How Flexible is Your Computing System?

SHIHUA HUANG, Prodrive Technologies, The Netherlands
LUC WAEIJEN, GrAI Matter Labs, The Netherlands
HENK CORPORAAL, Eindhoven University of Technology, The Netherlands

In literature, computer architectures are frequently claimed to be *highly flexible*, typically implying the existence of trade-offs between flexibility and performance or energy efficiency. Processor flexibility, however, is not very sharply defined, and consequently these claims cannot be validated, nor can such hypothetical relations be fully understood and exploited in the design of computing systems. This paper is an attempt to introduce scientific rigour to the notion of flexibility in computing systems. A survey is conducted to provide an overview of references to flexibility in literature, both in the computer architecture domain, as well as related fields. A classification is introduced to categorize different views on flexibility, which ultimately form the foundation for a qualitative definition of flexibility. Departing from the qualitative definition of flexibility, a generic quantifiable metric is proposed, enabling valid quantitative comparison of the flexibility of various architectures. To validate the proposed method, and evaluate the relation between the proposed metric and the general notion of flexibility, the flexibility metric is measured for 25 computing systems, including CPUs, GPUs, DSPs, and FPGAs, and 40 ASIPs taken from literature. The obtained results provide insights into some of the speculative trade-offs between flexibility and properties such as energy efficiency and area efficiency. Overall the proposed quantitative flexibility metric shows to be commensurate with some generally accepted qualitative notions of flexibility collected in the survey, although some surprising discrepancies can also be observed. The proposed metric and the obtained results are placed into context of the state of the art on compute flexibility, and extensive reflection provides not only a complete overview of the field, but also discusses possible alternative approaches and open issues. Note that this work does not aim to provide a final answer to the definition of flexibility, but rather provides a framework to initiate a broader discussion in the computer architecture society on defining, understanding, and ultimately taking advantage of flexibility.

CCS Concepts: • **General and reference** → **Metrics**; • **Theory of computation** → *Abstract machines;*

Additional Key Words and Phrases: Flexibility, versatility, metric

ACM Reference format:
Shihua Huang, Luc Waeijen, and Henk Corporaal. 2022. How Flexible is Your Computing System? *ACM Trans. Embedd. Comput. Syst.* 21, 4, Article 37 (August 2022), 41 pages.
https://doi.org/10.1145/3524861

37

Shihua Huang and Luc Waeijen contributed equally to this research.
Authors' addresses: S. Huang, Prodrive Technologies, Science Park Eindhoven 5501, Son, 5692 EM, The Netherlands; email: shihuahuang94@gmail.com; L. Waeijen, GrAI Matter Labs, High Tech Campus 68, Eindhoven, 5656 AG, The Netherlands; email: lwaeijen@graimatterlabs.ai; H. Corporaal, Eindhoven University of Technology, De Groene Loper 19, Eindhoven, 5612 AP, The Netherlands; email: h.corporaal@tue.nl.

1539-9087/2022/08-ART37
https://doi.org/10.1145/3524861

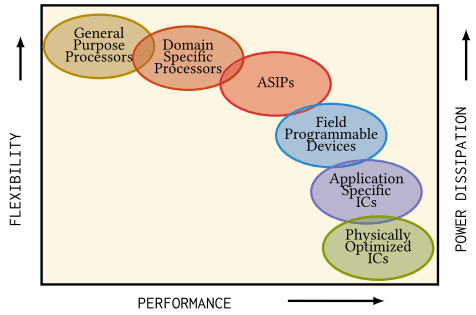
1 INTRODUCTION

Arguably, one of the most famous books in the field is "Computer Architecture — A Quantitative approach" by John L. Hennessy and David A. Patterson [18]. The title itself is concise and apt, so it is interesting the authors opted to add this particular subtitle: a *quantitative* approach. It implies the belief that quantifying design choices ultimately leads to better computer architectures, a message that certainly could be directed towards those who make claims about *flexible* architectures without means of quantifying these claims, or even without as much as a commonly accepted qualitative definition of flexibility. With Moore's law seemingly coming to an end, new advancements in computing will have to be made on the architectural side. To advance the state of the art, fundamental understanding of various trade-offs in computer design is vital. The way forward, therefore, is a quantitative one.

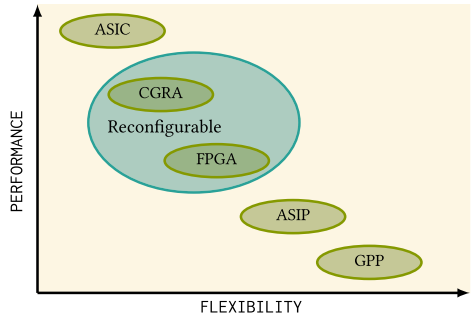
Many key system properties such as performance, power dissipation, and energy efficiency are all well defined in a quantitative manner. With these metrics in place, quantitative, objective comparisons can be conducted between different machines. For flexibility however, such a quantitative (and even qualitative!) definition is lacking, despite its increasing importance in system design. In product research and development, computing platforms are required to sufficiently support new or updated algorithms, as algorithms are changing at a striking speed. Exemplary are the current developments in artificial intelligence, which result in new compute-intensive algorithms at a high cadence. Such rapidly developing markets require systems that can deal with changing applications, which is the property flexibility typically seems to refer to. However, in absence of a proper definition, it is impossible to make solid statements, and compare designs on flexibility.

Despite the lack of a formal definition of flexibility, there appear to be some commonly accepted notions surrounding flexibility. In particular, flexibility seems mainly used to refer to the adaptability of processors to different applications. This leads to the common idea that a programmable processor which can be reused across applications is 'flexible'. On the other hand, a processor with fixed logic such as an ASIC cannot adapt, exposing its inflexibility [36]. As can be seen in Figure 1, the authors of these figures appear to agree with this sentiment. However, there are also some contradictions to this view on flexibility. For example, in Figure 1(a), among programmable processors the field-programmable devices are claimed by the authors to be less flexible than software programmable processors due to their inadequate programmability [25]. Unfortunately, the term "programmability" is also ill-defined here. Perhaps the best definition of programmability in existence is to check Turing completeness of a programmable device, but this would leave only two classes of programmability making it a measure with low practical value. Another perspective on flexibility refers to how well a processor supports different applications, in which case FPGAs could be seen as the most flexible, since any hardware, including DSPs, GPUs, and CPUs, can be instantiated on FPGAs. Apart from this debate on how to rank the flexibility of architecture classes, perhaps even more worrisome are the contradicting claims on relations between flexibility and other metrics. In Figure 1(d) A. Osman El-Rayis equates flexibility to area, whereas T. Noll sees it as directly related to power dissipation in Figure 1(a). While this is definitely not an exhaustive list of views on flexibility, it painfully exposes how the lack of a formal definition leads to a wild-west of claims and conflicting visions, none of which can be backed up with objective measurements.

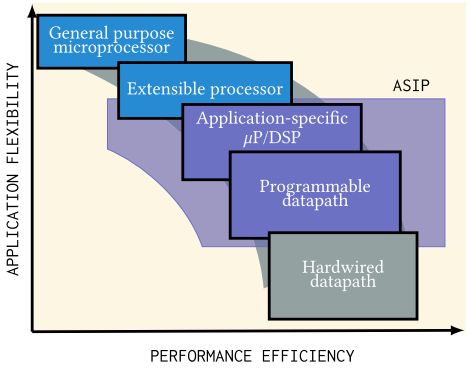
Despite the greatly varying interpretations of flexibility, many seem to agree that there may be interesting relations and trade-offs between flexibility and other properties, such as performance and energy efficiency. As illustrated in Figure 1, processing architectures have been evaluated and ranked in terms of flexibility, performance, power dissipation, and area [12, 24, 25, 35, 55]. A variety of architectures have been developed which claim to balance energy efficiency and flexibility. The development of domain-specific functional units and a transition to heterogeneous multi-core



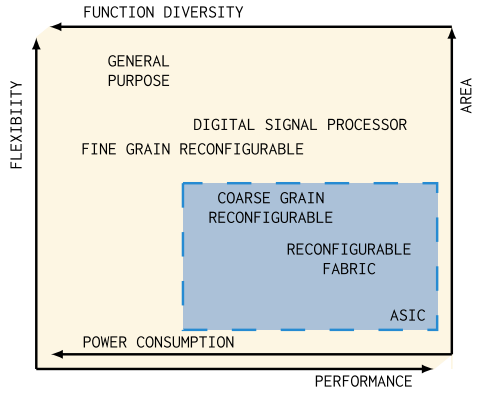
(a) Flexibility related to performance and power according to T. Noll [24, 25]. Note that according to this figure flexibility is directly related to power dissipation.



(b) Flexibility versus performance plot by G. Ndu [35]. The ordering of the flexibility of architecture classes aligns quite well with the ordering given in figure 1a, although curve is quite different.



(c) 'Application Flexibility' versus 'Performance Efficiency' by M. Willems from Synopsys [55]. Unfortunately no further definition of both these metrics is provided in the accompanying text.



(d) An interesting graph by A. Osman El-Rayis [12] which relates many metrics including flexibility. In contrast to figure 1a flexibility is here claimed to be directly related to area.

Fig. 1. Collection of published figures with claims about the flexibility of architecture classes, and relations between flexibility and other metrics such as performance and power. Note that none of the axes in these figures are labelled with units.

systems are a testament to this notion [13, 17, 29]. These hypothetical relations suggest that understanding flexibility is beneficial when designing a system, enabling informed trade-offs.

To overcome the lack of understanding of flexibility, this work sets out to provide both a qualitative and quantitative definition of flexibility. It should be noted though that, with such a fragmented landscape of interpretations of flexibility, the authors are under no illusion that it is possible to unify the field and reach consensus without a wider discussion. Instead, this work is to be seen as a first attempt, which does not so much aspire to provide a definitive answer, as it hopes to be thought provoking and spark a discussion within the community.

To arrive at a quantitative measure for flexibility, a qualitative definition is established by exploring uses of the term flexibility in literature and then examining various options. Based on this qualitative definition, a quantitative measure is derived. In the translation of flexibility, from a qualitative term to a quantitative definition, there exist several degrees of freedom. The final choices made in this translation are motivated extensively. However, more importantly, the alternatives

are discussed systematically in similar detail. The intention is that this systematic approach can provide an initial framework for a broader discussion in the community on how flexibility should be defined, such that eventually a standard accepted metric can be established.

To validate the metric proposed in this work, in total 14 applications are benchmarked on 25 different **commercial off the shelf (COTS)** platforms. It is shown that results align with several common concepts of flexibility found in literature. For example, GPUs deliver the highest performance in general for the used parallel benchmarks, but sacrifice in terms of flexibility, compared to **general purpose processors (GPPs)** in alignment with Figure 1(a). Furthermore the flexibility of 40 application specific architectures from literature is determined to evaluate the relation between specialisation [14] and flexibility.

The main contributions of this work can be summarized as follows:

(1) Survey of flexibility in computing and other engineering fields (Section 2);
(2) A qualitative and quantitative definition of flexibility (Section 3);
(3) Definition of intrinsic workload to normalize performance (Section 4), and accompanying open-source tool [19] for automated extraction (Section 5.1.4);
(4) Evaluation of the proposed metric on 25 COTS platforms for 14 benchmarks, and 40 application specific processors taken from literature (Section 6);
(5) In-depth comparison with alternative definitions of flexibility/versatility (Section 7);
(6) Extensive discussion on the proposed metric, and various alternatives, placing it in context of the current state of the art (Section 8).

The remainder of the paper is organized as follows: Section 2 presents a survey on flexibility definitions in the literature, both in the field of computer architecture, as well as related technical fields. Section 3 introduces both a qualitative and quantitative definition for processor flexibility based on the collected views in the survey. A novel normalization method based on the intrinsic workload of applications is included in Section 4. Section 5 explains the experimental setup, the implementation of the workload estimator, and the methodologies applied in this work. The flexibility results are analysed in Section 6. Comparison with alternative definitions is provided in Section 7. In-depth reflection and extensive discussion are presented in Section 8, which places the proposed definition in context of the field. Finally, Section 9 concludes this work.

2 SURVEY OF FLEXIBILITY IN LITERATURE

In the field of computer architecture, few studies have striven to define and quantify processor flexibility. Therefore, this section starts with discussing the existing flexibility definitions in other fields.

Various other fields have more properly defined flexibility, as demonstrated by the three following examples:

- A generic viewpoint on system flexibility is provided by Chryssolouris, who defines flexibility as the sensitivity of a system to (external) *changes* [8]. Lower sensitivity is understood to indicate higher flexibility, as the system is apparently able to operate relatively unaffectedly under the external changes.
- Conceptually identical is the definition proposed by Kellerer et al. who state that the flexibility of electronic networks refers to the ability to support new requests, such as *changes* in the requirements or new traffic distributions [27].
- In power systems, flexibility is also based on external *changes*. More precisely, it is defined as the ability of a power system to deploy its resources in response to *changes* in the net load, which is the residual demand that must be supplied after the depletion of renewable energy [31].

In general, these examples consider flexibility as a system property and quantify flexibility as the *insensitivity of the system based on external changes*, instead of formulating flexibility as a function of diverse system parameters. This approach is transferable to computing systems, as will be outlined in this work.

In literature related to computer architecture, approaches to define processor flexibility can be divided in two categories:

(1) Definitions that regard flexibility as an *intrinsic static property* of a system.
(2) Definitions that regard flexibility as an *extrinsic mutable property* of a system, dependent on and measured under the influence of external applications.

Works that fall into the first category are discussed in Section 2.1, while the second category is elaborated in Section 2.2.

2.1 Definitions of Flexibility as an Intrinsic Static Property

One definition that regards flexibility as an intrinsic property is proposed by D. Stigall et al. [47] as early as 1975. In their definition a computer is seen as major memory/compute units and their datapath connections. The more connections between the major components, the more options the machine has, and thus the more flexible it is. The authors also quantify flexibility, namely as the ratio of the number of data paths that connect major components to the maximum possible number. The idea is interesting but does not seem to hold for modern machines. For example, when directly applying this concept to an SIMD machine and a multi-core with the same number of cores, an SIMD processor that has fewer components (only a single instruction memory and decoder, instead of one per core), would usually have higher connectivity among components than a multi-core processor, implying higher flexibility. This conclusion seems counter-intuitive however, since an SIMD can only execute a single instruction at a time on all compute elements, while a multicore can execute different instructions on its compute units. Therefore, the set of operation modes of the multi-core is a superset of the SIMD modes, which implies an SIMD cannot be more flexible than a multi-core system.

Another intrinsic definition category is processor versatility as proposed by K. van Berkel [26]. As shown in (1), processor versatility is defined as the average number of instruction bits per useful operation. The amount of useful operations is extracted according to a complexity analysis of a single algorithm. Based on the intuition that when more bits are used to encode instructions, the processor is more versatile as more options are available. Therefore, versatility is a property of the **instruction set architecture (ISA)** and is independent of particular implementations or executed applications. For instance, versatility increases if the ISA is extended with special instructions serving dedicated hardware, while potentially being useless to accelerate the applied applications. Although the work by K. van Berkel [26] is the most rigorous attempt to formally define flexibility to date, some aspects hinder its application. Conducting complexity analysis of applications to obtain useful operations is challenging without manual effort. Using operations as basic units implies that different operations are weighted equally, such as multiplication and addition. This seems somewhat arbitrary given that the area or energy footprint of a hardware multiplier, for example, is many times that of an adder in the same technology. Another weakness of this metric is that it cannot be applied to all systems, such as processors that do not execute clock-based instructions, such as FPGAs.

$$versatility = \frac{average\ instruction\ size}{number\ of\ useful\ operations\ per\ instruction} \tag{1}$$

However, the concept of measuring the required bits to execute a task is intriguing, and possibly has a use of its own. Therefore, this versatility metric is further discussed in Section 7.1.

2.2 Definitions of Flexibility as an Extrinsic Mutable Property

Most flexibility definitions fall into the second, extrinsic category, and are considered a property measured under the influence of external applications. Sze et al., for example, provide the following view on flexibility: *"Flexibility refers to the range of DNN models that can be supported on the DNN processor and the ability of the software environment (e.g., the mapper) to maximally exploit the capabilities of the hardware for any desired DNN model."* [48]. Here, the DNN models, i.e., extrinsic properties, are used to define the flexibility of a system. Their conclusion on flexibility therefore also is: *"...to assess the flexibility of DNN processors, its efficiency (e.g., inferences per second, inferences per joule) should be evaluated on a wide range of DNN models".* Although not a quantifiable definition per se, the notion of defining the flexibility of a system in relation to the influence of relevant extrinsic properties is clearly present. Furthermore, it is important to note that various forms of flexibility are implied, since the effect of an extrinsic property can be expressed in terms of performance (inferences per second), and energy efficiency (inferences per joule). Also interesting is the inclusion of the software (mapper) into the equation, i.e., system flexibility does not only depend on the hardware, but also on the supporting software.

Tomusk et al. propose to quantify the flexibility of Single-ISA heterogeneous processors with entropy-based diversity [51]. The idea is that different cores in a flexible heterogeneous processor can cover more of the design space. Exploring the design space is achieved by selecting the cores of the system that are Pareto optimal for power and performance. Higher spread on these Pareto cores means better flexibility. However, this definition is strictly limited to heterogeneous processors, ruling it out as a general definition for all computing systems.

Fasthuber et al. propose a different model to define computer architecture flexibility [13]. The proposed model extracts system requirements from a set of applications to check if architectures provide sufficient flexibility to support the minimum requirements in a true/false manner. For instance, in case a hardware divider is imperative to reach the required performance for a division, a processor performing division by software emulation fails to meet the performance requirement. This model assesses architecture flexibility based on external applications, examining how well architectures support diverse applications. However, it is challenging to apply this model generically because of the need for a set of hard boolean requirements. Moreover, the range of the scale is severely limited, since the flexibility is the result of counting the number of met requirements. If the set of requirements is small, distinguishing various systems may be impossible.

Apart from the intrinsic versatility metric as proposed by Van Berkel [26], there is a competing extrinsic definition for versatility by Rabbah et al. [45]. To distinguish the two versatilities in this paper, the metric defined by Rabbah et al. in their VersaBench paper will be referred to as "VersaBench Versatility" or V_s. VersaBench Versatility is defined as the **geometric mean (GM)** of a processor's performance over a vector $\vec{X} = [x_1 \cdots x_n]$ of benchmark applications, normalized to the best execution time known for each of those applications $t_{fastest}(x_i)$, where $x_i \in \vec{X}$ (Equation (2)).

$$V_s = \left(\prod_{i=1}^{n} \frac{t_{fastest}(x_i)}{t_s(x_i)} \right)^{\frac{1}{n}} \tag{2}$$

This can be interpreted as the mean slowdown of a processor compared to an idealized fastest known execution time per application. As such, VersaBench Versatility has a range of $0 < V_s \leq 1$. Compared to the definitions provided by Tomusk et al. and Fasthuber et al., this approach resolves

the limitations on applicability of the metric to more diverse architectures. The use of only execution time measurements further increases the practicality of the proposed metric. However, improvements in VersaBench Versatility can result from an absolute increase in performance, i.e., an increase in clock speed to boost performance can effectively improve VersaBench Versatility. This makes VersaBench versatility and performance directly related, which we argue should be distinct, orthogonal features. In-depth analysis and comparison with the flexibility metric as proposed later in this work are provided in Section 7.2, and furthermore show that the normalization proposed by Rabbah et al. [45] is a mathematical unit operation and does not contribute to a change in the final V_s.

Although not explicitly targeting flexibility, the work of Fisher et al. [14] on customizing processors exhibits interesting parallels with work on flexibility in literature. In their work, Fisher et al. set out to optimize a VLIW processor for a set of tasks, which is again the extrinsic factor. It is argued that instead of optimizing a processor for only one application, some performance may be sacrificed for that application to achieve a better average performance for the entire dataset. In essence, by sacrificing performance for one application to benefit the overall benchmark set, it can be argued that the flexibility of the processor has been improved. It is important to distinguish though that the method applied by Fisher et al. still tries to obtain a higher overall performance, and does not necessarily consider minimizing the impact of external changes. A quantified analysis of this method and how it relates to the concept of flexibility is provided in Section 6.2.

The related work discussed in this section shows that there are tremendous diversities in understanding and quantifying flexibility. Overall, flexibility has more frequently been defined as an extrinsic metric based on *external changes*, than an intrinsic property. However, it can also be observed that the inherent properties of most of those definitions limit the scope of application. With this in mind, the metric proposed in this work aims to avoid this pitfall, and also be practical to apply generically.

3 DEFINING FLEXIBILITY

Despite a few valiant attempts to define flexibility for computing systems in the existing literature, it can be concluded there is no consensus in the community as to what flexibility exactly is, let alone how to objectively measure it. Unifying the various views on the topic into a single coherent definition is a daunting task, yet one that has to be faced if the rewards are to be reaped. This section outlines our attempt at defining flexibility for computing systems. Starting from a qualitative definition in Section 3.1, a quantitative metric is then derived in Section 3.2. In Section 3.2.1, several crucial properties of a universal flexibility metric are defined and proven to hold for the proposed metric. Finally, Section 3.3 details the scope of applying the proposed metric.

3.1 Qualitative Definition

Before determining a quantitative definition of flexibility, there has to be agreement on a qualitative definition. As outlined in Section 2, a recurring theme when dealing with flexibility in literature is *external changes*, or in particular, a system's *response* to external changes. A natural translation of this notion to computing systems is to regard changing applications as the external changes, while any secondary metric, such as performance or energy efficiency, can be used to express a system's response. Consider, for example, the benchmark data for two systems in Figure 2. As applications change, so does the (normalized) performance of these systems.

For the particular case in Figure 2: *Which system is more flexible?* Some may argue System I is more flexible, as System I can maintain the highest average performance when applications change. However, we postulate such reasoning is a fallacy, and that performance has to be an orthogonal measure to flexibility. Were this not the case, then flexibility would merely be a synonym

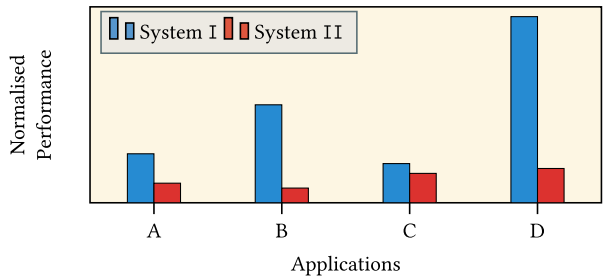


Fig. 2. Given the performance of two hypothetical systems I and II, normalised to some baseline system, System I consistently outperforms System II. However, the performance of System II is much less influenced by changing applications. Which system is more flexible?

of "average performance", and not an independent metric as appears to be the common notion. Rather, we argue that the system which supports different applications equally well is more flexible, regardless of its average performance. In Figure 2, System II obviously has lower performance than System I, however, it is stabler under application changes. This is a desirable property orthogonal to performance. For example, during the design of a computing platform with cost constraints, a processor has to be selected but the final applications are still subject to change. In this case, it could be beneficial to select a processor with overall lower, yet sufficient, performance, but higher flexibility, such that the performance is likely to be still sufficient when applications do change.

In the case of Figure 2 flexibility is defined in relation to performance variability. However, other established metrics can be used freely, such as energy efficiency, area efficiency, or a hybrid cost function. How much the secondary metric changes resulting from changes in applications is an indication of the flexibility of the platform in that regard.

Based on the reasoning above, we arrive at the following qualitative definition of flexibility:

> *Compute system flexibility refers to the invariance of a system's normalized*[1] *performance, energy efficiency, area efficiency (or other secondary metrics), to change of application.*

In particular, when the secondary metric is affected more by changes in an application, the system is considered to be less flexible.

Although it is just a qualitative definition, some general observations as to how it aligns with several notions regarding flexibility can already be made. Consider, for example, an arbitrary set of benchmark applications that expose different levels of **data-level parallelism (DLP)**. When mapped to a GPU, applications with high levels of DLP would benefit from the many vector cores and achieve high performance. Applications with limited DLP, however, would not be able to run efficiently on a GPU, and consequently, achieve low performance in comparison. Thus, for a mixed, arbitrary benchmark set a GPU would not be very flexible. In contrast, a simple single-core CPU without vector extension would not provide an unbalanced advantage for applications with high levels of DLP. Therefore, it would be ranked more flexible than the GPU, which aligns with commonly accepted notions. Similar examples can be made for various classes of architectures, such as CPUs with advanced branch predictions and applications with complex control flow, or DSPs and algorithms that require multiple floating-point multiply-accumulate operations. Specialization towards only a subset of relevant applications may improve overall performance but could degrade flexibility.

[1]N.B.: This 'normalization' is further clarified in Section 3.2.

The preceding example also illustrates that the selection of benchmark applications is an important factor in determining a system's flexibility. After all, if a dedicated parallel benchmark set was selected, the GPU would be ranked as more flexible. It is worth pointing out this is not a weakness nor flaw in the definition of flexibility, but merely emphasizes that benchmarks should be selected based on the application domain a system is targeting. Similarly, it makes no sense to use a graphics benchmark on a CPU, when the system is targeted for handling search engine queries. Proper benchmark set selection is just as crucial to obtain meaningful flexibility results as it is for measuring any other system property. When done properly though, the obtained flexibility ranking will be representative for the selected application domains.

Finally, it is highly important to note that flexibility as defined here can be artificially raised. By inserting `nop` operations in the fastest applications, the performance of all applications can be lowered to match the lowest-performing application. This would result in the most flexible system, although the overall achieved performance is degraded. For real-time systems, such an approach may not even be undesirable, as long as the performance requirements are still met. However, to account for the loss in performance, energy efficiency, or any other secondary metric, flexibility should always be reported coupled to these metrics. One possible way to couple flexibility with other metrics is through a compound metric, such as the classical **energy-delay product (ED)**, or **energy-delay-power (EDP)**.

3.2 Quantitative Definition

This section translates the qualitative definition of flexibility to a quantitative measure. In particular, the focus is on how to quantify "the invariance of a system's response". A measure has to be found which expresses variations in system performance, energy efficiency, or other secondary metrics. Several such measures for quantifying statistical dispersity, or variation, among data points exist. This section qualitatively explores these options, and finally selects the most suited approach to quantify flexibility.

There are two classes of variation measures:

(1) *Robust measures* are resilient to extreme values in a dataset, and in general, try to reduce the effect of outliers in the data. A typical example of robust measures is the **median absolute deviation (MAD)** (Equation (3)), defined as the median of the absolute deviations from the median of the original data. The MAD ignores a small number of extreme values, and only focuses on the median of the dataset.

$$MAD(\vec{X}) = median\left(|x_i - median(\vec{X})|\right) \tag{3}$$

where $\vec{X} = [x_1 \cdots x_n]$ is a vector of measurements.

(2) *Conventional measures*, in contrast, are sensitive to extreme values [54]. **Arithmetic standard deviation (ASD)** (Equation (4)) and **geometric standard deviation (GSD)** (Equation (5)) are two such conventional measures, both of which describe the dispersion degree of a data set.

$$ASD(\vec{X}) = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - AM(\vec{X}))^2},\ where\ AM(\vec{X}) = \frac{1}{n}\sum_{i=1}^{n} x_i \tag{4}$$

$$GSD(\vec{X}) = \exp\left(\sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\ln\frac{x_i}{GM(\vec{X})}\right)^2}\right),\ where\ GM(\vec{X}) = \left(\prod_{i=1}^{n} x_i\right)^{\frac{1}{n}} \tag{5}$$

Robust measures are particularly suited for noisy measurements with plenty of data points. In computer architecture, however, measurements are easily repeatable, allowing noise to be filtered by alternative means. Furthermore, the number of applications in benchmark sets is often quite limited, and as such ignoring points risks ignoring important data. Therefore, the conventional measures are more suited to represent dispersity in a set of benchmark applications.

The fundamental difference between the two conventional measures is the used average: **arithmetic mean (AM)** versus **geometric mean (GM)**. The AM simply characterizes the average value of the dataset by dividing the sum of all points by the length of the dataset. Thus, the ASD indicates the average distance of data points in the dataset to the AM, and has the same unit as the dataset. Since the AM and ASD are sum-based values, they are appropriate for additive processes. Different from the AM, the GM takes the product of all numbers, and then raises it to the inverse of the length of the dataset. Because of this, the GSD as defined in Equation (5) is a multiplicative factor and does not maintain the original dimension of the data [28]. When dealing with multiplicative relationships such as growth rate and speed-up, the ASD over-estimates data dispersity, while the GSD as a product-based value is the correct average to use [15, 34].

Based on this, the GSD is selected as the measure of dispersity for flexibility. In particular because, as also stated in the qualitative flexibility definition, benchmark data is to be first normalized when deriving flexibility. In the case of performance, the inverse of absolute runtime would not give an accurate view of which applications are supported better than others. Some applications may simply require more work than others, and thus the runtime needs to be normalized (more on this normalization in Section 4). Similarly, energy efficiency is the energy consumption normalized to the amount of work performed by each application. This normalization results in a multiplicative relation to the normalization baseline, and thus GSD is the only correct measure of dispersity.

Concluding the quantitative definition of flexibility:

> *Compute system flexibility is defined as the inverse of the geometric standard deviation of a system's normalized performance, energy efficiency, or other secondary metric, within a benchmark set with measurement vector* $\vec{X} = [x_1, \ldots, x_n]$ *(Equation (6)).*

$$Flexibility(\vec{X}) = \left[GSD(\vec{X})\right]^{-1} = \exp\left(-\sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\ln\frac{x_i}{GM(\vec{X})}\right)^2}\right), \text{ where } GM(\vec{X}) = \left(\prod_{i=1}^{n} x_i\right)^{\frac{1}{n}} \quad (6)$$

3.2.1 Flexibility Metric Properties. Validation of a new metric is a paradox, as there is no established ground truth available. However, it is possible to derive several necessary properties a flexibility metric should adhere to. This section discusses these properties and proves them for the proposed metric.

(1) Flexibility in performance should be independent of scaling the platform frequency (equally across all benchmarks). I.e., if a given platform executes applications at F cycles per second, it should measure the same flexibility if it for example runs at $0.5F$ for all applications. This holds for the proposed metric as the GSD is invariant to multiplicative scaling (see Lemma A.3). Note that this scaling over all benchmarks also holds for platforms that have a difference frequency per application, such as FPGAs. If all applications are executed at half *their* original speed, the measured flexibility will be the same.
(2) Stricter than the previous property, flexibility should be independent of performance. Indicated by Lemma A.1 and A.2, the increase of an element in a positive dataset always results in an increasing GM. However, the GSD value can increase or decrease, which depends on how the increase of an element impacts dataset diversity.

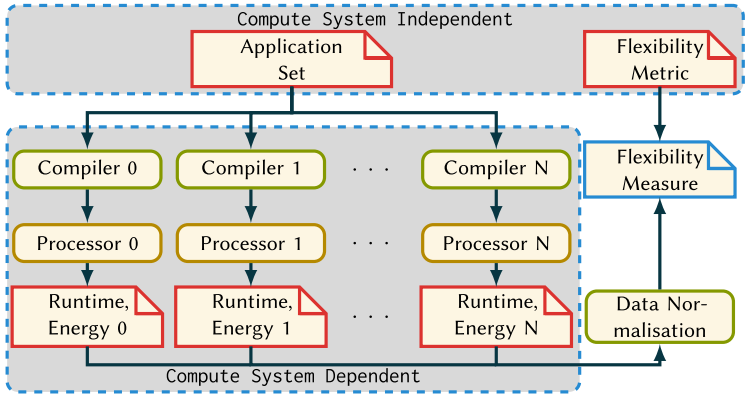


Fig. 3. Framework for flexibility measurements.

The two properties described are absolutely essential for any flexibility metric. Apart from these two, there are two properties which are nice to have, but not strictly required. For completeness it is shown that these property hold for the proposed metric.

(3) Preferably, flexibility, as a multiplicative measure, is invariant to using the reciprocal of the underlying metric. This property is particularly useful as it decouples clock frequency, and with that to some degree the technology node, from flexibility. Furthermore, in the case of flexibility in relation to energy, the flexibility calculated over J/op will be the same as the flexibility calculated over op/J. Lemma A.7 shows that this is the case for the proposed definition, since $GSD(\vec{X}/\vec{Y}) = GSD(\vec{Y}/\vec{X})$.

(4) It is convenient if the metric is bound to a fixed range. In the case of the proposed metric it can trivially be shown that flexibility has a range of $(0, 1]$. A flexibility of one is achieved when an architecture has the exact same speed up for each application in the benchmark set compared to the normalization reference, as further explained in Section 4.

3.3 Flexibility Scope

One aspect of flexibility that has not been addressed so far is the measurement scope. In the case of performance measurements, it is not only the performance of a processor that is measured, but also that of the surrounding memory system, interfaces, and even the compiler. This is not exclusive to performance measurements but also holds for energy, power, and many other metrics. The fact that the compiler is part of the measured system is in fact common practice, but often overlooked when publishing results. Therefore, we like to explicitly state the compiler used to perform benchmarking of a system should always be part of the results.

For the flexibility measurements in this work, the system border is drawn at the compiler, and the benchmark code itself is taken as universal for all platforms. When measuring across different platforms this may not always be feasible, so different choices can be made in specific situations. The recommendation is, however, to use a cross-platform language benchmark such as OpenCL. A benchmark set with multi-language support is a good alternative if no single language can support the systems under test. In general, the procedure of measuring flexibility will follow the flow illustrated in Figure 3.

The first step is to compile and run the same application set on the target systems, and measure the desired secondary metrics such as performance and energy consumption per benchmark. Next follows normalization of the obtained results. Data normalization is a prerequisite to ensure

benchmark results of diverse applications are comparable, as further discussed in Section 4. Finally, the proposed flexibility metric is computed from the normalized data, resulting in a flexibility ranking of the measured platforms.

4 NORMALIZATION TO INTRINSIC WORK

Before the GSD can be computed, secondary metrics such as execution time and energy consumption, measured from diverse applications, need to be normalized. The reason is that typically applications in a benchmark set represent inequivalent computational workloads. For instance, applying Gaussian filters with different kernel sizes results in different workloads. Therefore, normalization based on workload is required before any data analysis and comparison [40].

The general approach is normalizing to a reference set, i.e., each application x_i of a vector of applications $\vec{X} = [x_1, \ldots, x_n]$ is normalized according to:

$$m_{norm\_baseline}(x_i) = \frac{m_{target}(x_i)}{m_{baseline}(x_i)},$$

where m_{target} evaluates an arbitrary metric m for the target architecture for application x_i, and $m_{baseline}$ does the same for x_i on the baseline machine. However, determining a proper baseline for flexibility poses a challenge. Simply taking benchmark results of one system as reference implicitly makes the baseline system "the most flexible" by definition. For instance, when using a basic RISC-type processor as the baseline as is often done by Hennessy and Patterson [18], normalizing to itself transforms each value in the dataset to one, resulting in no deviation and a flexibility of one. Consequently, no system could then be more flexible than the baseline RISC processor, or whichever platform is selected as the baseline. This choice seems rather arbitrary, and a more fundamental baseline is desirable.

The key concept is that normalization is applied to equalize an imbalance in workload that each application represents. The underlying notion is thus that applications describe a certain amount of *work*. Normalization on this *intrinsic workload* W_{int} yields the following normalization procedure:

$$m_{norm\_intrinsic\_workload}(x_i) = \frac{m_{target}(x_i)}{W_{int}(x_i)}$$

This normalization results in measures as "intrinsic work per second" for performance, and "energy per unit of intrinsic work" for energy efficiency, yielding comparable numbers between various platforms. Unfortunately, a measure for intrinsic workload also does not exist. There are many possible viewpoints on how intrinsic workload could be defined, yet this section will focus only on the one selected for normalization in this work. Section 8 will, on the other hand, explore several alternatives.

Under the assumption that computation indeed is equivalent to work (something that can be questioned from a physics point of view as will be discussed in Section 8), the problem condenses to finding a unit for this work. Something often used in computer architecture is to count one RISC instruction as one unit of work. However, this implies multiplication and even division would represent the same amount of work as an addition or a logic and-operation. This is rather counter-intuitive, as the hardware complexity of a hardware divider is significantly greater than that of a logic and, i.e., $O(b^2)$ for a b-bit wide division, versus $O(b)$ for a logic and of b bits. This implies division is fundamentally more complex than a logic operation. A possible way of weighting RISC instructions then is by the complexity of their equivalent combinatorial circuits.

Taking this one step further, the division of work into RISC operations is also rather arbitrary. From a purely theoretical viewpoint, it can be argued that the workload of an application is represented by its combinatorial circuit, i.e., the combinatorial circuit that statically represents the entire

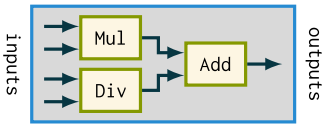


Fig. 4. Application expressed as a single combinatorial circuit.

application, reading inputs and producing the final outputs without a (clocked) state in between as illustrated in Figure 4. Such a circuit would clearly be completely impractical, but it can be argued than when written in a minimal form, i.e., with minimal basic gate (2 input — 1 output) count, it represents the intrinsic workload of the application. The gates toggling during the execution of this circuit approximate the minimum required toggles to complete the computation. Interestingly, memory and control flow operations are not required in such a completely combinatorial, spatial circuit, demonstrating that such operations are in essence an artefact of stateful Turing machines.

Note that this massive combinatorial circuit would have to be written in minimal form though, something intractable with modern technology since logic minimization is proven to be non-polynomial [6]. From a theoretical point, obtaining such a minimal circuit would be interesting, but to arrive at a practical measure for flexibility as is the goal of this survey paper, a more pragmatic approach has to be employed. Rather than approaching this minimal circuit bottom up, one solution is to return to the roots of practical computing and approach it top down. Instead of finding the absolute minimal circuit, the circuit can be divided into elementary blocks with common functionality. Optimizations are not employed across these blocks to keep the design tractable. As the only requirement for these elementary blocks is that the set is Turing complete, there are many options. Here, however, it is possible to fall back on decades of research in computer design. A natural choice would be RISC-like elementary operations, such as addition and multiplication. Essentially, this circles back to the earlier idea of weighing RISC operations by the complexity of their equivalent circuitry, but with a notable exception: Those operations in an application that deal with control flow operations should not be counted towards the intrinsic workload. Since memory operations are also an artefact of stateful machines, they could too be omitted, but since they play an ever more important role in modern technology this work proposes several methods to still take memory into account. The effect of memory operations is further investigated in the experiments described in Section 5.

Practically, the intrinsic workload of an application as defined above can be approximated automatically by leveraging the **intermediate representation (IR)** of the LLVM compiler framework. To be able to deal with many input languages and target platforms, LLVM front-ends translate code into a generic intermediate instruction set, the IR. From this generic IR, the back-ends generate target-specific code. This IR has to be very generic to support as many languages and platforms as possible, and as such is a good candidate to use in an automated intrinsic workload estimator. Furthermore, it has the advantage that operations related to control flow and memory are distinguishable from other operations, and as such can be rejected for the workload estimation. This particular approximation gives rise to the following definition:

> *Approximated intrinsic workload is given by the dynamic IR instruction count of all operations not related to control flow (and optionally memory), weighted according to the circuit complexity of the operations.*

More details on how this procedure is automated, and the weighting of the individual IR instructions used in this work can be found in Section 5.1.4.

Table 1. Overview of GPUs

Processor	Chip	Archi	#Cores	Compiler
Tegra K1	GK20A	Kepler	192	nvcc 6.5
GTX 570	GF110	Fermi	480	nvcc 7.5
GTX TITAN	GK110	Kepler	2688	nvcc 7.0
GTX 750 TI	GM107	Maxwell	640	nvcc 7.0

5 EXPERIMENTAL SETUP

Now that a flexibility metric has been established, measurements of various systems can be performed to both validate the metric against commonly accepted ideas surrounding flexibility, as well as investigate hypothetical relations between performance, energy, area, and flexibility. To achieve this a wide spectrum of computer architectures is examined in this work, including **commercial off the shelf (COTS)** CPUs, GPUs, FPGAs, and DSPs. The selection of these COTS systems, the selected benchmark set, and details regarding compiler settings and performed experiments are described in Section 5.1. Inspired by the work of Fisher et al. [14] the flexibility of custom, or **application specific, processors (ASIPs)** is also investigated. The setup and parameters of the related experiments are described in Section 5.2.

5.1 Commercial Off the Shelf Processors

Measuring the flexibility of various off the shelf processors provides insight into hypothesized relations between flexibility and metrics such as performance, energy efficiency and area efficiency. In particular, it is interesting to see if general notions surround architecture classes such as (multi-core) CPUs, GPUs, FPGAs, and DSPs can be seen back in the measured flexibility. This section describes the selection of 25 COTS systems in Section 5.1.1, and the selected benchmark set in Section 5.1.2. Since compilers are considered part of the system, Section 5.1.3 deals with the used compilers and their specific settings. The workload estimation of the benchmark set is captured in Section 5.1.4, and finally, Section 5.1.5 describes the measured properties and their relations.

5.1.1 Selected Systems. In this work, flexibility measurements are conducted on 25 different systems. Applications are directly executed on real GPUs and CPUs. For the embedded DSPs, cycle-accurate simulators from the manufacturer have been employed to extract execution times. The FPGAs, finally, have been characterized using **high-level synthesis (HLS)** combined with post place & route clock speed reporting. Note that in contrast to the other considered systems, for the FPGAs the clock-speed in fact varies per application, as measured using post place & route clock speed estimation per application, target pair. Compilers, as a part of target systems, are tuned for maximum optimization where possible, to exploit the capability of the target processors as good as possible with the given code-base.

- **GPU:** Aimed at comprehending the difference of flexibility between desktop and embedded GPUs, one embedded GPU Tegra K1 and three desktop GPUs are evaluated in this work by compiling the CUDA [39] versions of the applications. Note that the default datasets in the provided C and CUDA version varies, hence the dataset sizes are modified to be equal, ensuring the application workload is consistent over all platforms. The precise platforms and used compilers are listed in Table 1.
- **CPU:** In total, 10 CPUs are included, 6 Intel CPUs, and 4 ARM CPUs, to distinguish and compare the flexibility of embedded and desktop/server CPUs. The benchmarks are compiled

Table 2. Overview of CPUs

Processor	ISA	Micro-architecture	Cores	#Threads	Compiler
i76700	x86_64	Skylake	4	8	gcc 4.8
i74770	x86_64	Haswell	4	8	gcc 4.8
i7-960	x86_64	Bloomfield	4	8	gcc 4.8
i7-950	x86_64	Bloomfield	4	8	gcc 4.8
i7-920	x86_64	Bloomfield	4	8	gcc 4.8
Pentium 4	x86_64	Northwood	1	2	gcc 4.8
Processor	**ISA**	**System**	**Cores**	**#Threads**	**Compiler**
Cortex A15	ARMv7	Nvidia JTK1	4+1	4+1	gcc 4.8
Cortex A9	ARMv7	Odroid U3	4	4	gcc 4.8
Cortex A53	ARMv7	RPi3 Model B	4	4	gcc 6.3
ARM1176	ARMv6	RPi1 Model B	1	1	gcc 6.3

with the *gcc* compiler [49]. Table 2 provides detailed information of the examined CPUs and the used compilers.

- **FPGA:** As PolyBench/ACC does not include applications described in hardware description languages, Vivado **High-level Synthesis (HLS)** [56] is utilized with it's default settings to transform C applications into **register transfer level (RTL)** code, which can be directly targeted to Xilinx programmable devices. In Vivado HLS v2018.2 [56], when synthesizing a C function, a report is generated which provides performance metrics such as loop and function latency in clock cycles. To obtain more accurate estimates of resource utilization and the achieved clock period, the resulting designs have been synthesized towards the target FPGA platforms.

  Unfortunately, when the application involves a variable loop bound, Vivado HLS fails to compute the iteration count required for performance analysis. An example of code where Vivado HLS v2018.2 fails is shown in Code 1, where variable k causes the Vivado's loop analysis to fail for the loops L2 and L3.

```
1  for(k = 0; k < m; k++)                  (L1)
2     for(i = k + 1; i < m; i++)           (L2)
3        for (j = k + 1; j < m; j++)       (L3)
4           A[i][j] = A[i][k] * A[k][j];
```

Code 1. Example loop-nest with bounds of L2 and L3 based on variable k.

Fortunately, this scenario occurs only in four out of 14 benchmarks, specifically *correlation*, *covariance*, *gramschmidt*, and *lu* as described in Table 5. Nonetheless, for these four cases an alternative method is required to obtain performance estimates. A possible solution would be to use C/RTL Co-simulation in Vivado HLS, which simulates the application at the RTL level. However, for the selected benchmarks the simulation runtimes are prohibitively high, as well as the enormous amount of memory required for the simulation, which renders this option infeasible. Since the benchmark set targets polyhedral applications with strictly static control flow, manual derivation of the iteration counts is fortunately quite straightforward. Therefore, in this work manual static loop analysis is utilized to derive an *approximate* cycle count.

As an example, the latency of the loop-nest in Code 1 can be manually derived as follows. The number of iterations of L2 and L3 depend on variable k, which only varies in L1. Thus,

Table 3. Overview of FPGAs

Family	Device	LUTs	FFs	DSPs	BRAMs*
Artix7	xc7a200t	129000	269200	740	730
Kintex7	xc7k480t	597200	597200	1920	1910
Virtex7	xc7v2000t	1221600	2443200	2160	2584
Zynq	xc7z100	277400	554800	2020	1510
Virtexuplus	xcvu13p	1728000	3456000	12288	5376
Kintexu	xcku115	663360	1326720	5520	4320
Zynquplus	xczu19eg	522720	1045440	1968	1968

*17kB per BRAM.

Table 4. Overview of DSPs

Processor	#Cores	L1I	L1D	L2	Simulator	Compiler
Hexagon V60	4	16K	32K	512K	Hexagon SDK	X
Hexagon V5	3	16K	32K	256K	Hexagon SDK	X
TI C6747	1	32K	32K	256K	CCSv4	X

their iteration counts can be expressed as in Equations (7) and (8), respectively.

$$\#L2 = \sum_{i=1}^{m} (m - i) = \frac{1}{2} m(m-1) \tag{7}$$

$$\#L3 = \sum_{i=1}^{m} (m - i)^2 = \frac{1}{6} m(m-1)(2m-1) \tag{8}$$

By using an artificially small input, i.e., small `m`, these equations for all four affected benchmarks have been validated against short co-simulations and found to be exact for the unoptimized loop-nests. When Vivado HLS optimizations are enabled through pragmas such as loop unrolling and pipelining, these equations have to be adjusted accordingly and again verified for small input sizes using co-simulation. These adjusted equations were all exact with the exception of the *gramschmidt* application, where the equations slightly deviated from the measured cycle count.

Aimed at exploring the impact of the amount and the type of resources on flexibility, Xilinx FPGAs from different device families, and with different resources, are included in this study. The Virtexuplus, for example, is an UltraScale+ version of the Virtex FPGA, with many more resources compared to a normal Virtex7. Table 3 lists the details of the selected FPGAs. All simulations were performed with Vivado HLS v2018.2.

- **DSP:** DSPs are an important class of architectures that should be part of this study. Two multi-threaded Hexagon DSPs from Qualcomm, and one single-threaded DSP from **Texas Instruments (TI)** are therefore included. All measurements for these are based on simulations, as measuring on the actual devices was not available. The Hexagon V60 and V5 DSPs are simulated in the cycle-approximate mode provided by the Hexagon SDK [43], and **Code Composer Studio v4 (CCSv4)** [21] provides cycle-accurate simulations for TI C6747. Table 4 provides more details of these three DSPs.

5.1.2 Benchmark Set. Selection of a benchmark set is orthogonal to the definition of flexibility given in this work. It depends on the application domain which benchmarks make sense. Since in this work a generic comparison between platforms is desired, a generic benchmark set is required. A restriction is the support of different platforms, which needs to be broad in this work for comparison between different architectures. Although many options exist, eventually PolyBench/ACC [16]

Table 5. Descriptions of the Applied Benchmarks from PolyBench/ACC

Benchmark	Description
2mm	2 Matrix Multiplications (D=A.B; E=C.D)
3mm	3 Matrix Multiplications (E=A.B; F=C.D; G=E.F)
adi	Alternating Direction Implicit solver
correlation	Correlation Computation
covariance	Covariance Computation
doitgen	Multiresolution analysis kernel (MADNESS)
fdtd-2d	2-D Finite Different Time Domain Kernel
gemm	Matrix-multiply C=alpha.A.B+beta.C
gramschmidt	Gram-Schmidt decomposition
jacobi-1D	1-D Jacobi stencil computation
jacobi-2D	2-D Jacobi stencil computation
lu	LU decomposition
syr2k	Symmetric rank-2k operations
syrk	Symmetric rank-k operations

was selected for its wide support of languages/platforms. Moreover, it is a rather generic benchmark set, i.e., not specific to a specific application domain, and includes programs with static control flow which eases static analysis of the workloads. Furthermore, it provides multi-language versions of benchmarks, including C and CUDA, making it suited for cross-platform evaluation. In this work, 14 applications from this set, supported by multiple languages, are evaluated using their standard dataset. It should be noted that the overall benchmark set contains more applications, but critical issues were encountered for several when targeting the selected platforms. In particular, compilation issues led to blacklisting several applications. Table 5 provides details for each application.

5.1.3 Compiler Directives. It can be argued that not optimizing code gives a distorted image of reality, since programmers typically will spend some effort to manually optimize code for accelerators such as GPUs and FPGAs. As it is unfeasible to hand optimize all benchmarks for each platform, and the code quality would depend heavily on the programmer, a compromise is made by inserting compiler directives. Without directives, some compilers can hardly exploit the maximum potential of the target systems. Therefore, to further investigate the impacts of applying compiler directives on flexibility, compiler directives are inserted in the C-code for CPU, FPGA, and DSP. The GPUs form an exception, as during the transformation to CUDA already manual effort has been made to optimize the code. Extra compiler directives would not change their performance in any significant way. The remainder of this section describes the various compiler directives used for each platform category.

- **CPU:** Multi-threading that parallelizes tasks among multiple threads is enabled by OpenMP directives. The outermost loop is parallelized in each kernel. In case dependencies between loop iterations prohibit parallelization, the (next) inner loop is parallelized.
- **FPGA:** For FPGAs, directives are inserted to refine implementations, aiming at exploiting massive parallelism and increasing resource utilization. By default, Vivado HLS simply translates C functions into Verilog designs. Optimizations are applied rarely without directives. For instance, Vivado HLS does not apply loop unrolling to the C code, meaning that one iteration of the loop is synthesized into a block of logic, which executes sequentially [56]. In this manner, FPGAs cannot exploit their massive parallelism, and merely a diminutive part of resources are utilized. To enable some optimizations without modifying the code-base too

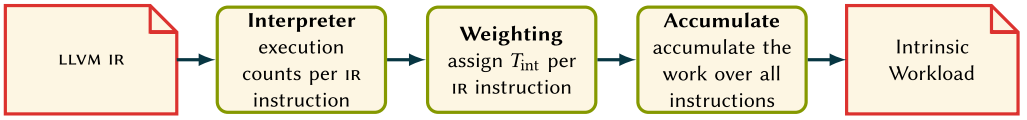


Fig. 5. Automated intrinsic workload estimation using LLVM.

extensively, PIPELINE directives were inserted to promote loop pipelining. When asked to pipeline a loop with an inner loop, Vivado HLS attempts to unroll all inner loops to enable the requested pipelining, increasing resource utilization. If the inner loop spans many iterations, this may lead to a significant increase in required resources. Therefore, pipelining of outer loops is used sparingly, only when the resulting design still fits all target devices, and performance does in fact improve. The default is to pipeline only the inner loop.

- **DSP:** To examine the hypothesis that best-effort techniques, including cache hierarchy and multithreading, negatively impact flexibility, Hexagon DSPs are simulated in two modes: timing accurate and inaccurate mode. With the accurate timing mode, Hexagon models cache, optimal multithreading mode, and processor stalls. With the inaccurate mode, caches are assumed to be perfectly accessed, stalls are excluded, and a simplified multithreading model is simulated [41].

5.1.4 Intrinsic Workload Estimator. As discussed in Section 4, the intrinsic workload of applications is used as the baseline to normalize against. In this section, a practical approach is proposed based on LLVM IR to extract the intrinsic workload from arbitrary applications. An embodiment of the described approach is made available as an open source tool [19] for anyone to use to estimate the intrinsic workload of their application.

In the compilation process that LLVM employs, applications written in diverse languages are translated by front-ends to a high-level intermediate language, IR. General optimization techniques are applied to this IR before generating target-specific code, allowing the reuse of optimization passes across different languages and targets. To serve this purpose the IR instruction set is relatively minimal, and more importantly, platform-independent. Thus, IR is selected as the elementary building blocks for approximating the minimal circuit of an application as detailed in Section 4.

Figure 5 illustrates the procedure to automatically estimate the intrinsic workload of applications. Applications are first translated into LLVM IR by a front-end. Next, the instrumentation code is inserted by a custom LLVM pass called *libDynCountPass*, which triggers a callback for every execution of an IR instruction by the IR interpreter. These callbacks record dynamic instruction counts for each IR instruction type. As loops are fully unrolled in the ideal combinatorial circuit, computations related to control flow and memory accesses are excluded. This is achieved by skipping instrumenting these particular operations in LLVM IR. In particular, a *for* loop represented by IRs is composed of several basic blocks with dedicated names. For example, blocks named *for.cond* and *for.inc* are meant for evaluating the loop condition and increasing the loop counter, respectively. When iterating through basic blocks in functions, the *libDynCountPass* pass skips instrumenting those blocks which names contain *for.cond* and *for.inc*. Furthermore, any parallelism in IR instructions due to vectorization is accounted for by passing the vector width to the instrumentation code. The instrumentation code then adjusts the dynamic instruction counts accordingly.

What remains is weighing all the IR instructions based on their equivalent circuit. To get the size of these circuits, the basic IR operations are mapped to basic 2-input-1-output and 1-input-1-output gates using Cadence Encounter RTL compiler [7]. The number of gates found for different IR instructions is listed in Table 6. Note that the synthesis tool is set up to optimize for area, not

Table 6. Estimated Hardware Cost for Spatial Implementations of Several IR Instructions

IR	32-bit			64-bit		
	Gates	Transistors	Depth	Gates	Transistors	Depth
add/sub[1]	188	880	63	380	1776	127
fadd/fsub	1905	8086	103	3494	14882	233
mul	5164	25458	130	20100	98476	313
fmul	4327	20490	97	16124	77246	215
udiv	4486	10763	1128	18217	44752	4350
sdiv	4777	21300	1189	18840	84616	4458
fdiv	12190	54146	992	66565	303284	2295
urem	4616	20426	1168	18452	82524	4440
srem	4882	21842	1228	19053	85682	4552
and	32	128	1	64	256	1
or	32	128	1	64	256	1
xor	32	256	1	64	512	1

[1] 16-bit *add/sub* operation: #(gate) = 92, #(trans.) = 432, #(depth) = 31;
8-bit *add/sub* operation: #(gate) = 44, #(trans.) = 208, #(depth) = 15;
1-bit *add/sub* operation: #(gate) = 2, #(trans.) = 12, #(depth) = 1.

Table 7. Basic 2 Input — 1 Output Logic Gates and the Number of Transistors Required to Implement Them in CMOS [44, Chapter 6]

Gate	AND	NAND	OR	NOR	XOR	XNOR	INV
Trans.	6	4	6	4	8	8	2

speed. This choice results in a minimal number of gates to achieve certain functionality, rather than a speed-optimized design. In the case of an addition, for example, a carry look-ahead adder would be considerably faster than a ripple adder. For the minimal circuit though, the ripple adder is desired.

Instead of merely counting the number of basic gates per instruction, the argument can be made that in CMOS technology some logic functions are harder to implement than others. An inverter only requires two transistors, where an XOR gate requires eight. To compensate for this, the workload per IR is not only measured in gates, but also in the number of transistors required to realize the circuitry. An approximation of the number of transistors per gate in CMOS technology is listed in Table 7. We refer to the number of these transistors as the intrinsic transistors, or T_{int}, of an application. The intrinsic transistors per application is more precisely defined in Equation (5.1.4). Note that op∈Application in Equation (5.1.4) represents the *dynamically* executed operations.

$$T_{int} = \sum_{(\text{op}\in\text{Application})} \sum_{(\text{gate}\in\text{MinCiruit(op)})} \text{Transistors(gate)} \tag{9}$$

An alternative to expressing circuit complexity in the number of gates or transistors is logic depth of the circuit. The rationale behind this is that the deeper the ideal circuit, the longer the minimum execution time would be. This approximation ignores parallelism inside the operation however, and would likely give rise to the selection of different circuits. For example, a straightforward ripple carry adder has the lowest gate count, but a relatively high logic depth. A carry look ahead circuit has a shallower logic depth, but increases the gate count. Which of these

Table 8. Specifications of Processors in This Work

No.	Processor	Freq. (MHz)	Trans. (M#)	TDP (watt)	Node (nm)	Ref.
1	GTX TITAN	837	7100	250	28	[37][3]
2	GTX 570	732	3000	219	40	[37]
3	GTX 750 TI	1020	1870	60	28	[37]
4	Tegra K1	756	— [1]	14	28	[37][33]
5	Pentium 4	3400	169	84	90	[23]
6	i7-920	2670	731	130	45	[23]
7	i7-950	3070	731	130	45	[23]
8	i7-960	3200	731	130	45	[23]
9	i74770	3400	1400	84	22	[23][2]
10	i76700	3400	1750	64	14	[23][22]
11	ARM1176	700	—	2.9	40	[9]
12	Cortex A9	1700	—	4	32	[1]
13	Cortex A15	2300	—	5	23	[38][32]
14	Cortex A53	1200	—	4.4	40	[9]
15	TI C6747	300	22[2]	0.45[3]	65	[11][50]
16, 19[4]	Hexagon V5	650	—	—	28	[10]
17, 18[4]	Hexagon V60	2000	—	—	14	[42]

[1]"-" means no information available.
[2]Speculated based on TI C66x DSPs [11].
[3]Estimated by TI Power Estimation Spreadsheet 2013.3.
[4]Simulated in inaccurate timing mode as detailed in Section 5.1.3.

approximations to use is debatable. In this work the gate count is used, as it bounds the designs to minimal compute effort for a particular operation. Nonetheless, logic depth could be an interesting alternative when computing the flexibility of high performance computing platforms for example.

5.1.5 Applied Methodologies. In practice benchmarking different systems in terms of performance, energy, and area efficiency is challenging, as it is difficult to unify results across different technology nodes. Furthermore, energy and area numbers are hard to obtain or measure accurately if the platform of interest is physically inaccessible, or not equipped for power measurements. Therefore, in several cases, we had to resort to extrapolation from publicly available data. For example, due to the lack of a power/energy measurement set up for each device, the **thermal design power (TDP)** of platforms is used to estimate their energy usage. The information used per platform and its sources are summarized in Tables 8 and 9.

The following list describes how each metric is obtained for the evaluated systems:

- **Normalized Performance:** Execution time, or cycle counts, can be reliably measured for each of the platforms. The measured times are normalized according to the estimated intrinsic workload of each benchmark, yielding normalized performance with the unit *number of intrinsic transistors per second.*

$$\text{Normalized Performance} = \left(\frac{T_{int}}{t_{exec}}\right) \tag{10}$$

  More intrinsic transistors that can be computed per time unit indicates higher performance. Note that some simulators only provide execution time in cycles, independent of frequency. This does not constitute a problem, since flexibility is invariant to frequency scaling.

Table 9. Specifications of FPGAs used in This Work

Processor	Freq. (MHz)[1]		#Trans.[2]	Pwr (watt)[3]		Node
	no opt.	opt.	(M#)	no opt.	opt.	(nm)
Artix7	116	102	1025	1.4	2	28
Kintex7	120	103	2370	1.7	2.5	28
Virtex7	118	105	9700	2.3	3	28
Zynq	118	96	2200	1.7	2.6	28
Virtexuplus	118	104	14000	4.1	5.5	16
Kintexu	120	101	5300	2.1	3.4	16
Zynquplus	117	103	4100	2.1	2.8	16

[1]Average estimated clock period from Vivado HLS when the target clock period is 10ns.
[2]Speculated based on a published value of Virtex UltraScale XCVU440 [46].
[3]Estimated by Xilinx Power Estimator (XPE) 2018.2.2 based on the average design utilization of benchmark set.

- **Energy Efficiency:** Defined as the quotient of normalized performance and power, normalized energy efficiency is defined as *intrinsic transistors per joule*.

$$\text{Energy Efficiency} = \left(\frac{T_{int}}{t_{exec} \cdot Power}\right) \tag{11}$$

The more intrinsic transistors that can be computed with a joule of energy, the higher energy efficiency.

Energy numbers were estimated based on publicly available data. For most cases this means the reported **thermal design power (TDP)** is used to estimate energy consumption. For multi-core CPUs, this means the same TDP value is used for both single-threaded and multi-threaded mode. As the whole multicore processor is considered as a system, only utilizing a single thread in this system leaves other threads idle, resulting in energy and area overhead.

- **Area Efficiency:** Area efficiency is the quotient of normalized performance by area, expressed in *intrinsic transistors per physical transistor*. Here a 'physical transistor', or T_{phy}, refers to all the transistors implemented on the actual platform under test. This leads to the following definition of area efficiency:

$$\text{Area Efficiency} = \left(\frac{T_{int}}{t_{exec} \cdot T_{phy}}\right) \tag{12}$$

By expressing area as the number of transistors on the device, and not the more typical μm^2, the area efficiency can be expressed independently of technology. Next to this it also allows an alternative interpretation of area efficiency, namely *(physical) transistor utilization*, which provides some insight into how many transistors are utilized effectively towards computing the application.

Note that the transistor count of the FPGAs is for the whole device. When a design uses fewer resources, it could be argued that only the mapped transistors should be included. However, in this work, the full number of transistors is used, as those are physically always present, no matter if an application can utilize them, much like the transistors in any system.

- **Flexibility:** Compute system flexibility is derived relative to performance, energy, and area efficiency. However, due to the way energy and area are measured in this work, these three flexibility values yield the same ranking. In particular, for energy, the same TDP value is used for all applications on a given platform. Hence, the TDP is merely a scaling factor compared

Table 10. Benchmarks for ASIPs Taken from the Work of Fisher et al. [14]

App.	Description
A	FIR symmetrical filter implemented using a 7×7 convolutional kernel.
C	Inverse DCT transform with dequantization of the DCT coefficients. The algorithm used is the Arai, Agui, and Nakjima algorithm for scaled FDCT/IDCT, with some improvements, as described in [4, 52].
D, E	Color conversion from the RGB to the YCbCr color space (and vice versa, as described in the JPEG standard).
F	Halftoning via standard Floyd-Steinberg error diffusion (no stochastic weights update). The benchmark produces triplets containing 1 bit halftoned pixel.
G	1D bilinear scaling by integral factors along columns.
H	3×3 median filter using the standard algorithms not using a "smart" version of the median.
GF	1D bilinear scaling followed by Floyd-Steinberg halftoning.
GEF	1D bilinear scaling followed by E, a YUV ⟵ RG color space conversion, followed by Floyd-Steinberg halftoning.
DH	RGB ⟵ YUV color space conversion followed by a 3×3 median filter.
DHEF	RGB ⟵ YUV color space conversion followed by a 3×3 median filter, followed by E, a YUV ⟵ RGB color space conversion, followed by Floyd-Steinberg halftoning.

The benchmark applications in the lower part of the table are constructed out of the applications in the upper part. Benchmark naming is kept identical with the original work for consistency.

to performance, and flexibility is invariant to constant scaling of data points (in accordance with Lemma A.3). The same holds for the intrinsic transistor count in the area estimates, which again works out to be a constant scaling factor concerning performance. Hence, in this particular case:

$$Flex_{perf} = GSD\left(\frac{T_{int}}{t_{exec}}\right) = GSD\left(\frac{T_{int}}{t_{exec} \cdot Power}\right) = GSD\left(\frac{T_{int}}{t_{exec} \cdot T_{phy}}\right) \tag{13}$$

Notably, this does not hold generically. When energy can be measured accurately across different applications and does vary, the equality no longer holds. Area efficiency flexibility strictly speaking would always be the same as performance flexibility, unless somehow only the active area of a system would be counted. Such a scenario may be meaningful in the context of multi-core systems executing a single thread. In this case, it may be preferable to exclude the inactive cores, resulting in a difference between area efficiency and performance flexibility.

5.2 Customized Processors

Apart from quantifying the flexibility of COTS systems, it is also interesting to investigate how flexibility relates to specialisation. To that end the work of Fisher et al. [14] is used, which reports speedups for a VLIW that is optimized to a varying degree for various target applications. More precisely, a baseline 2 issue slot VLIW processor with a 64 entry registerfile is constructed with one ALU capable of integer multiplication, and one issue slot for accessing L1 or L2 memory. Starting from this baseline, an architecture exploration extending the architecture in several aspects such as ALU and multiplier count, memory ports, and register file size, is performed targeting each of the benchmark applications listed in Table 10. This exploration is performed under a 'computing architecture cost' constraint, which is a custom metric defined in Section 3.3 of the work of Fisher

et al. [14]. This cost metric depends amongst other parameters on the number of ALUs, width of datapaths, and number of ports to the register file. Within a given cost constraint, the VLIW architecture is first optimised targeting each individual application. For the ten benchmarks listed in Table 10, this yields ten optimised architectures. Each of these architectures are still capable of executing the complete benchmark set, and as such a speedup can be reported for each application on each architecture. These speed up measurements are reported in Tables 8, 9, and 10 in the original paper [14]. Using these speedups, the flexibility of each resulting architecture can be computed, with the notable exception that normalisation is not based on intrinsic workload, but rather the single ALU baseline machine as described earlier.

Apart from just optimising for each application individually, Fisher et al. propose optimisation with a certain 'range'. When this range is set to $X\%$, it means the optimisation algorithm is allowed to sacrifice $X\%$ of the maximum performance of the application currently being optimized for, and trade that off for an improvement in the overall performance of the architecture over the complete benchmark set. For example, when the range is set to 0%, the processor is allowed to have a cost of 10 according to the defined cost metric, and the optimization target is benchmark GEF, the maximum speedup for GEF is 8.93×. The harmonic mean performance of that tuned architecture over the entire benchmark set then equals 3.9×. However, when the range is set to 50%, the optimisation procedure accepts a maximum penalty on the speedup of application GEF reducing it to 5.97×, in order to improve the harmonic mean performance over the complete benchmark set to 5.8×. These results can be verified in Table 9 of the original paper [14].

Using the reported speedups, the relation between this range parameter and flexibility can be investigated. The results and analysis of this experiment are presented in Section 6.2.

6 RESULTS AND ANALYSIS

The experiments are split into two categories. First, flexibility measurements are performed for commercial of the shelf systems in Section 6.1. Relations between flexibility and performance, energy efficiency, and area are investigated as part of this experiment. Furthermore, the relations between flexibility and several architecture classes is examined. Secondly, Section 6.2 analyses the relation between flexibility and processor specialisation, inspired by the work of Fisher et al. [14].

6.1 Commercial Off the Shelf Processors

This section presents the flexibility measurements of 25 platforms over 14 benchmarks, and investigates the hypothetical relations between flexibility, performance, energy efficiency, and area efficiency. In particular, Figures 6, 7, and 8, respectively represent the relations between flexibility and these metrics. The indexing in the figures corresponds to the platform numbering in Table 8. To achieve a fair comparison, performance results are scaled to the technology node of each platform. Accurate technology scaling is a topic on it's own, and different techniques should be used for different devices. For example, memories, wires, and gates all scale differently. To keep the comparison in this work straightforward, the basic assumption is used that performance, i.e., gate delay, scales linearly with the inverse of the technology node. Scaling under this assumption is sufficient to observe overall trends, and draw preliminary conclusions on the relation of the defined flexibility metric with respect to performance. Note that this technology scaling is an issue that arises in our particular measurement due to the lack of a set of platforms in the same technology node, and is orthogonal to the definition of flexibility itself. Any comparison of performance between architectures instantiated on different technology nodes requires such scaling.

Note that since performance, energy efficiency and area efficiency flexibilities are all equal for our measurements according to Equation (13), the flexibility rankings do not move horizontally between Figures 6, 7, and 8.

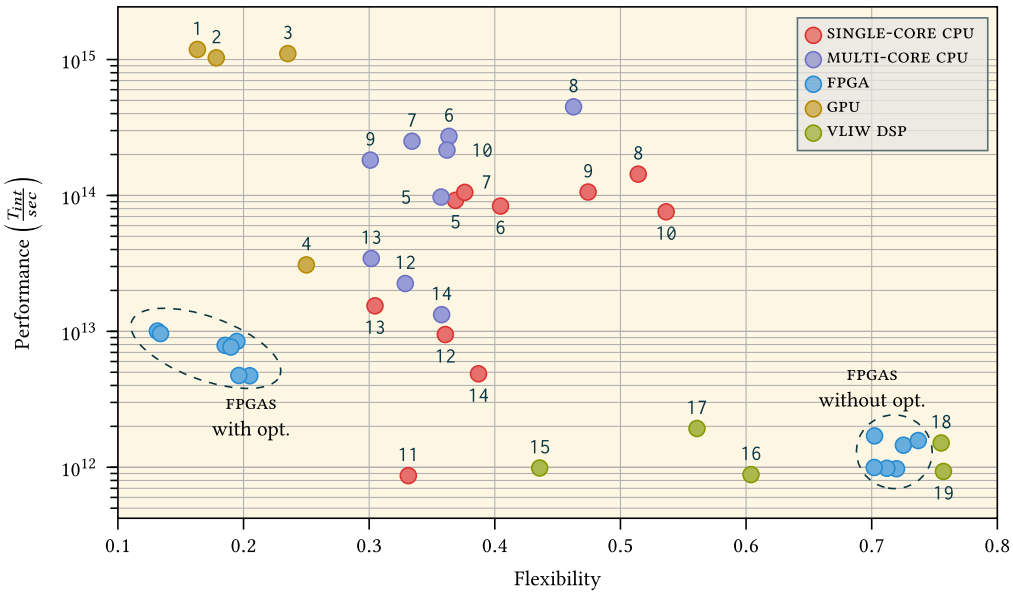


Fig. 6. Performance and Flexibility. Platform indexes according to Table 8.

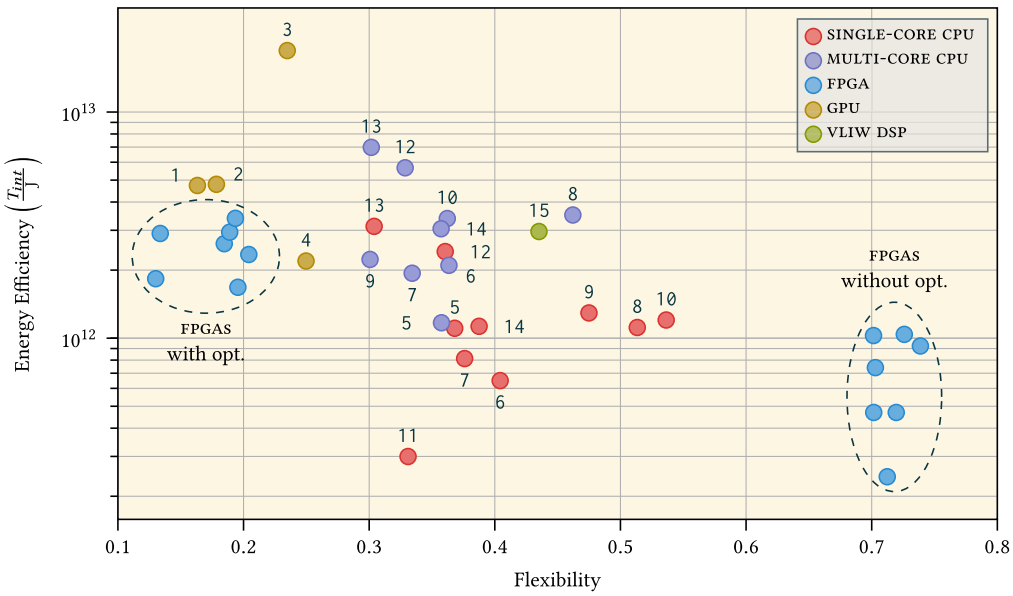


Fig. 7. Energy efficiency and flexibility. Platform indexes according to Table 8.

Before exploring the relations between flexibility and other metrics, it is interesting to note that the flexibility ranking has the power to discriminate between different architecture classes. From our measurements we make the following six observations:

(1) The GPUs are clearly the least flexible of the tested architectures, while FPGAs without optimization are highly flexible. This does align with the idea that GPUs are specialized

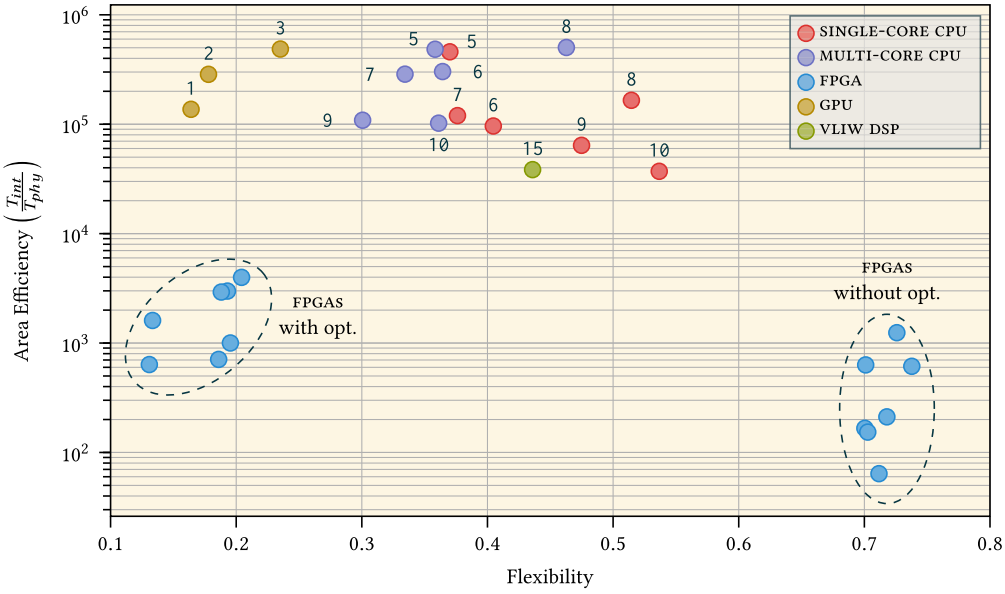


Fig. 8. Area efficiency and flexibility. Platform indexes according to Table 8.

devices, supporting only a specific subset of algorithms (with DLP) very well, while FPGAs on the other hand are generic devices.

(2) When optimization is turned on for FPGAs however, clearly some applications benefit more than others because of the extra resources, but without optimization their flexibility is far superior to other architectures.

(3) Furthermore, multi-core CPUs measure as slightly less flexible than single-core CPUs, which again is intuitive as not all applications will benefit equally from extra cores. In general, it seems that for this generic benchmark set, architectures that employ more parallel execution pay a penalty in flexibility.

(4) Predictably though, the same parallel architectures also have the highest performance, as can be seen in Figure 6. This confirms the general notion that there is a trade-off between performance and flexibility. Although there are definite outliers, overall higher flexibility implies lower performance. The ideal point in Figure 6 is at the top right, combining high performance with high flexibility. The points closest to that corner are the general-purpose CPUs, showing these devices cover the middle-ground in this trade-off as would be expected.

(5) For energy efficiency (Figure 7), a similar trend can be observed, although much less pronounced. In particular, unoptimized FPGA as the most flexible platforms have less of a gap to the CPUs in energy efficiency than they have in performance. A plausible explanation can be found in the much lower clock frequency of the flexible FPGA fabric, which obviously incurs a penalty in performance, but does not necessarily translate to low energy efficiency. Because the FPGAs essentially execute highly customized/parallel instructions, they may perform more useful work per cycle, leading to less register/memory overhead. The fact that CPUs execute their more generic, yet simple, operations much faster gives them a definitive edge in performance, however, the extra required cycles give them a relative handicap in energy efficiency. This shortage is overcome by the optimized FPGAs, which can customize their operations to achieve more work per cycle bringing them on par with the bulk of CPUs in terms of energy efficiency, even though their performance is lower.

(6) In area efficiency (Figure 8), the results are far less conclusive. The only outliers are the FPGAs, which are in a class of their own. This is to be expected, as the flexible FPGA fabric requires not only large silicon area for the LUTs but also routing, which makes it is very complex compared to other architectures. Between the other systems, the area efficiency numbers do seem to drop off slightly with increasing flexibility, but the results are too close to draw any significant conclusions at this stage.

In general, the results show the proposed flexibility metric can distinguish between various architecture classes, indicating it represents a fundamental property. The measurements also align with some generally accepted notions surrounding flexibility, such as the trade-off between performance and flexibility. Furthermore, it is interesting to see there is a gap between the fairly flexible FPGAs and the single-core CPUs, which indicates room for exploration and possibly exploration. For many years, **coarse grain reconfigurable architectures (CGRAs)** have claimed to combine the benefits of flexible FPGAs with the benefits of fixed-functionality hardware [53]. Introducing CGRAs for flexibility comparison would be very interesting.

6.2 Customized Processors

In this experiment, the relation between flexibility and the measure of customisation/optimisation of an **application specific processor (ASIP)** is investigated. In particular, the relations between flexibility, *computing architecture cost*, and *range* as defined in Section 5.2 are of interest. This evaluation is based on the reported speedups in Tables 8, 9, and 10 in the work of Fisher et al. [14]. Using these speedups, the flexibility of each architecture is computed, the results of which can be found in Tables 11 and 12 in this work. Each row in these tables shows the speedups of all individual applications compared to the baseline VLIW processor, when the architecture is optimized for a particular application. The last two columns respectively give the harmonic mean speedup which Fisher et al. used as the cost function in their architecture search algorithm, and the flexibility as defined in this work. Note that when the 'range' parameter is set to infinite, it does not matter what the optimisation target application is, the optimisation algorithm will find only one architecture. After all, if it is allowed to compromise the speedup of the target optimization by an infinite amount, it does not matter what the target application is. The architecture with the best harmonic mean speedup over all applications will be selected, regardless.

From the results in Tables 11 and 12, three key observations are made:

(1) In general, the high cost architectures achieve better generalisation than the low cost architectures. With plenty of compute resources available the benefits for each application average out, while with low compute cost only specific parts of all the applications benefit, leading to a more unbalanced speed up over the entire benchmark set.
(2) For the architectures with cost ≤ 5, the flexibilities of the resulting architectures are almost all identical to the architecture found with range ∞. In fact, for most applications the selected architecture is actually equal, as can be seen in Table 9 of the original paper by Fisher et al. [14]. A clear outlier is application A, which when set as the optimization target actually yields a very high flexibility. As can be seen in the table, the speedup of A is rather significant, while the other benchmarks appear to benefit fairly equally. The conclusion must be that application A is quite different from the other benchmarks in the set, and optimising for it within low cost constraint does not benefit the other applications. In the original paper of Fisher et al. [14] it can be seen that the number of registers for architecture A is higher than the other architectures, suggesting that application A benefits heavily from more registers, while that does not help the other benchmarks that much.
(3) A higher range interestingly does not always result in a more flexible architecture, as is the case for application G at cost ≤ 15 in Table 12, for example. The sacrificed performance of

Table 11. Speedup and Flexibility of ASIP Architectures with Cost ≤ 5, Based on Fisher et al. [14]

Arch.	Speedup for application *X*										HMean	Flex.
	A	C	D	F	G	H	GF	GEF	DH	DHEF		
Cost 5 — Range 0%												
A	**6.12**	3.60	3.52	3.54	3.43	3.58	3.53	3.52	3.56	3.60	3.7	0.848
C	1.05	**3.93**	4.09	6.00	5.72	6.35	6.16	5.86	6.33	6.38	3.8	0.590
D	1.05	3.93	**4.09**	6.00	5.72	6.35	6.16	5.86	6.33	6.38	3.8	0.590
F	1.05	3.93	4.09	**6.00**	5.72	6.35	6.16	5.86	6.33	6.38	3.8	0.590
G	1.05	3.93	4.09	6.00	**5.72**	6.35	6.16	5.86	6.33	6.38	3.8	0.590
H	1.05	3.93	4.09	6.00	5.72	**6.35**	6.16	5.86	6.33	6.38	3.8	0.590
GF	1.05	3.93	4.09	6.00	5.72	6.35	**6.16**	5.86	6.33	6.38	3.8	0.590
GEF	1.04	3.93	4.09	4.53	5.72	6.15	6.14	**5.97**	6.31	6.36	3.8	0.593
DH	1.05	3.93	4.09	6.00	5.72	6.35	6.16	5.86	**6.33**	6.38	3.8	0.590
DHEF	1.05	3.93	4.09	6.00	5.72	6.35	6.16	5.86	6.33	**6.38**	3.8	0.590
Cost 5 — Range 10%												
A	**6.12**	3.60	3.52	3.54	3.43	3.58	3.53	3.52	3.56	3.60	3.7	0.848
C	1.05	**3.93**	4.09	6.00	5.72	6.35	6.16	5.86	6.33	6.38	3.8	0.590
D	1.05	3.93	**4.09**	6.00	5.72	6.35	6.16	5.86	6.33	6.38	3.8	0.590
F	1.05	3.93	4.09	**6.00**	5.72	6.35	6.16	5.86	6.33	6.38	3.8	0.590
G	1.05	3.93	4.09	6.00	**5.72**	6.35	6.16	5.86	6.33	6.38	3.8	0.590
H	1.05	3.93	4.09	6.00	5.72	**6.35**	6.16	5.86	6.33	6.38	3.8	0.590
GF	1.05	3.93	4.09	6.00	5.72	6.35	**6.16**	5.86	6.33	6.38	3.8	0.590
GEF	1.05	3.93	4.09	6.00	5.72	6.35	6.16	**5.86**	6.33	6.38	3.8	0.590
DH	1.05	3.93	4.09	6.00	5.72	6.35	6.16	5.86	**6.33**	6.38	3.8	0.590
DHEF	1.05	3.93	4.09	6.00	5.72	6.35	6.16	5.86	6.33	**6.38**	3.8	0.590
Cost 5 — Range ∞%												
All	**1.05**	**3.93**	**4.09**	**6.00**	**5.72**	**6.35**	**6.16**	**5.86**	**6.33**	**6.38**	3.8	0.590

G when increasing the range from 0% to 10% does lead to a higher harmonic mean performance, but the variation in speed ups increases. This again demonstrates that performance and flexibility are orthogonal properties, and while Fisher et al. optimised for overall performance, the flexibility decreased. If the goal of an architect is to design a processor that is likely to perform well under varying applications, the optimisation goal should thus have been flexibility and not overall performance.

Note that observation 1 is supported by the findings of Arnold and Corporaal [5], who investigate the benefit of adding custom instructions to a processor that replace two basic operations. Figure 9 is taken from their work, and shows the reduced operation count by adding a library of size x with more complex operation patterns of size two, i.e., replacing two operations in the original execution graph. The theoretical best case is an operation count reduction of 50%. As can be seen in the figure, when the number of added complex patterns is on the low end, e.g., 10 patterns, the vertical spread is relatively high. A high vertical spread equals low flexibility, since there is large variation between different applications. To validate this, the operation reductions for 10 and 40 patterns were extracted from the image[2], and summarized in Table 13. It shows that

[2]The original work [5] does not list the raw numbers, but a vector image of the graph could be recovered from the pdf file which allowed accurate reconstruction of the measurements.

Table 12. Speedup and Flexibility of ASIP Architectures with cost ≤ 15, based on Fisher et al. [14]

Arch.	Speedup for application *X*										HMean	Flex.
	A	C	D	F	G	H	GF	GEF	DH	DHEF		
Cost 15 — Range 0%												
A	**13.06**	5.88	3.52	5.63	4.95	9.68	8.13	8.65	9.60	9.14	6.8	0.690
C	11.04	**7.46**	3.86	5.25	5.41	10.50	8.39	8.93	10.61	9.88	7.2	0.710
D	10.72	6.07	**4.42**	6.13	5.42	6.35	6.16	5.86	6.31	6.38	6.1	0.811
F	10.72	6.07	4.42	**6.13**	5.42	6.35	6.16	5.86	6.31	6.38	6.1	0.811
G	9.38	6.15	4.33	6.13	**5.72**	6.35	6.16	5.86	6.33	6.38	6.1	0.838
H	5.95	7.46	3.86	3.98	5.41	**10.52**	5.75	6.79	10.58	9.74	6.2	0.705
GF	11.04	7.46	3.86	5.25	5.41	10.50	**8.39**	8.93	10.61	9.88	7.2	0.710
GEF	11.04	7.46	3.86	5.25	5.41	10.50	8.39	**8.93**	10.61	9.88	7.2	0.710
DH	11.04	7.46	3.86	5.25	5.41	10.50	8.39	8.93	**10.61**	9.88	7.2	0.710
DHEF	10.54	6.43	3.86	5.25	5.41	10.50	8.39	8.93	10.55	**10.06**	7.1	0.710
Cost 15 — Range 10%												
A	**13.06**	5.88	3.52	5.63	4.95	9.68	8.13	8.65	9.60	9.14	6.8	0.690
C	11.04	**7.46**	3.86	5.25	5.41	10.50	8.39	8.93	10.61	9.88	7.2	0.710
D	10.72	6.07	**4.42**	6.13	5.42	6.35	6.16	5.86	6.31	6.38	6.1	0.811
F	13.06	5.88	3.52	**5.63**	4.95	9.68	8.13	8.65	9.60	9.14	6.8	0.690
G	11.04	7.46	3.86	5.25	**5.41**	10.50	8.39	8.93	10.61	9.88	7.2	0.710
H	11.04	7.46	3.86	5.25	5.41	**10.50**	8.39	8.93	10.61	9.88	7.2	0.710
GF	11.04	7.46	3.86	5.25	5.41	10.50	**8.39**	8.93	10.61	9.88	7.2	0.710
GEF	11.04	7.46	3.86	5.25	5.41	10.50	8.39	**8.93**	10.61	9.88	7.2	0.710
DH	11.04	7.46	3.86	5.25	5.41	10.50	8.39	8.93	**10.61**	9.88	7.2	0.710
DHEF	11.04	7.46	3.86	5.25	5.41	10.50	8.39	8.93	10.61	**9.88**	7.2	0.710
Cost 15 — Range ∞%												
All	**11.04**	**7.46**	**3.86**	**5.25**	**5.41**	**10.50**	**8.39**	**8.93**	**10.61**	**9.88**	7.2	0.710

Table 13. Operation Count Reductions from Figure 9, and Associated Flexibility

Top X	Application											Flex.
	bspline	dft	pse	iir	foewf	fir	flatten	smooth	expand	compress	edge	
10	45.8	31.1	44.1	34.6	22.0	42.2	32.6	38.5	32.5	31.6	38.5	0.82
40	45.8	44.5	45.6	39.3	39.1	42.1	42.7	42.4	42.4	43.4	41.5	0.95

overall flexibility increases from 0.83 to 0.95 when adding 30 extra patterns. This corresponds to the cost ≤ 5 architectures from Fisher et al. with relatively limited resources, and low flexibility. When the number of available resources increases however, such as the cost ≤ 15 architectures or 30+ patterns, the flexibility increases. With more resources available, there is more room to have every application profit maximally.

7 COMPARISON WITH EXISTING DEFINITIONS

This section compares the proposed flexibility metric with related work that provide alternative definitions. In particular, the proposed method is compared against versatility as defined by Van Berkel [26], and 'VersaBench versatility' as defined by Rabbah et al. [45], in Sections 7.1 and 7.2, respectively.

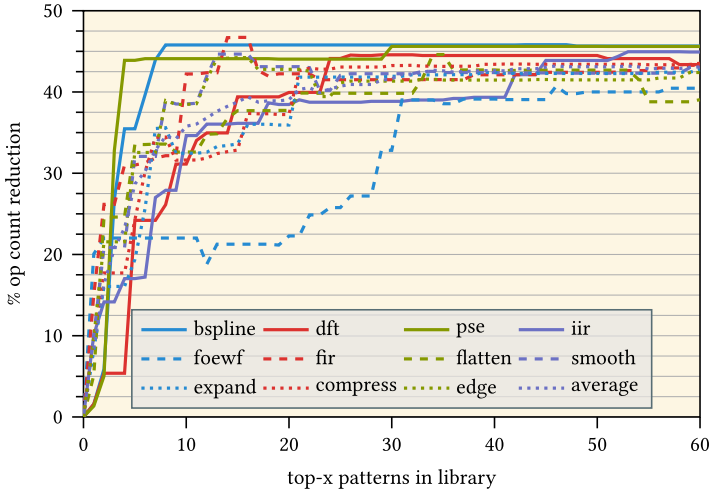


Fig. 9. Operation count reduction by library with x patterns [5].

7.1 Flexibility and Versatility

In the search for alternative definitions of flexibility in Section 2, the qualitative and quantitative definitions of *versatility* provided by Van Berkel [26] appeared to be most related. Rather than looking at change of a system's performance metrics under the influence of external changes, the assumption is made that the less information required to specify an amount of work to a system, the less versatile it must be. In terms of computing systems this translates to the number of dynamic (instruction) bits required to execute a given program on a given processor. The fewer bits required, the fewer options were available to select functionality from, hence the less versatile the system must be. This is captured in the versatility formula in Equation (1).

This interesting approach does not directly measure the effects of external changes on the system. Rather, it makes the underlying assumption that more functionality to select from *should* result in a more flexible system. After all, if there are more options/instructions, it is easier to adapt to a new program. The danger in this assumption is that the computer architect now has become part of the metric, i.e., it is assumed that the extra added functionality is diverse enough to handle more cases. In a way, the number of dynamic bits per workload is a measure of both how flexible the system is, and how well the architect anticipated and addressed possible changes to the system.

Therefore, we still reason the qualitative definition provided at the start of this section is more suited for defining a flexibility metric, and versatility is in fact different yet very related. In particular, the difference between measured versatility and flexibility is an indication of how well a system architect has designed the system. When a system is not flexible, but highly versatile, apparently a price is paid for having more options/functionality, but it did not translate into added flexibility. In fact, the ratio between flexibility and versatility can be regarded as a measure of success of an architect to balance the cost of instruction size with return in flexibility.

Given this conclusion, it is interesting to compare flexibility and versatility for various architectures. Application of the versatility metric is slightly more involved than the proposed flexibility metric, however, for two reasons:

(1) Versatility is only defined per application, and as such for different applications one architecture would have multiple flexibilities. To be able to compare, we therefore propose to use the

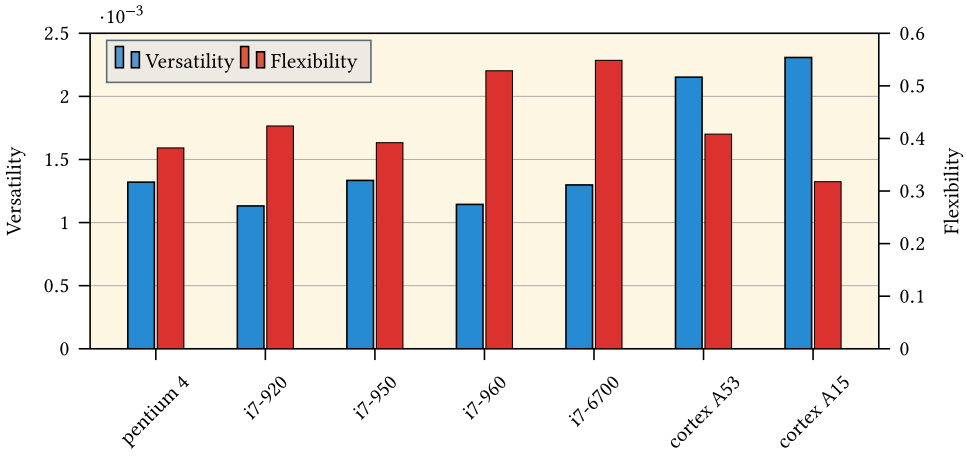


Fig. 10. Flexibility and versatility comparison between Intel and ARM CPUs.

geometric mean of all versatilities of a benchmark set to obtain a single flexibility number per architecture.

(2) The definition of versatility includes the number of *useful operations*. This term is not defined exactly by Van Berkel. In this analysis we will therefore use the proposed intrinsic workload instead, which also should yield a fair comparison between versatility and flexibility.

The resulting definition of Versatility as used in this comparison is given in Equation (14), where x_i is an application in benchmark vector $\vec{X}$, and $W_{int}(x_i)$ is its intrinsic workload.

$$Versatility(\vec{X}) = \left(\prod_{i=1}^{n} \frac{avg\_instruction\_size(x_i)}{W_{int}(x_i)/\#instructions(x_i)} \right)^{\frac{1}{n}} \tag{14}$$

Finding the number of instructions on Intel and ARM machines is straightforward by merit of the available performance counters. The average instruction size for the considered ARM devices is also simple, a fixed 32 bits. The instruction size of the Intel machines is dynamic on the other hand, and not monitored by hardware performance counters. Therefore, an estimate of 20 bits per instruction on average is used for the Intel machines as per the work of Ibrahim et al. [20].

Figure 10 shows a side-by-side comparison of performance flexibility with versatility for several Intel and ARM CPUs. The results are quite interesting, and two observations can be made.

(1) First, the flexibility of the Intel machines increases with newer generations, while the versatility is more or less equal. This implies that the designs improve in such a way that without spending more instruction bits, the Intel processors have become more flexible.

(2) Second, the ARM processors have a much higher versatility than the Intel processors, yet fail to capitalize on this in particular compared to the later generations of Intel processors.

This outcome seems to make a case for variable instruction width as opposed to fixed.

However, not covered here is the use of the 16 bit thumb instruction set of ARM, which may paint a different picture. Such investigations are left as future work, but it can clearly be concluded that by having quantitative metrics insight can be gained, and processor design can potentially be guided by such metrics to truly get to machines that balance average instruction size, performance, energy, and flexibility.

7.2 Flexibility and VersaBench Versatility

Apart from versatility as defined by Van Berkel [26], there is a variant defined by Rabbah et al. [45] as discussed briefly in Section 2.2. As the motivation behind this second definition is also close to the goals of this paper, this section provides a short comparison between flexibility as defined in this work, and this "VersaBench Versatility".

To understand the differences between VersaBench Versatility and flexibility, it makes sense to look at the properties of flexibility as defined in Section 3.2.1. In particular, we state that flexibility should be orthogonal to performance. The motivation behind this property is illustrated in Figure 2, where the most performant system shows a higher variation to change than the slowest system.

VersaBench Versatility on the other hand normalizes performance based on the fastest processor known for each application, and uses this to rank processors. Therefore, performance and VersaBench versatility are directly related. In fact, it can be shown that the normalisation by the fastest processor for each application is irrelevant for the final versatility **ranking**. Lemmas A.4 and A.6 show that the normalisation baseline is cancelled out when calculating the ratio of two positive datasets as shown in Equation (15), where $\vec{X}$, $\vec{Y}$ and $\vec{B}$ are positive datasets.

$$\frac{GM\left(\frac{\vec{X}}{\vec{B}}\right)}{GM\left(\frac{\vec{Y}}{\vec{B}}\right)} = \frac{GM(\vec{X})}{GM(\vec{Y})}, \tag{15}$$

$$\frac{GSD\left(\frac{\vec{X}}{\vec{B}}\right)}{GSD\left(\frac{\vec{Y}}{\vec{B}}\right)} \not\equiv \frac{GSD(\vec{X})}{GSD(\vec{Y})}, \tag{16}$$

Hence, the ranking obtained using the metric proposed by Rabbah et al. is equivalent to ranking on average performance. For completeness, flexibility as defined in this work does depend on the chosen baseline, as shown in Equation (16), supported by Lemma A.8: $GSD(\vec{X}/\vec{B}) \geq {}^{GSD(\vec{X})}/_{GSD(\vec{B})}$. Note that '$\not\equiv$' here denotes that the relation does not necessarily hold. In conclusion, VersaBench versatility is in fact a ranking based on average performance, while flexibility is truly an orthogonal property.

8 DISCUSSION & OPEN ISSUES

The work presented in this document is an attempt at defining a flexibility metric for processors. The lack of ground truth, however, combined with several existing related studies presented in Section 2, entails that the resulting definition is to be placed into context for it to carry any meaning. The definition presented in this work is not to be taken as final, as many open questions underlie its definition to which there are no definitive answers yet. This section discusses these open issues, and how they were handled in this particular work.

8.1 From Qualitative to Quantitative

To systematically identify open issues and the choices made when deriving a quantitatively metric form the qualitative definition of flexibility, we identify three key components in the qualitative definition of flexibility as given in Section 3.1:

(1) The measured system and its set of (changing) external inputs (S_i)
(2) Observed Performance Metric (m)
(3) "Measure of Affectedness" ($f(S_i, m)$)

Each of these components is to be mapped to computing systems to properly define computing system flexibility. However, each component leaves room for different interpretations, which is the root cause of different definitions of processor flexibility in related work. What follows is an attempt to capture different interpretations of these terms, and motivate the choices made in this work.

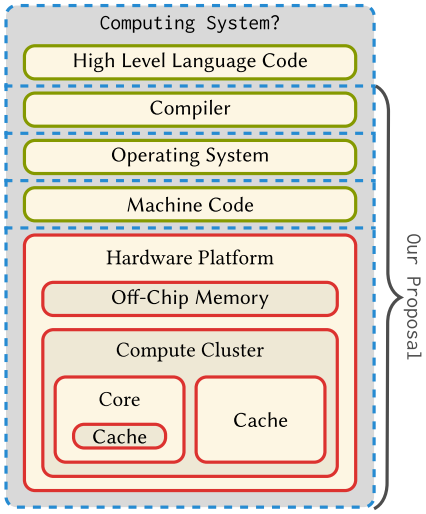


Fig. 11. Where to draw the line, what is part of a compute system, and what is not?

(1) **System and External Inputs**
Defining what the measured system and its (changing) inputs, S_i, seems trivial, but turns out to be both complex and very relevant for the resulting definition. For example, if the system is defined as a bare processing core, then the system's external inputs would be machine instructions and data. Changes in the instruction and data stream influence the energy usage of the core in different ways, and flexibility may measure the sensitivity of this energy usage for different instruction streams on a per cycle basis. However, if the system includes not only the core but also an instruction cache, the fine grain energy consumption may already vary without new instructions coming in from *outside* the system. In this case, some forms of temporal integration have to be used, and flexibility measures the effects of new blocks of instructions loaded on this averaged energy usage. The key message is that the cache can be part of the system, and the cache size can also influence the system's flexibility in this case. In fact, many components in the chain from application idea in the mind of a developer, up to final execution on a core, can influence the system's performance metrics. Therefore, each of these components *could* be seen as part of the computing system, as illustrated in Figure 11. Although not shown in the figure, in an extreme case the programmer who writes a program according to changing specifications could even be considered part of the system. The supported language, compiler, memory hierarchy, and processor architecture then all influence the measured system. Note that the world is much larger than what is captured in Figure 11. For example, loop-buffers, or configuration memory in an FPGA could all be different points to draw a system boundary.

Where the system boundaries are defined is rather arbitrary from this viewpoint, although it would be sensible to not make an individual programmer part of the system. Yet, it is preferable to stay at a higher level, as the lower levels quickly become more specific for a certain subclass of systems, e.g., configuration memory for FPGAs. Therefore, we have chosen to stop at the compiler level, where the compiler is still considered part of the system, and the source code is the external input. In particular, the source code of different applications, which is also the external input used for most commonly used processor benchmark suites. In these benchmarks, the applications are taken as the changing input, hence this best

practice is followed in this work. This choice also aligns with the view of Zse et al. [48], who consider the mapper (compiler) part of the system.

(2) **Observed Performance Metrics**
There exist many widely used metrics in computer system design, such as energy-efficiency, area-efficiency, and runtime. In principle, any of these or even a combination, can be selected as the observed performance metric. Related work typically ties flexibility exclusively to runtime, e.g., the work of Rabbah et al. [45]. We argue this is too restrictive, and a plurality of meaningful flexibilities can be defined. Zse et al. [48] already hint at flexibility in terms of performance and energy efficiency, which is more in line with the reasoning of our work. In particular, it is worth noting that flexibility in our view as such is a derived metric, since it measures changes in other primary metrics. For any primary metric, flexibility can be defined.

(3) **"Measure of Affectedness"**
Whereas for the other two terms, there is a history of common practices in computer system design to build upon, it is the "measure of affectedness" ($f(S_i, m)$) about which there is the least consensus in the community. There are numerous ways to quantify changes in a metric m. If the flexibility function f is to be generic for any metric m however, it is clear the changes in m caused by changing inputs have to be normalized. The selection of a normalization method is another degree of freedom, which is part of defining f.

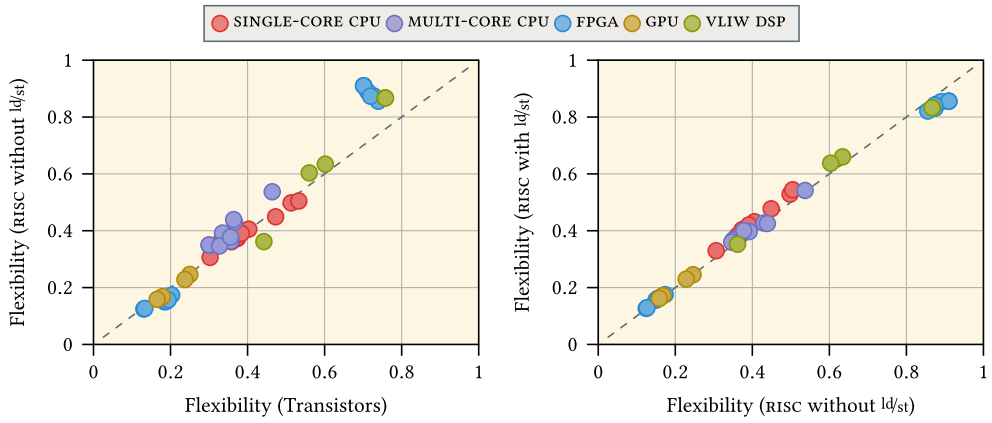
In the work of Rabbah et al. [45], the runtime is normalized by comparing the runtime of applications to the best-known runtime over all processors. Change in the runtime of a processor compared to this "optimal runtime" is seen as inflexible. The total change over a selected benchmark set is then seen as the flexibility of a processor. Thus, higher absolute performance over the benchmark set is taken to mean higher flexibility. Although this direct coupling of performance and flexibility may seem appealing, in particular for a designer who needs to design or select a system with flexibility as a metric, we argue that such reasoning is a fallacy. The implication that the most performant machine automatically is the most flexible machine is unfounded, and the selection as the best-known runtime for each application is an arbitrary baseline for a flexible machine.

Instead of normalizing against the best known runtime, we postulate there must be a notion of intrinsic workload for each application that can be normalized against. The definition of this intrinsic workload again poses several challenges, which is further discussed in Section 8.2.

Finally, we argue that *any* change in intrinsic workload normalized metric m, either positive or negative, makes a system less flexible. As a measure of how affected a metric m is under a set of changes, *variance* seems a natural choice to us.

8.2 Intrinsic Workload

In Section 4, the term *intrinsic workload* is introduced, which refers to the notation that an application inherently describes a fixed amount of work. An open question is, assuming the notion of such a fixed amount of work per application is correct, how to properly define this intrinsic workload. There are many possibilities, and selecting one that is both theoretically and practically appealing is a difficult task. A fundamental approach may consider Landauer's principle [30], which states there is a minimum amount of energy that is dissipated when a bit of information is erased. For a typical irreversible computation that consumes two operands and produces one output value, this principle can be used to compute a minimum amount of work for that operation. However, if the computation is reversible, no information is lost in the system, and theoretically no energy would be required to perform such a computation. Thus, if an application is expressed in reversible operations, it may not have an intrinsic workload at all, and computation may in fact be free. This is the



(a) Transistor based vs RISC operation based flexibility

(b) Impact of accounting for load/stores

Fig. 12. Effect of different normalisation strategies.

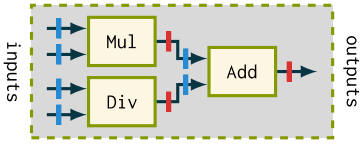
promise of the field of reversible computing, and maybe the only fundamentally correct answer to the question of how much workload any given application represents, zero.

This definition of (the non-existence of) intrinsic workload from a physics perspective does not provide any insight for the practical machines in current technology however, so for practical reasons, a more pragmatic approach is taken in this paper. As extensively described in Section 4, an approach is chosen which expresses workload in terms of the size of the minimal circuit that implements an application. The motivation for this approach is that it automatically weights operations based on their complexity, and a multiplier circuit will require more transistors than an adder.

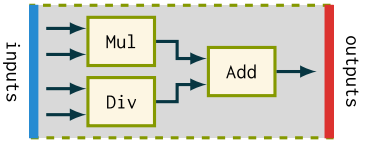
A downside of this choice is the infeasibility to construct the schematic of a truly minimal circuit for any application. Such a circuit would be extremely large, and logic minimization is proven to be NP-complete [6]. As discussed in Section 5.1.4, the practical choice was made to approximate the size of the ideal minimized circuit by dividing applications into LLVM IR instructions and weighting those based on their approximated minimal circuits. This choice is very much motivated by the desire to develop a flexibility metric that is also applicable in the real world and not just a theoretical notion. In particular, this approximation may be done in various different ways, and remains an open topic of research.

8.2.1 RISC vs Transistors. In Section 4, the choice is made to express workload in intrinsic transistors. It can be questioned, though, how much this refinement of RISC-like operations to intrinsic transistors impacts the resulting flexibility measure. To investigate this, Figure 12(a) plots flexibility based on intrinsic transistors (horizontally) versus flexibility based on RISC-like operations. The flexibility based on RISC-like operations is calculated similarly to the proposed transistor-based flexibility, except that all operations in Table 6 are set to one. If both flexibility metrics are exactly the same, the points in Figure 12(a) would be on the diagonal of the plot. Diversion of points from the diagonal indicates differences between the two metrics.

As can be seen in the figure, the majority of the points are close to the diagonal, indicating the refinement towards transistors does not change the flexibility significantly. Only the most flexible points, which represent the unoptimized FPGAs, seem to be classified as significantly more flexible when normalized to RISC instructions rather than transistors. A possible explanation for this phenomenon is that the FPGA uses its DSP slices to perform the multiplications, relatively



(a) Arithmetic operations + internal memory operations (2 loads and 1 store for each operation)



(b) Arithmetic operations + external memory loads and stores

Fig. 13. Two conceptual combinatorial-circuit models with loads (grey) and stores (blue).

lowering their performance complexity compared to other operations. As such, weighting the multiplications as more work based on the circuit complexity may expose some inflexibility of the unoptimized FPGA solutions. For the RISC baseline this skewing is not present, and hence the solutions are quantified as more flexible. On the optimized FPGA designs, higher degrees of parallelism may hide this effect. It is, however, difficult to reason about such effects. Nonetheless, it can be concluded that although the refinement into transistors is from the theoretical viewpoint arguably "more correct" than not accounting for operation complexity, omission of this refinement for practical considerations would not have a large impact on the measured flexibility.

8.2.2 Loads and Stores. Although the use of an approximate minimal circuit makes the definition of intrinsic workload practical for real machines, one problem mentioned at the start of this section still remains. The minimal circuit of a matrix transpose algorithm would not involve any transistors, and would consist only of wires, i.e., loads and stores do not exist. Although this may be a fair game from the theoretical point of view, the retrieval or storage of information should not need to cost any energy, for practical purposes it may not be the most workable approximation. An alternative that is possible within the proposed intrinsic workload estimation framework is to weight IR based loads and stores.

How these loads and stores are to be weighted is again a point of discussion. One additional practical consideration could be to weight external and internal loads differently, since external memory accesses are typically much more expensive than internal memory accesses. A very crude way of separating the two would be to count an internal load for each input operand of an operation, and a store for each output operand. External loads are then defined by the input to an application that has to be loaded once, and the output produced by the application which has to be stored once externally. These options to weighting loads and stores are illustrated in Figure 13. Other approaches, like taking reuse distance into account to decide on internal versus external memory accesses, are again possible, although with each more practical consideration for memory levels and technology guided design it becomes more polluted with memory architecture specifics.

Figure 12(d) plots the flexibility based on RISC operations with, and without loads and stores. For this evaluation, the RISC based metric is used, as it allows loads and stores to be simply weighted by one. This avoids the difficult problem of weighting the loads and stores in terms of transistors, which could, depending on the approach chosen, yield a very different flexibility ranking. Instead, Figure 12(b) shows that, when loads and stores are simply counted as a single RISC instruction, their impact on the resulting flexibility metric is minimal for the selected benchmark set. It should be noted that this observation is dependent on the evaluated benchmarks. For a matrix transpose algorithm, for example, the outcome is expected to be completely different.

9 CONCLUSIONS

The term flexibility is frequently used in computer architecture literature [12, 24, 25, 35, 55], despite the lack of both a proper qualitative and quantitative definition. This is a harmful situation

which leads to contradictory statements regarding flexibility as a property and its relation to other system metrics, and as such does not advance knowledge of computer architectures but rather dilutes fundamental reasoning. In an attempt to address this matter, a survey of compute system flexibility in literature was performed in order to collect general ideas about flexibility in the community, and hypothesized relations between other metrics such as performance and energy efficiency. Furthermore, existing definitions of flexibility and related notions were collected and classified. Based on these statements regarding flexibility in literature, first a qualitative, and consequently a quantitative definition of flexibility in computing systems was derived. As part of the quantitative definition, intrinsic workload is introduced as a generic method of normalizing applications. An accompanying open source tool was released to automate the estimation of intrinsic workload [19]. Using this tool, flexibility is evaluated on 25 platforms over 14 benchmarks, validating that the proposed metric conforms with some commonly accepted notions of flexibility.

Globally, the proposed flexibility metric orders some major architecture classes from least flexible to most flexible as: GPUs, CPUs, DSPs, and FPGAs. In particular, it is shown that the proposed metric is capable of distinguishing diverse architecture classes. The GPUs showed to be the least flexible, which seems intuitive as their performance is heavily impacted by the amount of parallelism present in applications. Most flexible are the FPGAs, but interestingly only when the high level synthesis was not optimising. This and the other measurements also align with the idea that high flexibility is on tension with high performance, especially when not all applications profit from the applied optimisation. A similar conclusion can be drawn for the relation between energy efficiency.

Apart from the 25 COTS platforms, 40 ASIPs were used to investigate the relation between flexibility and customisation. Interestingly the results show that flexibility is a property that can be improved by adding more resources, similar to performance for example. When the available resources are scarce, flexibility is typically low as only some benchmarks benefit from added compute capabilities. With more resources available, all applications in the benchmark set can be accelerated, often leading to better flexibility.

Furthermore, an extensive discussion is provided on the state of the art, the proposed flexibility metric, and several alternative choices that could be made when moving the field forward. For instance, it is shown that the impact of the proposed intrinsic workload normalisation compared to normalising by a standard RISC is fairly limited. While the theoretical case for intrinsic workload is arguably stronger, using a RISC as normalisation may be a more practical way to move forward. The inclusion of loads and stores did not impact the flexibility significantly for the selected benchmarks, although for more complicated memory systems it could still be interesting to consider them. In particular for applications that have a lot of data movement such as matrix transpose, accounting for loads and stores is expected to have a significant impact.

Finally, the proposed flexibility metric is compared in depth to the two alternative definitions found in literature. In the case of VersaBench versatility, it is shown that in fact performance is measured, and not a orthogonal property. For versatility as defined by Van Berkel [26], it is argued that it measures a slightly different, yet related property. Instead of the direct flexibility, it measures how efficiently a computer architect managed to encode the workload of the application domain. Given these observations, it is concluded that the proposed flexibility metric has its own unique merits which warrant its introduction. Furthermore, it aligns with several key notions of flexibility that seem to be shared by a majority of the community, and as such serves as a good starting point in defining a commonly accepted definition of compute system flexibility.

Overall, this work provides a survey of the current situation, a starting point in assessing processor flexibility in a quantitative manner, and lays the foundation for a broader discussion in the community.

APPENDIX

A FLEXIBILITY METRIC RELATED LEMMAS

This section lists the lemmas that are instrumental to the properties of flexibility metric discussed in Section 3.2.1. Note that in this section the term "positive dataset" is to be understood as a set of positive real numbers.

Lemma A.1. *The **geometric mean (GM)** of a positive dataset $\vec{X}$ increases if an element $x_i \in \vec{X}$ increases.*

Proof. Let $GM(\vec{X}) = (x_1 \cdot x_2 \cdots x_n)^{\frac{1}{n}}$ denote the geometric mean of positive dataset $\vec{X}$. Then for dataset $\vec{X}$ and its incremented version $\vec{X'}$:

$$GM(\vec{X})^n = x_1 \cdot x_2 \cdots x_n, \text{ and } GM(\vec{X'})^n = x_1 \cdot x_2 \cdots x_k{}' \cdots x_n, \text{ where } x_k{}' = x_k + \epsilon \text{ with } \epsilon > 0$$

It follows that:

$$GM(\vec{X'})^n - GM(\vec{X})^n = (x_1 \cdot x_2 \cdots x_k{}' \cdots x_n) - (x_1 \cdot x_2 \cdots x_k \cdots x_n) = x_1 \cdot x_2 \cdots (x_k{}' - x_k) \cdots x_n > 0$$

And since $GM(\vec{X})^n$ is a positive, monotonically increasing function for positive dataset $\vec{X}$ and $n = |\vec{X}|$, it follows that $GM(\vec{X'}) > GM(\vec{X})$. □

Lemma A.2. *The **geometric standard deviation (GSD)** of a positive dataset $\vec{X}$ can either increase or decrease when an element of $\vec{X}$ increases.*

Proof. Let $\vec{X} = [1, 3, 2, 2]$ be the original positive dataset, and $\vec{X'} = [2, 3, 2, 2]$, $\vec{X''} = [10, 3, 2, 2]$ be two datasets after increasing the first element of $\vec{X}$.

$$GSD(\vec{X}) = GSD([1, 3, 2, 2]) \approx 1.48$$
$$GSD(\vec{X'}) = GSD([2, 3, 2, 2]) \approx 1.19$$
$$GSD(\vec{X''}) = GSD([10, 3, 2, 2]) \approx 1.93$$

Hence, an increase of an element in $\vec{X}$ can either increase or decrease the geometric standard deviation of $\vec{X}$. □

Lemma A.3. *The geometric standard deviation (GSD) is invariant to multiplicative scaling, i.e., $GSD(s \cdot \vec{X}) = GSD(\vec{X})$, where $\vec{X}$ is a positive dataset and 's' is a positive constant.*

Proof.

$$GM(s \cdot \vec{X}) = \left(\prod_{i=1}^{n} s \cdot x_i\right)^{\frac{1}{n}} = \left(s^n \prod_{i=1}^{n} x_i\right)^{\frac{1}{n}} = s \cdot GM(\vec{X})$$

$$GSD(s \cdot \vec{X}) = \exp\left(\sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\ln \frac{s \cdot x_i}{GM(s \cdot \vec{X})}\right)^2}\right) = \exp\left(\sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\ln \frac{x_i}{GM(\vec{X})}\right)^2}\right) = GSD(\vec{X})$$ □

Lemma A.4. *The geometric mean (GM) of a dataset $\vec{X}$ normalized to dataset $\vec{B}$ is equal to the ratio of the GMs of $\vec{X}$, and $\vec{B}$, i.e., $GM(\frac{\vec{X}}{\vec{B}}) = \frac{GM(\vec{X})}{GM(\vec{B})}$, where $\vec{X} = [x_1, x_2, \ldots, x_n]$ and $\vec{B} = [b_1, b_2, \ldots, b_n]$ are positive datasets.*

Proof.

$$\frac{GM(\vec{X})}{GM(\vec{B})} = \frac{\left(\prod_{i=1}^{n} x_i\right)^{\frac{1}{n}}}{\left(\prod_{i=1}^{n} b_i\right)^{\frac{1}{n}}} = \left(\frac{\prod_{i=1}^{n} x_i}{\prod_{i=1}^{n} b_i}\right)^{\frac{1}{n}} = \left(\prod_{i=1}^{n} \frac{x_i}{b_i}\right)^{\frac{1}{n}} = GM\left(\frac{\vec{X}}{\vec{B}}\right)$$ □

Lemma A.5. *The geometric mean of dataset $\vec{X}$ normalized to dataset $\vec{B}$ is equal to the reciprocal of the geometric mean of $\vec{B}$ normalized to $\vec{X}$, i.e., $GM(\frac{\vec{X}}{\vec{B}}) = GM(\frac{\vec{B}}{\vec{X}})^{-1}$*

Proof.

$$GM\left(\frac{\vec{X}}{\vec{B}}\right) \overset{\text{Lemma A.4}}{=\!=} \frac{GM(\vec{X})}{GM(\vec{B})} = \left(\frac{GM(\vec{B})}{GM(\vec{X})}\right)^{-1} \overset{\text{Lemma A.4}}{=\!=} GM\left(\frac{\vec{B}}{\vec{X}}\right)^{-1} \qquad \square$$

Lemma A.6. *The ratio of the geometric means (GMs) of different normalized dataset is the same as the ratio of the GMs of the original datasets, i.e., $\frac{GM(\frac{\vec{X}}{\vec{B}})}{GM(\frac{\vec{Y}}{\vec{B}})} = \frac{GM(\vec{X})}{GM(\vec{Y})}$, where $\vec{X}$, $\vec{Y}$, and $\vec{B}$ are positive datasets.*

Proof.

$$\frac{GM\left(\frac{\vec{X}}{\vec{B}}\right)}{GM\left(\frac{\vec{Y}}{\vec{B}}\right)} \overset{\text{Lemma A.4}}{=\!=} \frac{\frac{GM(\vec{X})}{GM(\vec{B})}}{\frac{GM(\vec{Y})}{GM(\vec{B})}} = \frac{GM(\vec{X})}{GM(\vec{Y})} \qquad \square$$

Lemma A.7. *The geometric standard deviation (GSD) of a normalized dataset is equal to the GSD of the reciprocal of that normalized dataset, i.e., $GSD(\frac{\vec{X}}{\vec{B}}) = GSD(\frac{\vec{B}}{\vec{X}})$, where $\vec{X}$ and $\vec{B}$ are positive datasets.*

Proof.

$$GSD\left(\frac{\vec{X}}{\vec{B}}\right) = \exp\left(\sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\ln\frac{\frac{x_i}{b_i}}{GM\left(\frac{\vec{X}}{\vec{B}}\right)}\right)^2}\right) = \exp\left(\sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\ln x_i - \ln\left(b_i \cdot GM\left(\frac{\vec{X}}{\vec{B}}\right)\right)\right)^2}\right)$$

$$= \exp\left(\sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\ln\left(b_i \cdot GM\left(\frac{\vec{X}}{\vec{B}}\right)\right) - \ln x_i\right)^2}\right) = \exp\left(\sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\ln\frac{b_i \cdot GM\left(\frac{\vec{X}}{\vec{B}}\right)}{x_i}\right)^2}\right)$$

$$\overset{\text{Lemma A.5}}{=\!=} \exp\left(\sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\ln\frac{\frac{b_i}{x_i}}{GM\left(\frac{\vec{B}}{\vec{X}}\right)}\right)^2}\right) = GSD\left(\frac{\vec{B}}{\vec{X}}\right) \qquad \square$$

Lemma A.8. *The geometric standard deviation (GSD) of a normalized dataset is always greater than or equal to the ratio of the GSDs of the original dataset and the baseline, i.e., $GSD(\frac{\vec{X}}{\vec{B}}) \geq \frac{GSD(\vec{X})}{GSD(\vec{B})}$, where $\vec{X}$ and $\vec{B}$ are real positive datasets.*

Proof. First rewrite $GSD(\frac{\vec{X}}{\vec{B}})$ and $\frac{GSD(\vec{X})}{GSD(\vec{B})}$:

$$GSD\left(\frac{\vec{X}}{\vec{B}}\right) = \exp\left(\sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\ln\frac{\frac{x_i}{b_i}}{GM\left(\frac{\vec{X}}{\vec{B}}\right)}\right)^2}\right) \overset{\text{Lemma A.4}}{=\!=} \exp\left(\sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\ln\frac{\frac{x_i}{b_i}}{\frac{GM(\vec{X})}{GM(\vec{B})}}\right)^2}\right)$$

$$= \exp\left(\sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\ln\frac{x_i}{GM(\vec{X})} - \ln\frac{b_i}{GM(\vec{B})}\right)^2}\right)$$

$$\frac{GSD(\vec{X})}{GSD(\vec{B})} = \frac{\exp\left(\sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\ln\frac{x_i}{GM(\vec{X})}\right)^2}\right)}{\exp\left(\sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\ln\frac{b_i}{GM(\vec{B})}\right)^2}\right)} = \exp\left(\sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\ln\frac{x_i}{GM(\vec{X})}\right)^2} - \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\ln\frac{b_i}{GM(\vec{B})}\right)^2}\right)$$

Since $\exp(y)$ is a positive monotonic function, it is sufficient to prove:

$$\sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\ln\frac{x_i}{GM(\vec{X})} - \ln\frac{b_i}{GM(\vec{B})}\right)^2} \geq \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\ln\frac{x_i}{GM(\vec{X})}\right)^2} - \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\ln\frac{b_i}{GM(\vec{B})}\right)^2}$$

Substituting $\vec{U} = \ln\frac{x_i}{GM(\vec{X})}$, and $\vec{V} = \ln\frac{b_i}{GM(\vec{B})}$ yields:

$$||\vec{U} - \vec{V}|| \geq ||\vec{U}|| - ||\vec{V}||$$

Which holds according to the reverse triangle inequality. □

REFERENCES

[1] David Abdurachmanov, Peter Elmer, Giulio Eulisse, and Shahzad Muzaffar. 2014. Initial explorations of ARM processors for scientific computing. *Journal of Physics: Conference Series* 523, 1 (2014), 012009.

[2] AnandTech. 2013. The Haswell Review: Intel Core i74770K & i5-4670K Tested. https://www.anandtech.com/show/7003/the-haswell-review-intel-core-i74770k-i54560k-tested, accessed 2021-3-30.

[3] Chris Angelini. 2015. GeForce GTX Titan X Review: Can One GPU Handle 4K? https://www.tomshardware.com/reviews/nvidia-geforce-gtx-titan-x-gm200-maxwell,4091.html, accessed on 2021-3-30.

[4] Y. Arai, T. Agui, and M. Nakajima. 1988. A fast DCT-SQ scheme for images. *IEICE Transactions* (1988).

[5] M. Arnold and H. Corporaal. 1999. Automatic detection of recurring operation patterns. In *Proceedings of the Seventh International Workshop on Hardware/Software Codesign: (CODES'99), May 3–5, 1999, Rome, Italy*. Association for Computing Machinery, Inc., United States, 22–26.

[6] David Buchfuhrer and Christopher Umans. 2008. The complexity of Boolean formula minimization. *Journal of Computer and System Sciences - JCSS* 77, 24–35.

[7] cadence. 2009. *Encounter(R) RTL Compiler*. https://www.csee.umbc.edu/ tinoosh/cmpe641/tutorials/rc/rc_commandref.pdf. Version v11.20.

[8] G. Chryssolouris. 1996. Flexibility and its measurement. *CIRP Annals* 45, 2 (1996), 581–587.

[9] Michael Cloutier, Chad Paradis, and Vincent Weaver. 2016. A Raspberry Pi cluster instrumented for fine-grained power measurement. *Electronics* 5 (Sept. 2016), 61.

[10] L. Codrescu, W. Anderson, S. Venkumanhanti, M. Zeng, E. Plondke, C. Koob, A. Ingle, C. Tabony, and R. Maule. 2014. Hexagon DSP: An architecture optimized for mobile multimedia and communications. *IEEE Micro* 34, 2 (Mar. 2014).

[11] R. Damodaran, T. Anderson, S. Agarwala, R. Venkatasubramanian, M. Gill, D. Gopalakrishnan, A. Hill, A. Chachad, D. Balasubramanian, N. Bhoria, J. Tran, D. Bui, M. Rahman, S. Moharil, M. Pierson, S. Mullinnix, H. Ong, D. Thompson, K. Gurram, O. Olorode, N. Mahmood, J. Flores, A. Rajagopal, S. Narnur, D. Wu, A. Hales, K. Peavy, and R. Sussman. 2012. A 1.25GHz 0.8W C66x DSP core in 40nm CMOS. In *2012 25th International Conference on VLSI Design*. IEEE, Hyderabad, India, 286–291.

[12] Ahmed Osman El-Rayis. 2014. *Reconfigurable Architectures for the Next Generation of Mobile Device Telecommunications Systems*. Ph. D. Dissertation.

[13] Robert Fasthuber, Francky Catthoor, Praveen Raghavan, and Frederik Naessens. 2013. *Energy-Efficient Communication Processors: Design and Implementation for Emerging Wireless Systems*. Springer Publishing Company, Incorporated.

[14] Joseph A. Fisher, Paolo Faraboschi, and Giuseppe Desoli. 1996. Custom-fit processors: Letting applications define architectures. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture* (Paris, France) *(MICRO 29)*. IEEE Computer Society, USA, 324–335.

[15] Philip J. Fleming and John J. Wallace. 1986. How not to lie with statistics: The correct way to summarize benchmark results. *Commun. ACM* 29, 3 (March 1986), 218–221.

[16] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos. 2012. Auto-tuning a high-level language targeted to GPU codes. In *2012 Innovative Parallel Computing (InPar)*. 1–10.

[17] Rehan Hameed. 2013. *Balancing Efficiency and Flexibility in Specialized Computing*. Ph. D. Dissertation.

[18] John L. Hennessy and David A. Patterson. 2011. *Computer Architecture, Fifth Edition: A Quantitative Approach* (5th ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[19] Shihua Huang and Luc Waeijen. 2021. Intrinsic WorkloadEstimator. https://gitlab.com/lwaeijen/WorkloadEstimator.
[20] Amr Hussam Ibrahim, Mohamed Bakr Abdelhalim, Hanadi Hussein, and Ahmed Fahmy. 2011. An Analysis of x86-64 Instruction Set for Optimization of System Softwares.
[21] Texas Instruments. 2011. *Code Composer Studio*. https://software-dl.ti.com/ccs/esd/documents/ccs_downloads.html#code-composer-studio-version-4-downloads. Version V4.
[22] Intel. 2016. Inside 6th GEN Intel Core: New Microarchitecture Code Named Skylake. https://old.hotchips.org/wp-content/uploads/hc_archives/hc28/HC28.23-Tuesday-Epub/HC28.23.90-High-Perform-Epub/HC28.23.911-Skylake-Doweck-Intel_SK3-r13b.pdf. accessed 2021-3-30.
[23] Intel. 2021. https://ark.intel.com/#@Processors, accessed 2021-3-30.
[24] Götz Kappen and Tobias Noll. 2006. Application specific instruction processor based implementation of a GNSS receiver on an FPGA. 58–63.
[25] Kingshuk Karuri and Rainer Leupers. 2014. *Application Analysis Tools for ASIP Design: Application Profiling and Instruction-set Customization*. Springer Publishing Company, Incorporated.
[26] Kees van Berkel. 2013. *Processor Versatility (Flexibility) - An Attempt at Definition and Quantification*. MPSoC 2013. http://mpsoc-forum.org/archive/2013/slides/12-Van_Berkel.pdf.
[27] W. Kellerer, A. Basta, P. Babarczi, A. Blenk, M. He, M. Klugel, and A. M. Alba. 2018. How to measure network flexibility? A proposal for evaluating softwarized networks. *IEEE Communications Magazine* (2018), 2–8.
[28] T. B. L. Kirkwood. 1993. Geometric standard deviation - reply to Bohidar. *Drug Development and Industrial Pharmacy* 19, 3 (1993), 395–396.
[29] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. 2003. Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction. In *Proceedings 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.* 81–92.
[30] R. Landauer. 1961. Irreversibility and heat generation in the computing process. *IBM Journal of Research and Development* 5, 3 (1961), 183–191.
[31] E. Lannoye, D. Flynn, and M. O'Malley. 2012. Evaluation of power system flexibility. *IEEE Transactions on Power Systems* 27, 2 (May 2012), 922–931.
[32] Fei Liu, Yi Liang, and Lingze Wang. 2017. A survey of the heterogeneous computing platform and related technologies. *DEStech Transactions on Engineering and Technology Research* (May 2017).
[33] Pedro M. M. Pereira, Patrício Domingues, Nuno Rodrigues, Gabriel Falcao, and Sergio De Faria. 2016. Assessing the performance and energy usage of multi-CPUs, multi-core and many-core systems: The MMP image encoder case study. *International Journal of Distributed and Parallel Systems* 7 (Sept. 2016), 01–20.
[34] M. N. Martinez and M. J. Bartholomew. 2017. What does it mean? A review of interpreting and calculating different types of means and standard deviations. *Pharmaceutics* (April 2017).
[35] Geoffrey Ndu. 2012. *Boosting Single Thread Performance in Mobile Processors using Reconfigurable Acceleration*. Ph. D. Dissertation.
[36] Linda Null and Julia Lobur. 2014. *The Essentials of Computer Organization and Architecture* (4th ed.). USA.
[37] Nvdia. 2021. GEFORCE. https://www.nvidia.com/en-us/geforce/, accessed 2021-3-30.
[38] Nvidia. 2014. Whitepaper NVIDIA tegra K1: A new era in mobile computing. (January 2014).
[39] Nvidia. 2022. *CUDA Zone*. https://developer.nvidia.com/cuda-zone.
[40] Vishwamitra Oree and Sayed Z. Sayed Hassen. 2016. A composite metric for assessing flexibility available in conventional generators of power systems. *Applied Energy* 177 (2016), 683–691.
[41] Qualcomm. 2016. Hexagon Simulator User Guide. (December 2016).
[42] Qualcomm. 2017. Qualcomm Hexagon DSP. (December 2017).
[43] Qualcomm. 2021. *Hexagon SDK*. https://developer.qualcomm.com/downloads/hexagon-sdk-v354-linux. Version v3.5.3 Linux.
[44] Jan M. Rabaey, Anantha Chandrakasan, and Borivoje Nikolic. 2004. *Digital Integrated Circuits – A Design Perspective* (2nd ed.). Prentice Hall.
[45] Rodric Rabbah, Ian Bratt, Krste Asanovic, and Anant Agarwal. 2005. Versatility and VersaBench: A new metric and a benchmark suite for flexible architectures. (Dec. 2005).
[46] Mike Santarini. 2014. Xilinx ships industry's first 20-nm all programmable devices. *Xcell Journal* (2014), 9–15.
[47] Paul D. Stigall and Ömür Tasar. 1975. A measure of computer flexibility. *Computers & Electrical Engineering* 2, 2 (1975).
[48] V. Sze, Y. H. Chen, T. J. Yang, and J. S. Emer. 2020. *Efficient Processing of Deep Neural Networks – Synthesis Lectures on Computer Architecture*. Morgan & Claypool Publishers.
[49] GCC team. 2021. *GNU Compiler Collection*. https://gcc.gnu.org/.
[50] Texas Instruments. 2014. TMS320C6745, TMS320C6747 fixed- and floating-point digital signal processor. (2014).
[51] E. Tomusk, C. Dubach, and M. O'Boyle. 2014. Measuring flexibility in single-ISA heterogeneous processors. In *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*. 495–496.

[52] Gregory K. Wallace. 1991. The JPEG still picture compression standard. *Commun. ACM* 34, 4 (April 1991), 30–44.
[53] M. Wijtvliet, L. Waeijen, and H. Corporaal. 2016. Coarse grained reconfigurable architectures in the past 25 years: Overview and classification. In *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*. 235–244.
[54] Rand R. Wilcox and H. J. Keselman. 2003. Modern robust data analysis methods: Measures of central tendency. *Psychological Methods* 8, 3 (2003).
[55] Markus Willems. 2019. *Application-Specific Processors for High Throughput, Low Latency, and Flexible 5G Communication SoCs*. Synopsis. https://www.synopsys.com/designware-ip/technical-bulletin/5g-asips-communication-socs.html.
[56] Xilinx. 2018. *Vivado High-Level Synthesis*. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug902-vivado-high-level-synthesis.pdf. Version v2018.2.

Received August 2021; revised January 2022; accepted March 2022