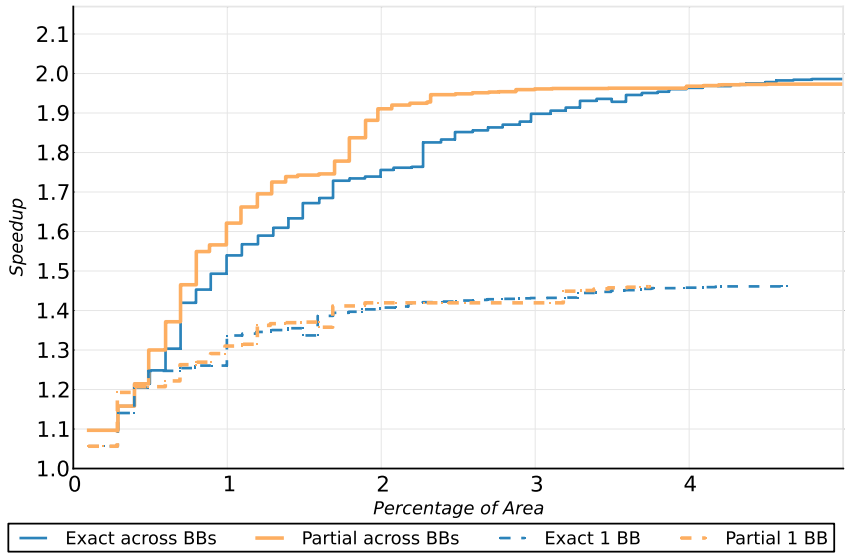


*Figure 4.4: Average speedup versus percentage of area occupancy of the DSFU for exact and partial matching methods, targeting one or many basic blocks.*

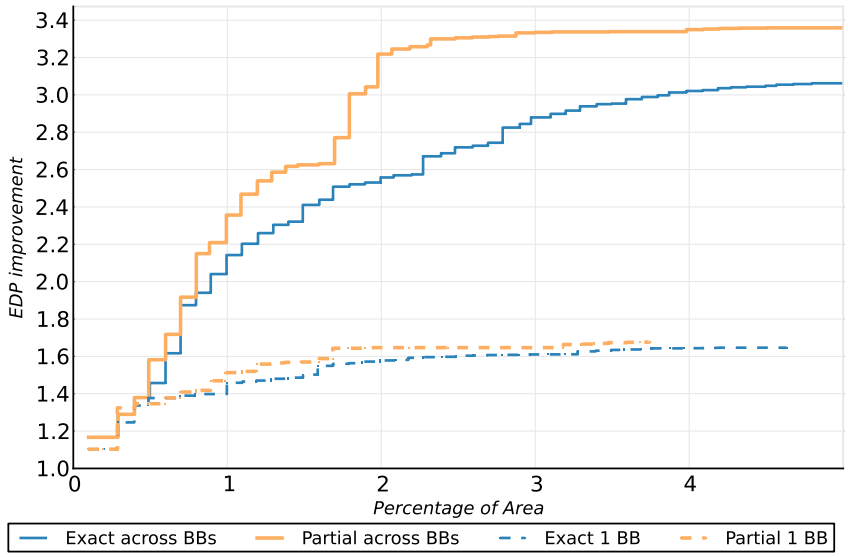


*Figure 4.5: Average EDP improvement versus percentage of area occupancy of the DSFU for exact and partial matching methods, targeting one or many basic blocks.*

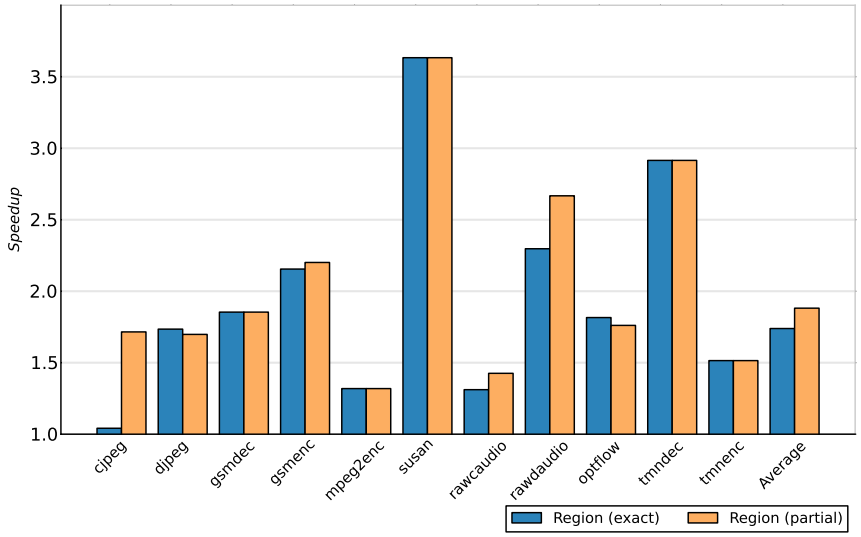


*Figure 4.6: Speedup for each benchmark at a limited implementation area (1.8%) across basic blocks.*

the acceleration opportunities, and thus achieves higher speedups.

In the same figures, we analyze the efficacy of exact versus partial matching by comparing blue and orange lines, respectively. Note that partial matching choices include all those CIs matched with exact, and then additional CIs that could be partially matched. We start seeing a difference around 0.5% of the area across basic blocks, noting that partial matching achieves larger speedups and EDP improvements as compared to exact matching, given the same area. For instance, with a limited area budget (1.8%), we observe a speedup of 1.88× and an EDP improvement of 3.04× when using partially matched CIs, while with exact matching we obtain a speedup of 1.73× and an EDP improvement of 2.53×. At 2.2% of the area, the EDP improvement difference is more noticeable, 2.57× against 3.25×. Alternatively, we see that for a given EDP improvement, partial matching saves area. For an EDP improvement of 3×, exact matching takes 4% of the area, whereas partial matching takes only 1.8% of the area: a savings of 55% of the chip's reconfigurable area. This is important as the area available for the reconfigurable DSFU in a low-end processor like the one evaluated would be much less than the area available in a Virtex 7.

Figures 4.6 and 4.7 show results for speedup and EDP improvement, respectively, for each benchmark at the limited area (1.8%) discussed above, comparing exact and partial matching across basic blocks. As our selection

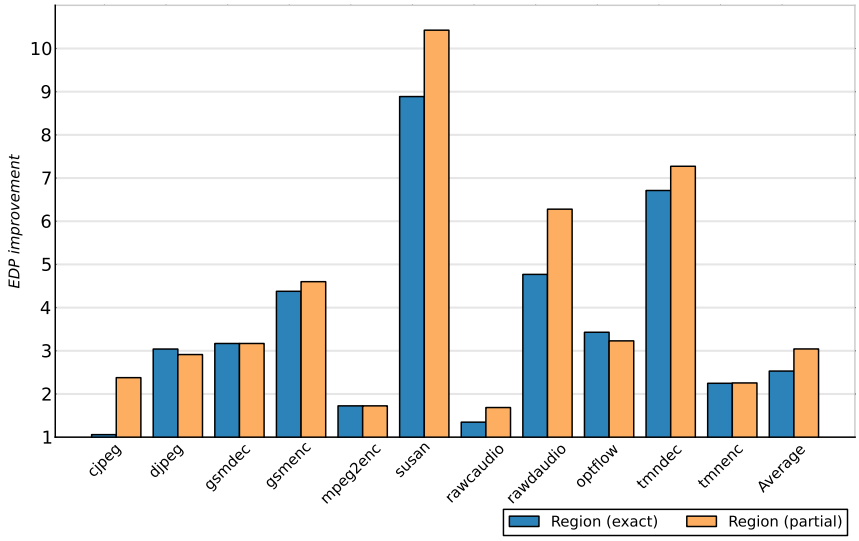


*Figure 4.7: EDP improvement for each benchmark at a limited implementation area (1.8%) across basic blocks.*

optimizes for EDP, we see larger EDP gains than speedup gains, when going from exact to partial matching. The speedup difference is moderate because of our selection objective; for instance, a power-hungry CI with high speedup but low energy efficiency will not be selected. Looking at the EDP of particular benchmarks, only two benchmarks marginally suffer a speedup and energy efficiency reduction: djpeg and optflow. However, most benchmarks have a significant improvement in their performance and EDP. For instance, the energy efficiency of cjpeg improves from 1.06× to 2.38×, for susan goes from 8.88× to 10.42×, and rawdaudio gets 4.76× with exact matching and 6.28× with partial ones. The average of all EDP improvements with partial matching is positive and therefore fair to all applications. Partial similarities contribute to area shrinking, which is key to energy efficiency. For example, with partial matching one of the selected CIs targets hot regions in seven different benchmarks, which results in an area reduction of 80% compared to exact matching.

## 4.10. Summary

This chapter presents a methodology and framework to automatically extract CIs to accelerate a domain of applications, ultimately selecting

those that achieve the highest performance improvements and energy efficiency when accelerated. To do so, our proposal explores the design space of tightly-integrated configurable functional units of limited size that accelerate applications across a domain. The presented MInGLE framework converts code sequences at the loop body level into CIs, considering several implementations for each of them. CIs are further transformed into a canonical representation, the Merging Diagram, which facilitates fast similarity detection. We then cluster CIs to be able to find partially-matching sequences to minimize specialized area. Our experimental results with 11 media benchmarks show that looking across basic blocks achieves a speedup of 1.98× and an EDP improvement of 3.35×, a significant gain over looking within a single basic block (speedup of 1.48× and EDP improvement of 1.67×). Across basic blocks, partial matching compared against exact matching is crucial for achieving larger performance (1.88× versus 1.73×) and EDP improvements (3.04× versus 2.53×) for a limited hardware area (1.8%). At very low areas, however, the gains are not noticeable. We explore new techniques that deal with that issue in the next chapter.

# 5 Fragments of Domain-Specific Instructions

## 5.1. Introduction

In the last two chapters, we explored the design of CIs that augment a general-purpose processor to accelerate an application domain. We found that domain-specific CIs deliver higher speedup than application-specific ones at realistic implementation areas. In order to expand the accelerating opportunities and to minimize specialized area, we identified partially-matching CIs across basic blocks, and integrated energy-efficiency in our methods. However, when the available area to implement the CIs is very small, domain-specialized CIs gains are limited.

We are interested on exploiting these low areas, since the implementation space for DSFUs has to be treated as an expensive commodity; it is not only reusability that drives domain-specific specialization, but also the optimal use of the available area. With a shrunk circuit we can either decide to add more functionalities, or to benefit from a low-power design.

Therefore, in this chapter, we extend previous techniques to expand CIs' usage and gains with minimal extra overhead. A new analysis step that detects fragments of CIs from which previously unseen acceleration opportunities are detected. The technique is critical to improve reutilization of hardware at the most limited areas, because we partially reuse an already

merged CI cluster, with minimum additional overhead.

We integrate the new CI fragment analysis within the automated framework MInGLE+, to strengthen the design of CIs across applications from a domain. Our techniques still tractably generate a non-exponential search space with merging and fragments generation. The framework selects CIs among exact-matched, partially-matched and matching with fragments that fit into a particular hardware area, maximizing energy efficiency and speedup across the applications. We compare the effectiveness of all matching configurations across and within basic blocks, and we evaluate different design parameters of the framework and we study the area usage for the exposed techniques.

The rest of this chapter is organized as follows: Section 5.2 explains the motivation behind the main contribution of this chapter. Section 5.3 introduces the framework in which the CIs are created. Section 5.4 presents a new matching technique with CI fragments, Section 5.5 explains the new distance calculation, and Section 5.6 includes the new selection strategy. For the evaluation, Section 5.7 presents results with several applications of the media domain, and Section 5.8 closes this chapter with the summary.

## 5.2. Motivation

Consider the clustering dendrogram of Figure 5.1 that organizes a hierarchy of CI similarities. Baseline CIs are located at level 0, while merged CIs start from level 1 and go upwards from more to less similarity degree. At each new level, two CIs from lower levels are merged. In the merging process, the distance (*dist*) between CIs is evaluated to determine which is the next merging pair of CIs. Each one of the merged CIs has an overhead from multiplexer switches represented as *MUX*, and a figurative saved cycles and new area placed in adjacent boxes.

Each one of the *N* baseline CIs is composed of one or several fragments, that we define as computation blocks that can be separated from the CI without incurring structural problems. Although those computation blocks can overlap, in this example, the internal CI operations covered by each fragment are fixed for illustrative purposes. Also, to simplify the example, we consider that each CI has only one variant. For instance, the only variant of *CI_1*, at level 0, is composed of fragments *F1_A* and *F1_B*. The partial merging explained in Chapter 4 merges whole CIs, based on increasing distance. With that method, we obtain first a new merged *CI_2+3* at level 1 and then *CI_1+2+3* at level 2. With each new merged CI, we obtain a speedup based on the combined saved cycles, and a new area that includes non-common operations, merged common computations, and the switching

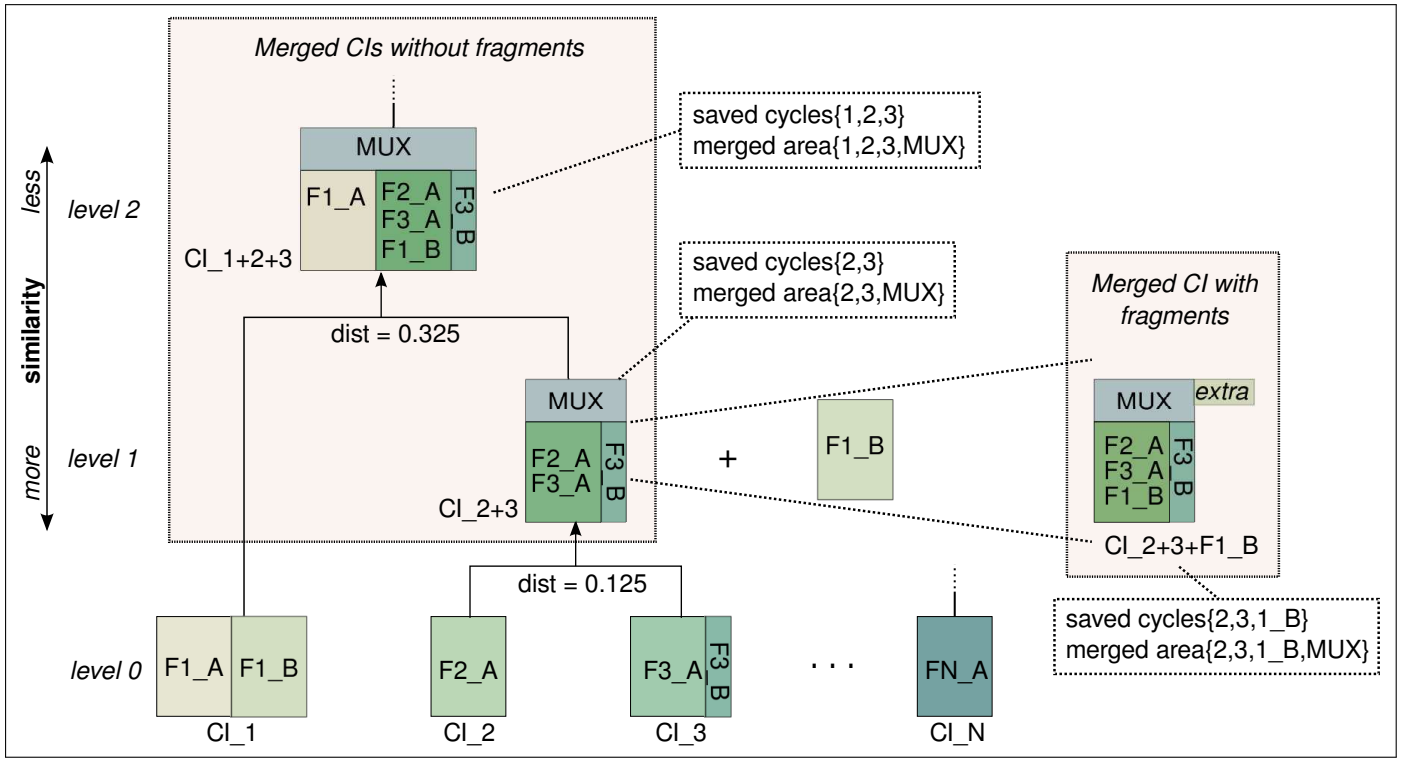


*Figure 5.1: Motivational example of partial merging without and with CI fragments.*

logic overhead (*MUX*).

However, consider *CI_1+2+3* and its fragments at level 2, product of merging *CI_1* with *CI_2+3*. Fragment *F1_B* from *CI_1* is completely merged with *CI_2+3*, avoiding a significant area increase. In contrast, fragment *F1_A*, also from *CI_1*, is fully incorporated at a substantial area increase. Consequently, we can argue that if we merge only one of the fragments, we could obtain savings in cycles at a low area cost. This is the case that *CI_2+3+F1_B* on the right of the figure illustrates (*Merged CI with fragments*). If we merge only fragment *F1_B* from *CI_1* with *CI_2+3*, the area increase from additional switching logic (*extra*) will be negligible, while performance will improve due to the additional saved cycles of computation block that *F1_B* accelerates.

## 5.3. MInGLE+ Automatic Framework

We adopt the same baseline processor as in Chapter 4: an in-order Intel Atom modified accordingly to Figure 2.4. CIs execute on a DSFU that reads and writes data from the processor's register files. Data transfers are therefore completely decoupled from CIs' execution, that is multi-cycle, with variable latency, and not parallel with the processor's functional units.

We follow the same naming conventions as in Chapter 4, thus a **CI** is the high-level representation of a loop body or inline function that can be accelerated in hardware, and **CI variants** or only **variants** are distinct implementations of a CI.

Figure 5.2 shows the high-level representation of MInGLE+ automated framework, composed of five steps. The framework follows a similar flow as described in Section 4.3. However, a new component is added after the *Merged CIs Generation*, which is the main contribution of this chapter. New Step 4, *CI Fragments Generation*, implements a new method to obtain larger improvements in performance and energy efficiency with small hardware areas available, which we explain in the following section.

To adapt the CIs to the fragment recognition step, we also modified the *Distance calculation*, step 3.1, that obtains the pairwise distances to measure similarities between CIs (Section 5.5), and the objective function of the relabeled Step 5, *CI Selection* (Section 5.6), to fit the best CI configuration that includes fragments in a limited area to save energy and improve performance across the domain.

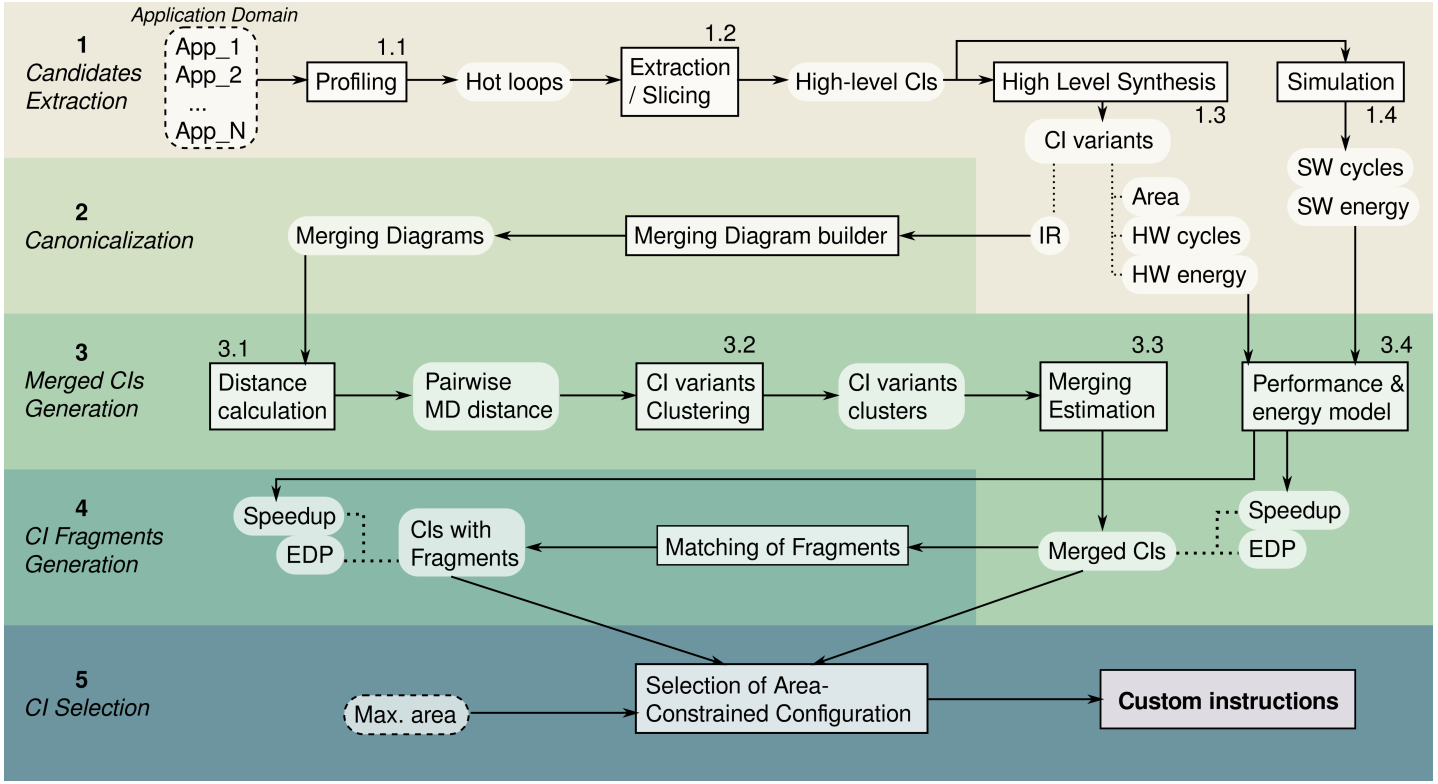


*Figure 5.2:* MInGLE+ *automated framework for the implementation and generation of partially-merged CIs with fragments.*

## 5.4. Generation of Custom Instruction Fragments

We call *CI fragments* a variation of partially matched CIs that will not include the full original CI, but parts (fragments) of it. This kind of matching is aimed to improve reutilization of hardware at the most limited areas. With CI fragments we can partially reuse an already merged CI cluster for CIs that were initially not included in that cluster, with minimum additional overhead. We obtain CI fragments in the *CI Fragments Generation* step of the MInGLE+ framework.

There are some conditions to specify how suitable CI fragments are found:

- The size of a CI fragment is at most the same as the CI that matches, which is generally much bigger. Therefore, for a given merged CI, we can have several fragments from different applications matching.
- Operations included in a fragment do not depend on excluded ones, to avoid a *convexity violation* [61], or circular dependency between operations that could result in wrong scheduling.
- CI fragments should not add logic to perform computations, but they can add some additional overhead for switching circuits. They also may have extra cycles to transfer data and the total number of saved cycles are probably less than if the full CI was included. All this additional overhead and reduced gains are carefully weighed to determine if a CI fragment is worth including.
- We can create CI fragments from the basis of CIs from any level of the dendrogram, either with exact or with partial similarities.
- Starting the fragment search from a basis CI variant, merged or not merged, that implements a set of CIs $C$, we will only consider adding fragments from variants not included in $C$.

Under those conditions, note that fragments of a given CI differ depending on the matching target, therefore their area coverage and saved cycles vary across solutions.

Algorithm 2 lists the pseudo-code that detects fruitful fragments to augment the initial set of solutions generated after the hierarchical clustering. We evaluate against each solution the possible matches of any CI variant, represented as an MD, that is not yet part of the solution. We start with the clustering solutions *MS*, that are the base to the new solution set plus fragments (*MSF*). For each MD evaluated, we obtain the fragment matches

**Algorithm 2**: Fragment Matching

**input** : Merged Diagrams $MDs$, merged clustering solutions $MS$, $threshold$
**output**: Solutions $MSF$

```
1  Array MSF ⟵ MS
2  for Sol ∈ MS do
3      Candidates ⟵ ∅
4      for md ∈ MDs do
5          if CI(md) ∉ Sol then
6              FM ⟵ ∅
7              for WholeFrag ∈ md do
8                  FM ←
                   FM ∪ GetMatchesOneWay(WholeFrag, Sol)
9              end
10             if matches(FM) > threshold then
11                 EDPImprov ← GetEDPImprovement(FM)
12                 Candidates ← Candidates ∪ < FM,
                   EDPImprov >
13             end
14         end
15     end
16     BestCandidates ← FilterCIVariant(Candidates)
17     NewSols ← CreateSolutions(BestCandidates)
18     MSF ← MSF ∪ NewSols
19 end
20 return MSF
```

(*FM*) evaluating separately the sequence of solutions that lead to each output (lines 6 – 9). Then, we can easily limit the fragmented matches to the boundaries of a certain output to control the convexity of the selected operations. With the function *GetMatchesOneWay* we perform a matching as explained in Section 5.5. In this case, we are only interested in knowing the coverage of each valid, whole fragment (*WholeFrag*) within *sol*. In lines 10 – 13, we evaluate if the percentage of matches of the fragments found reach a user-defined *threshold*. If they do, an estimation of the expected EDP improvement is calculated, and the fragments of that CI variant are considered to be included. As several variants of the same CI could be in the set of candidates, we filter them based on the best estimated EDP im-

provement in line 16. Finally, in the next line, we create a new solution structure with updated information over the area and the CI fragments that it includes, applying again the *Performance and Energy Model* step.

## 5.5. Distance and Matching Calculation

We modify the *Distance Calculation* step of *Merged CIs Generation* (3.1 in Figure 5.2), to measure the similarities among CIs and to use the calculation also for finding CI fragments.

For each pair of MDs of variants that do not implement the same CI, $CI_X$ and $CI_Y$, we compare them using the previously built global diagrams 4.5.2 to still compute distances fast. Thus, based on the pre-built global diagrams, we obtain the number of AugTED-operations and LinBDD-branches in $CI_X$ that do not match with those in $CI_Y$, namely $nM_X$, and vice versa, $nM_Y$.

However, looking at how fragments could match, we identify now three different types of matches with MDs: perfect, hidden and with overhead. An MD subdiagram $S$ with nodes $< v_1, \ldots, v_n >$ and edges $< e_1, \ldots, e_n >$ has a perfect match with another MD subdiagram $S'$ with nodes $< v'_1, \ldots, v'_n >$ and edges $< e'_1, \ldots, e'_n >$ if their labels and edges types match exactly. Afterwards, we identify a hidden match if the types of the outgoing edges of nodes $v_z$ and $v'_z$ match and are connected to subdiagrams with a perfect match. Finally, a match with overhead identifies only nodes that represent the same operations, but that do not share the same computational structure and would need a multiplexer to be shared. $Mo_X$ and $Mo_Y$ are then the number of nodes of $CI_X$ and $CI_Y$, respectively, with the same operations but with that extra overhead. As those matches with overhead incur in area costs, we count them also for the dissimilarity metric.

We also count the number of total AugTED and LinBDD nodes that each MD variant has: $Tot_X$ and $Tot_Y$. Then, we compute the distance $\delta$ as:

$$\delta(CI_X, CI_Y) = average\left(\frac{\frac{Mo_X}{2} + nM_X}{Tot_X}, \frac{\frac{Mo_Y}{2} + nM_Y)}{Tot_Y}\right). \tag{5.1}$$

We use the same methodology for the function *GetMatchesOneWay* of Algorithm 2. In this case, we define a base MD *Sol*, and the potential MD fragment $f$ and we obtain the matching information as described above. However, the distance value $\delta'$ will be in this case:

$$\delta'(Sol, f) = \frac{\frac{Mo_f}{2} + nM_f}{Tot_f}, \tag{5.2}$$

with $Mo_f$ matches of $f$ with overhead, $nM_f$ the non-matching subdiagram nodes, and $Tot_f$ the total nodes of $f$. Apart from the distance, the matching information is also passed along to the CI fragments generation.

## 5.6. Custom Instruction Selection with Fragments

In the last *CI Selection* step of MInGLE+, we choose again the best fitting CI configuration for a given hardware area. As explained in 4.7, we model this step as a Knapsack problem, trying to fit $n$ CI variants, merged and not merged, with fragments included and not included, within a limited hardware area $A$. Each $c_i$ candidate still has a value $v_i$, and a hardware occupancy $hw_i$.

Using MILP to solve the problem, the area constraint is:

$$\sum_{i=0}^{n} c_i \times hw_i \leq A. \tag{5.3}$$

The additional requirement of each CI being selected only once still holds, but expanding it with fragments. Now we not only have different variant implementations for each CI due to distinct unrolling factors or to merging, but we also include CIs with partially added fragments. Thus, we have to follow the same rule of invalidating a CI for further selection once a variant or only a fragment of that CI are selected. We do so by defining the constraint:

$$\sum_{i=0}^{m} vf_i \leq 1, \tag{5.4}$$

with $vf_i$ the high-level CI that can be implemented by ($m$) CI variants and variants with fragments.

The objective function, related to the overall energy-efficiency is:

$$\sum_{i=1}^{n} c_i \times \sigma\_EDP_i \rightarrow max. \tag{5.5}$$

The metric $\sigma\_EDP_i$ is computed as:

$$\sigma\_EDP_i = \sum_{j}^{B} \|\sigma\_EDP_{ij}\|, \tag{5.6}$$

where $B$ is the number of applications that the currently considered variant targets, and $\|\sigma\_EDP_{ij}\|$ is the original application $j$'s EDP minus the EDP

| Benchmark | #CIs | #variants | % dyn. ins. | Partial | Fragments |
|---|---|---|---|---|---|
| cjpeg | 4 | 16 | 81.6% | 461 | 2890 |
| djpeg | 3 | 12 | 45.3% | 434 | 1756 |
| gsmdec | 1 | 4 | 70.8% | 399 | 2281 |
| gsmenc | 2 | 7 | 56.5% | 406 | 1788 |
| mpeg2enc | 3 | 6 | 45.4% | 364 | 2084 |
| optflow | 2 | 7 | 49.5% | 440 | 2130 |
| rawcaudio | 1 | 4 | 87.0% | 402 | 2078 |
| rawdaudio | 1 | 4 | 85.2% | 410 | 2841 |
| susan | 1 | 4 | 95.4% | 427 | 2825 |
| tmndec | 3 | 4 | 87.2% | 401 | 2282 |
| tmnenc | 2 | 6 | 50.6% | 385 | 2632 |

*Table 5.1: For each application, number of CIs and CI variants considered, the percentage of dynamic instructions covered by them, and the number of candidates found with partial matching and matching with fragments for regions across basic blocks.*

with the variant, normalized to the observed maximum for that application. In this new definition of $\sigma\_EDP_i$, we have simplified the equation with respect to that in Section 4.7, eliminating the part that involved area savings. If the area is involved, partially-matched fragments are prioritized because the area portion is larger, and those with fragments, with less area impact, are not selected. Thus, with the new metric we aim to select above all CIs with fragments when it is possible to maximize area savings with a low overhead.

## 5.7. Evaluation

### 5.7.1. Experimental Setup

The setup information of MInGLE+ is the same as in Chapter 4. We refer to Section 4.9.1 of that chapter for details about the tools and platforms used.

We evaluate the framework the eleven applications from the media domain listed in Table 5.1. For each one of the benchmarks in the first column, we show in the second column the number of critical CIs found across basic blocks. The third column lists the number of CI variants or distinct implementations, for several unrolling factors; only those implementations that yield some performance improvement are considered. The fourth column shows the percentage of dynamic instructions covered if all the CIs were selected, with all of them over 45%. Such a large code coverage is key for performance improvement, and better achieved with CIs that cover regions across basic blocks. Benchmark cjpeg has the highest number of CIs and variants; however, the highest coverage of dynamic instructions corresponds to susan. The two rightmost columns list the number of merged CIs generated with partial matching and matching with fragments, respectively. Note that both numbers include exact matchings, and partial matching is a subset of matching with fragments. The threshold of similarity matching with fragments is set at 50%, as we discuss in more detail in Section 5.7.2.2.

## 5.7.2. Results

We first compare different techniques implemented in the framework to identify CIs across and inside basic blocks to be accelerated by a DSFU in hardware, measuring both speedup and improvement in EDP across various area settings. We subsequently evaluate the effect of different threshold values on fragment matching. Finally, we present results of the shared hardware area characterization when we use different matching techniques.

### 5.7.2.1. Speedup and EDP Improvement

Figures 5.3 and 5.4 presents a comparison of different configurations that the framework generates for the benchmarks in Table 5.1, with DSFU area on the x-axis expressed as a percentage of the Virtex 7's area. Figure 5.3 shows the average performance speedup and Figure 5.4 the average EDP improvement across the domain on the y-axis. Speedup and EDP improvement are calculated with respect to the baseline processor. Lines marked with *1 BB* show improvements achieved when we use CIs targeting code within basic blocks. At the largest areas, performance improvement reaches a maximum of 1.48× and EDP improvement goes up to 1.74× the baseline. We compare this to the lines marked with *Region* in the figures, which target code regions across basic blocks. In this case, speedup reaches a maximum of 2.09× and EDP improvement goes up to 3.84×. Considering regions with multiple basic blocks gives us a significant boost in both performance and energy efficiency, because we are able to accelerate 31% more statically

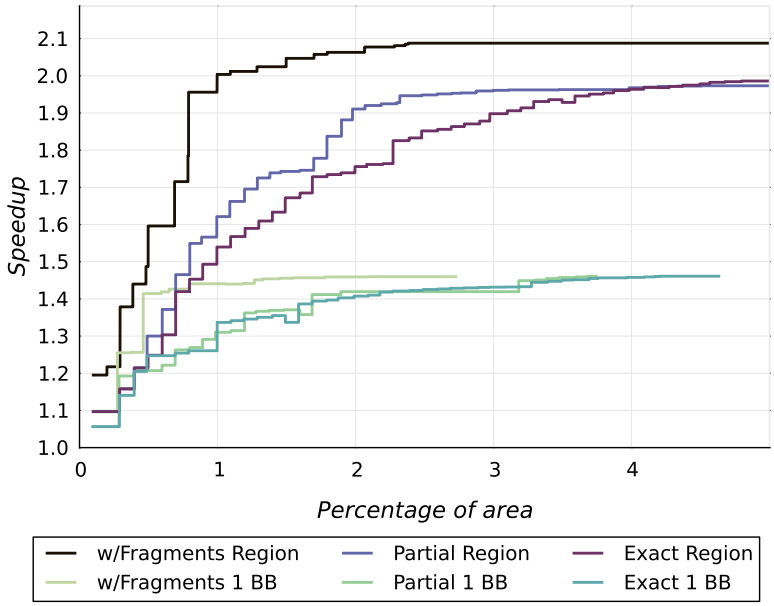


*Figure 5.3: Average speedups (y-axis), against increasing area percentages (x-axis), for exact and partial matching, and matching with fragments, across and within the basic block level.*

counted body loops than with one basic block. Also, CIs across basic blocks cover 41% more dynamic instructions on average. Exploring CIs across basic blocks covers more code, expands the acceleration opportunities, and thus achieves higher energy efficiency and speedup.

In the same figures, we analyze the efficacy of exact matching, partial matching and matching with fragments by comparing those lines marked as *Region*. Note that partial matching choices include all those CIs matched with exact, and then additional CIs that could be partially matched. The same case applies for matching with fragments, with partial matching choices included among newly generated ones. In the case of partial matching, we start seeing a difference around 0.5% of the area across basic blocks, noting that partial matching achieves larger speedups and EDP improvements as compared to exact matching, given the same area. For instance, with a limited area budget (1.8%), we observe a speedup of 1.88× and an EDP improvement of 3.04× when using partially matched CIs, while with exact matching we obtain a speedup of 1.73× and an EDP improvement of 2.53×. At 2.2% of the area, the EDP improvement difference is more noticeable, 2.57× against 3.25×. Alternatively, for a given EDP improve-

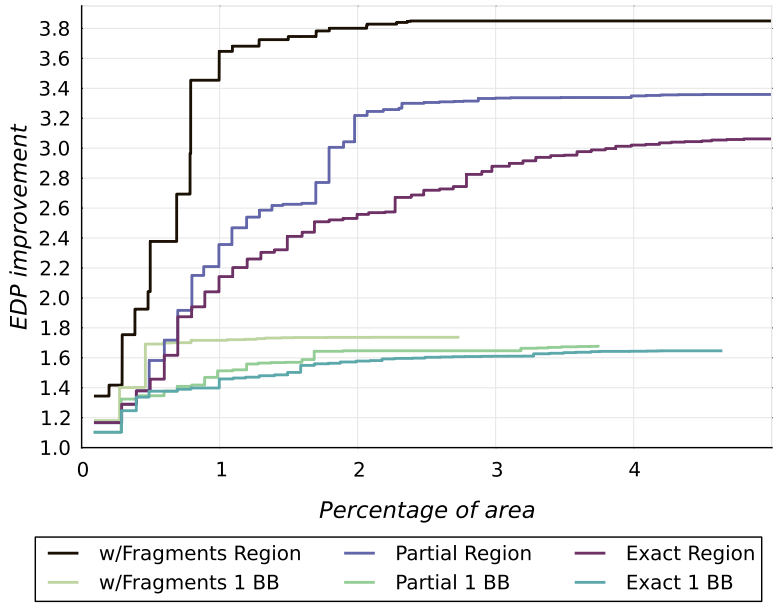


*Figure 5.4: Average EDP improvements (y-axis), against increasing area percentages (x-axis), for exact and partial matching, and matching with fragments, across and within the basic block level.*

ment, partial matching saves area. For instance, for an EDP improvement of 3×, exact matching takes 4% of the area, whereas partial matching takes only 1.8% of the area: a savings of 55% of the chip's reconfigurable area. Matching with fragments, though, outperforms previous techniques from the beginning, at very limited areas. With only 1% of the Virtex 7, we have a speedup of 2× and EDP improvement of 3.65×, clearly higher than the same values for partial matching, 1.63× and 2.35×, respectively. Matching with fragments for CIs across basic blocks helps to reach the best speedup and energy efficiency at larger areas. However, the most important feature of matching with fragments is to enable high performance at smaller areas either within or across basic blocks. Hence, matching with fragments uses area more effectively; a speedup of 1.96× is achieved with fragments at 0.75% of the area, in contrast with the 2.5% needed with partial matching. This is important as the area available for the reconfigurable DSFU in a low-end processor like the one evaluated would be much less than the area available in a Virtex 7. As a rule of thumb, an Atom implementation took about 85% of the Virtex 5 LX330 that has roughly 25% of the capacity of the Virtex 7.

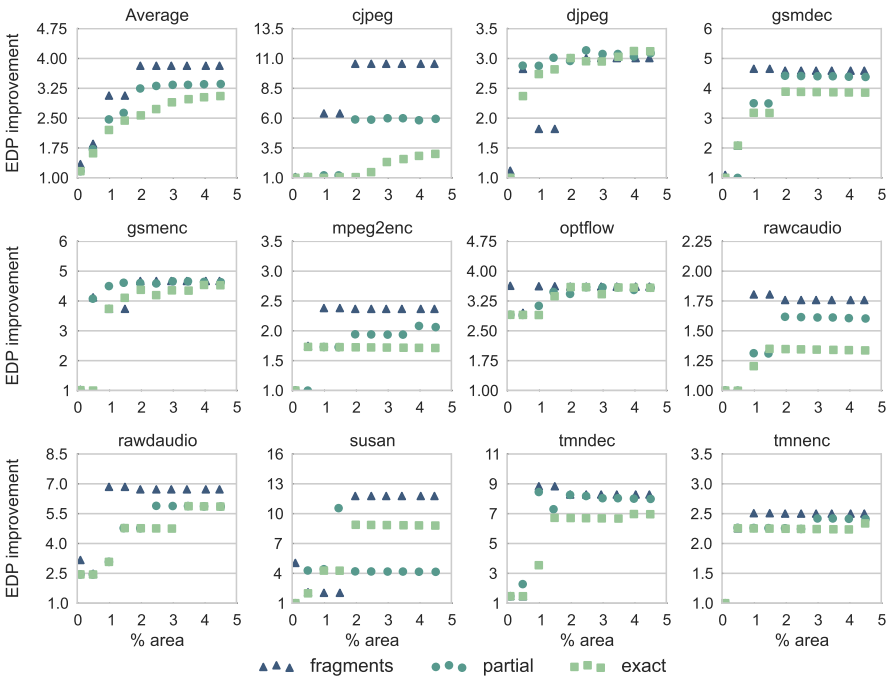


*Figure 5.5: EDP improvement for each benchmark, up to the 5% of the area, with CIs selected across basic blocks with fragments, partial matching and exact matching.*

Figure 5.5 presents a graph for each benchmark with a range of area percentages dedicated to the CIs on the x-axis, and EDP improvement on the y-axis. Here, we only include CIs across basic blocks. Results of the matching with fragments use a threshold of 50%, which we discuss in detail in the next section. Each point on the graphs represents a group of selected CIs that uses a particular area. Only some area values are displayed, with a stride of 0.5%. Note that each benchmark has a different y-axis scale for readability. The average of all applications is shown in the top left graph.

As we pointed out before, matching with fragments is, on average, the most effective technique at finding domain-specific CIs. This technique achieves higher EDP improvement at smaller areas, always increasing the speedup faster than the other two techniques. All but three benchmarks show the best efficiency with fragments regardless of the area. We can observe, though, that for djpeg, gsmenc and susan, between 0.5% and 1.5%

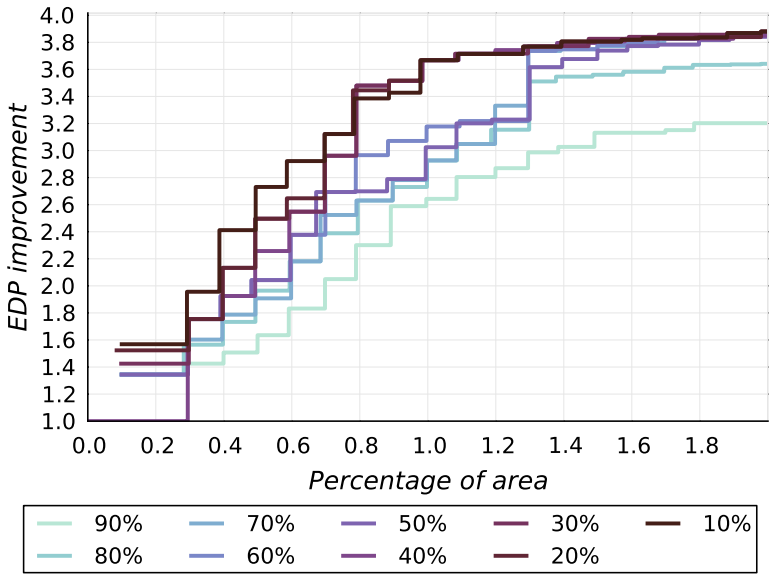


*Figure 5.6: Percentage of area (x-axis) versus average EDP improvement (y-axis) for the matching with fragments for different thresholds.*

of the area, solutions with fragments yield lower efficiency than with partial matching. In the case of djpeg, even at higher areas, the EDP improvement of the three methods overlaps. This is due to a great dependency of the benchmark on application-specific CIs, with very low sharing rates in all the CIs generated. Regarding gsmenc and susan, although the selected fragments at low area improve the EDP, they cannot reach the gains of CIs that cover the full body loop, and not only parts of it. However, for the other eight benchmarks, matching with fragments is clearly the best choice, since we are able to cover more CIs with less area. For instance, CIs that could give more than 10× EDP improvement to cjpeg are not selected with partial matching because of unavailable area resources. With fragments, there is virtually more area available from the low overhead costs of including a new fragment, hence better performing CI variants can be selected.

#### 5.7.2.2. Threshold Analysis

We recall the user-defined threshold for fragment matching from Section 5.4 as the value that establishes the minimum percentage of matching operations of a fragment with respect to the evaluated CI, in order to generate a new CI that includes both the evaluated CI and the fragment. Figure

| *T* | Candidates | | Time to solve | | EDP improvement |
|---|---|---|---|---|---|
| | *num.* | *% inc.* | *secs.* | *% inc.* | |
| 90 | 363 | – | 24.4 | – | 2.64× |
| 80 | 390 | +7.4 | 26.5 | +8.6 | 2.94× |
| 70 | 456 | +16.9 | 27.4 | +3.4 | 2.94× |
| 60 | 581 | +27.4 | 33.7 | +22.9 | 3.19× |
| 50 | 633 | +8.9 | 35.9 | +6.5 | 3.64× |
| 40 | 923 | +45.8 | 52.4 | +45.9 | 3.67× |
| 30 | 1056 | +14.4 | 59.3 | +13.2 | 3.67× |
| 20 | 2117 | +100.4 | 125.5 | +111.6 | 3.67× |
| 10 | 2263 | +6.9 | 135.3 | +7.8 | 3.67× |

*Table 5.2: Number of candidates in the selection step and time to solve the selection problem for different thresholds (T) using matching with fragments, for* 1% *of the area.*

5.6 presents a comparison of solutions with different threshold values, with area percentage on the x-axis up to 2% and the average EDP improvement across the domain on the y-axis. The legend shows the thresholds that go from 90% to 10% of matching. A higher threshold corresponds to a higher similarity. The CI candidates with a given threshold include all those CIs from higher thresholds. For instance, a threshold of 70% also includes the CIs of thresholds 80% and 90%. We observe that up to 0.8% of the area, 10% threshold obtains the highest EDP improvement. However, from that area onwards, thresholds up to 50% yield the same EDP improvement and, from 1.3% of the area, thresholds 60% and 70% join the efficiency ceiling. The EDP improvement with a threshold of 90% at 2% of the area equals the one achieved with partial matching (no fragments). At larger areas, we have room to choose bigger variants that provide the full CI acceleration instead of fragments that do not give the maximum efficiency. Also, fragments with 90% similarity matching are more difficult to find than those with lower thresholds and therefore scarcer.

The threshold level has an immediate effect on the number of CI candi-

dates in the selection pool and the runtime of the selection process, which is shown in Table 5.2. Data in the table refer to the selection step for 1% of the area. For each threshold percentage (T), we list first the number of candidates considered for selection with the percentage increase with respect to the previous row. We list also the time in seconds to solve the selection problem with the pool of candidates and, again, percentage increases. In the last column we list the EDP improvement achieved. Note that, for different areas, the number of CI candidates varies because some CIs are pre-filtered by area occupancy; if their area is greater than the maximum area targeted, they will not be considered. Also note that, as the number of candidates of a given threshold includes those of higher ones, the amount of candidates increases as the threshold value decreases. The time to solve increases linearly with the number of candidates; the largest difference in both the amount of CIs considered and seconds to solve happens relaxing the threshold from 30% to 20%, showing that smaller fragments are more frequent than larger ones. However, the EDP at those low thresholds is not better than thresholds of $40 - 50\%$ because larger fragments achieve better EDP, and the threshold is related to the size of the fragment. Thus, the increase in the problem complexity of the lowest thresholds weighed against the problem size and time to solve of a threshold of the 50% has no advantage because similar CIs are chosen.

#### 5.7.2.3. Sharing Characterization

In this last analysis, we evaluate how area is shared among CIs to understand the high gains that the matching with fragments provides. Figures 5.7 and 5.8 show two graphs that display the increasing percentage of the Virtex 7 occupancy (up to 5%) of different CI configurations on the x-axis versus the normalized percentage of LUTs on the y-axis, broken down by non-shared operators, shared operators and multiplexer overhead for circuit sharing. The CI configurations target all the applications in Table 5.1. The graph on Figure 5.7 shows the characterization for partial matching, whereas the graph on Figure 5.8 shows that of matching with fragments.

First, we observe that, for partial matching, the CI configuration with the smallest area (0.2%) does not share any of the CIs included. At that small area, the cost for merging CI variants is too high to compete against lighter application-specific CIs. In contrast, matching with fragments devotes 80% of the LUTs on the smallest configuration to shared resources. We find the maximum percentage of shared circuits at smallest areas, which explains why the configurations with CI fragments are more efficient than those without. The percentage of overhead due to multiplexers is more no-

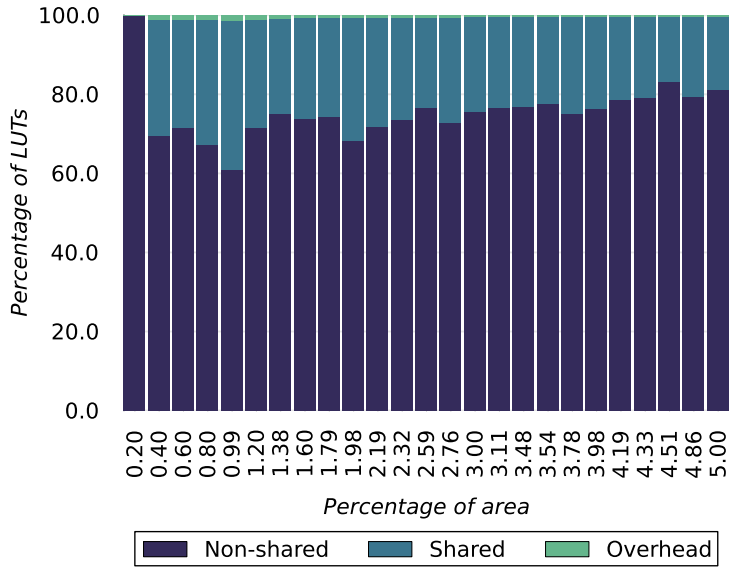


*Figure 5.7: Characterization of shared FPGA hardware for different area utilizations with partial matching.*

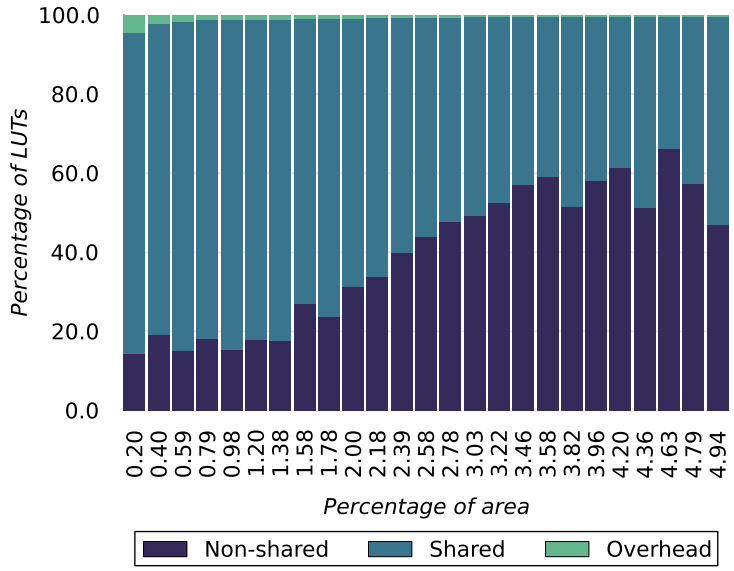


*Figure 5.8: Characterization of shared FPGA hardware for different area utilizations with fragments.*

ticeable at lower areas also, correlated with the amount of shared resources. Although at larger area utilizations the sharing levels decrease, they are steadily higher than those for partial matching, where the sharing percentages are around the 30% on average.

## 5.8. Summary

This chapter presents MInGLE+, an automated framework that extracts CIs from a domain of applications that are executed on a DSFU. We aim to select CIs that improve performance and energy efficiency for all the target applications, without substantially compromising any of them. Hence, we propose techniques to perform partial matching of CIs based on their similarity. Our techniques here build on top of the work of the previous chapter, now focusing on getting high benefits for DSFU with very limited areas. Therefore, we introduce an analysis step that detects fragments of CIs that can use the existing merged clusters with minimal extra overhead. Our experimental results with eleven media benchmarks show that the new matching technique with fragments achieves a speedup of 2.1× and an EDP improvement of 3.8×, on average, across basic blocks, while within a basic block we obtain a speedup of 1.5× and EDP improvement of 1.7×. Compared to partially matched CIs, CIs with fragments are key for achieving larger performance (2× versus 1.6×) and EDP improvements (3.6× versus 2.4×) for a limited hardware area (1%). This means that we achieve a particular energy efficiency with a greatly reduced hardware area. The work presented in this chapter, complemented with the analysis and techniques from Chapters 3 and 4, shows the applicability of introducing configurable accelerators with limited area inside embedded processors. We demonstrate the viability of accelerating many applications from a domain, improving the system's performance and energy efficiency.

# 6
# Conclusion

## 6.1. Overview

Hardware specialization has gained attention over the past few years in an effort to improve performance and energy-efficiency at the end of Dennard scaling. However, an application-specific processor delivers high performance for that specific application only, and is costly to manufacture. In contrast, domain-specific acceleration may deliver better overall system performance when different applications run on the custom device, and may be more economically viable by targeting a larger market. In addition, the design of custom instructions (CIs) enables a fast way of creating specialized hardware from the extension of a general-purpose processor. CIs run on specialized functional units, which are an area-efficient alternative of hardware specialization, extending a processor with minimal impact on the existing design. The memory hierarchy is maintained, and although the amount of data that can be processed at once is limited, hardware extensions are more efficiently managed, and energy consumption can be better controlled to not be increased over the original baseline.

Although domain-specific CIs stand as a plausible option for acceleration design, the differences among coding styles across applications complicate the task of finding patterns that can be executed by a single CI. Therefore, we propose using a canonical representation, the Taylor Expansion Diagram (TED), to identify CIs that are functionally similar. Com-

pared to the commonly used Directed Acyclic Graph (DAG) representation, TEDs are more effective at identifying those functional equivalences along sequences of code within and across applications. We also propose a Hybrid representation, which uses TEDs when it is possible by the nature of the computation, and otherwise uses DAG.

Additionally, we propose four scoring heuristics to quickly compare and rank CIs in the selection phase, which is known to be an NP-complete problem. These heuristics rank potential configurations of groups of CIs to smooth the gains obtained across applications, making them suitable for domain-specific acceleration.

We combine the functional equivalence identification and the scoring heuristics in our framework FuSInG, which allows a design space exploration of new acceleration designs through performance and area estimation. With the framework, we evaluate the proposed methodologies using a set of application benchmarks from the media domain. We first explore the effectiveness of a canonical representation for the design of domain-specific CIs. While TED's canonical representation does not lead to better results than DAG with code sequences within the same application, it is key to achieve higher speedups when generating domain-specific CIs. As individual applications are coded following the same style, the benefit of a canonical representation is unnoticeable, as opposed to CIs across applications with different code styles. In the domain-specific case, we find that Hybrid and TED techniques perform very similarly. Hybrid is, though, the most effective technique, achieving higher speedups at smaller areas and increasing the speedup faster than only TED or DAG. The main advantage of the Hybrid technique is that it tries to identify identical CIs using TED first, and when it cannot find any more, it complements the identification using DAG. Furthermore, we evaluate the behavior of the heuristic selection techniques. We observe that, from all four, the random-scaled sharing technique performs best on average, since it maximizes the overall performance across applications. Also, we explore the trade-offs of different SFU configurations to optimize full-system performance across applications subject to area constraints. We compare configurations with application-specific only, domain-specific only, or both application- and domain-specific CIs. We find that, while application-specific CIs result in the highest possible performance at large or unbounded core areas, considering domain-specific CIs next to application-specific CIs yields to the highest possible speedup at realistic, smaller core areas. In addition, we cross-validate the results looking at the performance of non-analyzed applications that run upon a machine configured with a set of CIs for a particular application domain. We demonstrate that the identified CIs are effective for previously unseen

applications within the same domain, making specialization more generally applicable. These findings underline the need of domain-specific instructions for practical and flexible hardware specialization.

In order to achieve high and balanced speedups with domain-specific CIs, reusability across applications is a critical factor. However, targeting code sections beyond the basic block level to achieve higher speedups results in not enough exact equivalences to generalize the accelerating hardware. Therefore, to improve the code coverage, we propose a new canonical representation of code sequences across basic blocks, the Merging Diagram. This representation is partly based on TED-DAG Hybrid, and its structure facilitates the identification of partial similarities. We also introduce a clustering-based partial matching of CIs, that we merge, calculating their potential performance and energy improvement.

Also, having confirmed the suitability of the CI selection in an application domain, we introduce energy efficiency as a new parameter to take into account, since in the context of devices with limited power-budgets, focusing only on performance could introduce power-hungry CIs. We propose then a constraint-based selection mechanism that, with a novel objective function, balances speedup and energy efficiency across an application domain.

These techniques expand the opportunity for CIs with a limited area budget inside embedded processors to accelerate several applications from a domain, improving the system's energy efficiency. We evaluate our claims with MInGLE, an automated framework that converts code sequences at the loop body level into CIs, considering several implementations for each of them. CIs are further transformed into Merging Diagrams, and clustered to find partially-matching sequences to minimize specialized area.

Experimental results with a set of media benchmarks show that looking across basic blocks achieves a speedup of 1.98× and an EDP improvement of 3.35×, a significant gain over looking within a single basic block (speedup of 1.48× and EDP improvement of 1.67×). Improvements in both performance and EDP across basic blocks are due to be able to cover 41% more dynamic instructions on average, which expands the acceleration opportunities. Furthermore, across basic blocks, partial matching achieves larger speedups and EDP improvements as compared to exact matching, given the same area. For example, for a limited hardware area (1.8%), partial matching achieve larger performance than exact matching (1.88× versus 1.73×) and EDP improvements (3.04× versus 2.53×). Alternatively, for a given EDP improvement, partial matching saves area, which is important as the area available to implement the CIs in a low-power processor is very limited.

Previous results show that partially-similar domain-specific CIs outperform application-specific ones when the area for implementation is over a given threshold. However, at small areas, application-specific CIs are still in dominance, since the overhead added when two CIs are merged override the potential gains. Consequently, we extend the CI merging of MInGLE with an analysis step that detects fragments of CIs that can use the existing merged clusters with minimal extra overhead. With CI fragments we can improve reutilization of hardware at the most limited areas, because we partially reuse an already merged CI cluster. Experimental results show that the new matching technique with fragments achieves a speedup of 2.1× and an EDP improvement of 3.8×, on average, across basic blocks, while within a basic block we obtain a speedup of 1.5× and EDP improvement of 1.7×. Comparing matching techniques across basic blocks, matching with fragments outperforms previously proposed techniques from the beginning, at very limited areas. With only 1% of the area, we have a speedup of 2× and EDP improvement of 3.65×, clearly higher than the same values for partial matching, 1.63× and 2.35×, respectively. Matching with fragments uses area more effectively; a speedup of 1.96× is achieved with fragments at 0.75% of the area, in contrast with the 2.5% needed with partial matching. As there is virtually more area available from the low overhead costs of including a new fragment, better performing CIs can be selected.

The results presented in this dissertation show the applicability of introducing configurable accelerators with limited area inside low-power processors to accelerate many applications from a domain, improving the system's performance and energy efficiency.

## 6.2. Future work

The techniques for domain-specific acceleration proposed in this dissertation could be extended and adapted to new environments.

In order to improve performance, we could broaden the coverage of CIs beyond the loop body level. For example, the analysis phase might try to identify a coarser type of CIs than body loops, possibly at the function level. However, finding similarities across larger sequences of code might pose a problem, once again due to the disparities of coding strategies across applications. That said, a new canonical representation at the algorithmic level, conscious of data structures, and acting as a functional intermediate representation, could help with the similarity identification. With a beyond-loop-level type of CI, memory transfers should be included, which might imply architectural changes to allow more direct memory connections. To improve the similarity matching, we could also consider approximate com-

puting approaches, such as relaxing type constraints to be able to generalize more CIs across applications.

To extend the applicability of the techniques presented in this dissertation, instead of an off-line analysis and generation of CIs, we could do both the identification of CIs and configuration of SFUs in runtime. Runtime configuration would need architectural support to raise an exception and stop the program, flushing and updating the SFU. Identification at runtime is a more interesting and difficult problem. We could approach it, for instance, by establishing an initial baseline of CIs that could evolve and change depending on real workload dynamics.

Additionally, we could extrapolate the domain-specific design methods to other, not yet explored, architectures. For instance, the design and implementation of non-tightly-coupled accelerator types would be of interest, because off-core accelerators would allow better performance gains. This direction would be related to the expansion of CIs beyond the loop body, since full functions would be offloaded on accelerators to minimize communication with the main processor. However, without changing the granularity of the CIs, we could start extending the current work for out-of-order processors, which in the latest years are growing as the default choice in embedded devices. Also, we could evaluate our techniques in other high-performance environments, like multiprocessors. Using an Intel Atom prototype with 2-way SMT, we can have up to 4 simultaneous threads with a dual-core machine. We could study the architectural possibilities on such a platform, and the possible impact of introducing CIs in the system. For example, in a scenario of two applications running concurrently in the same chip, we could design exact duplicates of the DSFUs as it is done in Atom for rest of the FUs. Alternatively, we could design different kinds of DSFUs in each processor, and a scheduler would choose in which one of them a process that uses CIs would run.

# Bibliography


[1] G. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, pp. 114–117, Apr. 1965.

[2] R. H. Dennard, F. H. Gaensslen, H. Yu, V. L. Rideout, E. Bassous, and A. R. LeBlanc, "Design of ion-implanted MOSFET's with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, pp. 256–268, Oct. 1974.

[3] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Proceedings of the 38th Annual International Symposium on Computer architecture*, ISCA '11, pp. 365–376, ACM, 2011.

[4] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz, "Understanding sources of inefficiency in general-purpose chips," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pp. 37–47, ACM, 2010.

[5] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor, "Conservation cores: Reducing the energy of mature computations," in *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pp. 205–218, ACM, 2010.

[6] W. Arden, M. Brillouët, P. Cogez, M. Graef, B. Huizing, and R. Mahnkopf, "More-than-Moore white paper," *International Technical Roadmap for Semiconductors*, 2010.

[7] G. Estrin, "Organization of computer systems: The fixed plus variable structure computer," in *Papers Presented at the May 3-5, 1960, Western Joint IRE-AIEE-ACM Computer Conference*, IRE-AIEE-ACM '60 (Western), pp. 33–40, ACM, 1960.

[8] K. Keutzer, S. Malik, and A. R. Newton, "From ASIC to ASIP: The next design discontinuity," in *Computer Design: VLSI in Computers and Processors*, pp. 84–90, IEEE, 2002.

[9] Xilinx, "Accelerating integration – Vivado High-Level Synthesis." `http://www.xilinx.com/products/design-tools/vivado/integration.html`, 2015.

[10] R. Kastner, a. Kaplan, S. O. Memik, and E. Bozorgzadeh, "Instruction generation for hybrid reconfigurable systems," *ACM Transactions on Design Automation of Electronic Systems*, vol. 7, pp. 605–627, Oct. 2002.

[11] B. Middha, A. Kumar, V. Raj, M. Balakrishnan, P. Ienne, and A. Gangwar, "A Trimaran Based Framework for Exploring the Design Space of VLIW ASIPs with Coarse Grain Functional Units," in *In Proceedings of the 15th International Symposium on System Synthesis*, pp. 2–7, 2002.

[12] P. Yu and T. Mitra, "Scalable custom instructions identification for instruction-set extensible processors," in *Proceedings of the 2004 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '04, pp. 69–78, ACM, 2004.

[13] P. Yu and T. Mitra, "Disjoint Pattern Enumeration for Custom Instructions Identification," *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, pp. 273 – 278, 2007.

[14] J. Cong, Y. Fan, G. Han, and Z. Zhang, "Application-specific instruction generation for configurable processor architectures," in *Proceedings of the 2004 ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays*, FPGA '04, pp. 183–189, ACM, 2004.

[15] L. Pozzi and P. Ienne, "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*, CASES '05, pp. 2–10, ACM, 2005.

[16] L. Pozzi, K. Atasu, and P. Ienne, "Exact and approximate algorithms for the extension of embedded processor instruction sets," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, pp. 1209–1229, July 2006.

[17] A. K. Verma, P. Brisk, and P. Ienne, "Rethinking custom ise identification: a new processor-agnostic method," in *Proceedings of the 2007 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '07, pp. 125–134, ACM, 2007.

[18] P. Biswas, N. Dutt, P. Ienne, and L. Pozzi, "Automatic identification of application-specific functional units with architecturally visible storage," *Design, Automation, and Test in Europe*, 2006.

[19] N. T. Clark, H. Zhong, and S. A. Mahlke, "Automated custom instruction generation for domain-specific processor acceleration," *IEEE Transactions on Computers*, vol. 54, p. 2005, 2005.

[20] K. Atasu, O. Mencer, W. Luk, C. Ozturan, and G. Dundar, "Fast custom instruction identification by convex subgraph enumeration," in *Proceedings of the 2008 International Conference on Application-Specific Systems, Architectures and Processors*, ASAP '08, pp. 1–6, IEEE Computer Society, 2008.

[21] T. Li, Z. Sun, W. Jigang, and X. Lu, "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*, CASES '09, pp. 29–36, ACM, 2009.

[22] A. C. Murray, R. V. Bennett, B. Franke, and N. Topham, "Code transformation and instruction set extension," *ACM Transactions on Embedded Computing Systems*, vol. 8, pp. 1–31, July 2009.

[23] K. Atasu, W. Luk, O. Mencer, C. Ozturan, and G. Dundar, "FISH: Fast Instruction SyntHesis for Custom Processors," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 20, no. 99, pp. 1–1, 2012.

[24] K. Martin, C. Wolinski, K. Kuchcinski, A. Floch, and F. Charot, "Constraint Programming Approach to Reconfigurable Processor Extension Generation and Application Compilation," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 5, pp. 1–38, June 2012.

[25] N. Arora, K. Chandramohan, N. Pothineni, and A. Kumar, "Instruction selection in ASIP synthesis using functional matching," *International Conference on VLSI Design*, vol. 0, pp. 146–151, 2010.

[26] a. Lodi, M. Toma, F. Campi, a. Cappelli, R. Canegallo, and R. Guerrieri, "A VLIW processor with reconfigurable instruction set for embedded applications," *IEEE Journal of Solid-State Circuits*, vol. 38, pp. 1876–1886, Nov. 2003.

[27] J. Hauser and J. Wawrzynek, "Garp: a MIPS processor with a reconfigurable coprocessor," in *Proceedings of the 5th Annual IEEE Symposium*

*on Field-Programmable Custom Computing Machines)*, pp. 12–21, IEEE Computer Society, 1997.

[28] J. E. Carrillo and P. Chow, "The effect of reconfigurable units in superscalar processors," in *Proceedings of the 2001 ACM/SIGDA Ninth International Symposium on Field Programmable Gate Arrays*, FPGA '01, pp. 141–150, ACM, 2001.

[29] R. Razdan and M. D. Smith, "A high-performance microarchitecture with hardware-programmable functional units," *Proceedings of the 27th Annual International Symposium on Microarchitecture - MICRO 27*, no. November, pp. 172–180, 1994.

[30] C. Ralph and R. D. Wittig, "OneChip: An FPGA Processor With Reconfigurable Logic," in *In IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 126–135, 1995.

[31] Z. A. Ye, A. Moshovos, S. Hauck, and P. Banerjee, "CHIMAERA: A High-Performance Architecture with a Tightly-Coupled Reconfigurable Functional Unit," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pp. 225–235, ACM Press, 2000.

[32] R. Gonzalez, "Xtensa: a configurable and extensible processor," *IEEE Micro*, vol. 20, no. 2, pp. 60–70, 2000.

[33] R. Gonzalez, "A software-configurable processor architecture," *IEEE Micro*, pp. 42–51, 2006.

[34] Altera Corporation, "Altera Nios II Processor." `https://www.altera.com/products/processors/overview.html`, 2015.

[35] Altera Corporation, "Nios II custom instruction user guide," tech. rep., 2011.

[36] R. Dimond, O. Mencer, and W. Luk, "Application-specific customisation of multi-threaded soft processors," *IEE Proceedings - Computers and Digital Techniques*, vol. 153, no. 3, pp. 173–180, 2006.

[37] V. Govindaraju, C.-H. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam, and C. Kim, "Dyser: Unifying functionality and parallelism specialization for energy-efficient computing," *Micro, IEEE*, vol. 32, pp. 38–51, Sept 2012.

[38] J. Benson, R. Cofell, C. Frericks, C.-H. Ho, V. Govindaraju, T. Nowatzki, and K. Sankaralingam, "Design, integration and implementation of the DySER hardware accelerator into OpenSPARC," in *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture*, HPCA '12, pp. 1–12, IEEE Computer Society, 2012.

[39] S. Gupta, S. Feng, A. Ansari, S. Mahlke, and D. August, "Bundled execution of recurring traces for energy-efficient general purpose processing," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pp. 12–23, 2011.

[40] M. Haaß, L. Bauer, and J. Henkel, "Automatic custom instruction identification in memory streaming algorithms," in *Proceedings of the 2014 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '14, pp. 6:1–6:9, ACM, 2014.

[41] L. Bauer, M. Shafique, and J. Henkel, "Run-time instruction set selection in a transmutable embedded processor," in *Proceedings of the 45th annual Design Automation Conference*, pp. 56–61, ACM, 2008.

[42] L. Bauer, M. Shafique, and J. Henkel, "Concepts, architectures, and run-time systems for efficient and adaptive reconfigurable processors," in *Adaptive Hardware and Systems (AHS)*, pp. 80–87, IEEE, 2011.

[43] T. Halfhill, "Intel's tiny Atom," *Microprocessor Report*, vol. 22, 2008.

[44] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '04, pp. 75–86, IEEE Computer Society, 2004.

[45] R. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Transactions on Computers*, vol. C-35, no. 8, pp. 677–691, 1986.

[46] M. Ciesielski, P. Kalla, and S. Askar, "Taylor expansion diagrams: A canonical representation for verification of data flow designs," *IEEE Transactions on Computers*, vol. 55, no. 6, pp. 1188–1201, 2006.

[47] D. Gomez-Prado, Q. Ren, S. Askar, M. Ciesielski, and E. Boutillon, "Variable ordering for taylor expansion diagrams," in *Proceedings of the High-Level Design Validation and Test Workshop, 2004. Ninth IEEE International*, HLDVT '04, pp. 55–59, IEEE Computer Society, 2004.

[48] M. Arnold and H. Corporaal, "Designing domain-specific processors," in *Proceedings of the 9th International Symposium on Hardware/Software Codesign*, pp. 61–66, ACM, 2001.

[49] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring network structure, dynamics, and function using NetworkX," in *Proceedings of the 7th Python in Science Conference (SciPy2008)*, pp. 11–15, Aug. 2008.

[50] W. Stein *et al.*, *Sage Mathematics Software (Version 5.8)*. The Sage Development Team, 2013. `http://www.sagemath.org`.

[51] SRISC, "Simply risc s1 core," 2012.

[52] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000.

[53] J. E. Fritts, F. W. Steiling, J. A. Tucek, and W. Wolf, "MediaBench II video: expediting the next generation of video systems research," *Microprocessor and Microsystems*, vol. 33, pp. 301–318, June 2009.

[54] I. Kuon and J. Rose, "Measuring the gap between FPGAs and ASICs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, pp. 203–215, Feb 2007.

[55] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, WWC '01, pp. 3–14, IEEE Computer Society, 2001.

[56] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout, "An evaluation of high-level mechanistic core models," *ACM Transactions on Architecture and Code Optimization (TACO)*, 2014.

[57] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pp. 469–480, ACM, 2009.

[58] D. Müllner, "fastcluster: Fast hierarchical, agglomerative clustering routines for R and Python," *Journal of Statistical Software*, vol. 53, no. 9, pp. 1–18, 2013.

[59] IBM, "ILOG CPLEX Optimizer." http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/index.html, 2015.

[60] D. Kroshko, "OpenOpt: Free scientific-engineering software for mathematical modeling and optimization," 2007–2015.

[61] K. Karuri and R. Leupers, "A primer on ISA customization," in *Application Analysis Tools for ASIP Design*, pp. 93–109, Springer New York, 2011.