computers

MDPI

Article

Low Effort Design Space Exploration Methodology for Configurable Caches †

Mohamad Hammam Alsafrjalani [1,*] and Ann Gordon-Ross [2]

[1] Department of Electrical and Computer Engineering, University of Miami, Coral Gables, FL 33146, USA
[2] Department of Electrical and Computer Engineering, University of Florida, Gainesville, FL 32608, USA; Ann@ece.ufl.edu
* Correspondence: Alsafrjalani@miami.edu
† This paper is an extended version of our paper published in Proceedings of the 12th IEEE International Conference on Embedded and Ubiquitous Computing, Milano, Italy, 26–28 August 2014.

Received: 12 February 2018; Accepted: 17 March 2018; Published: 27 March 2018

Abstract: Designers can reduce design space exploration time and efforts using the design space subsetting method that removes energy-redundant configurations. However, the subsetting method requires a priori knowledge of all applications. We analyze the impact of a priori application knowledge on the subset quality by varying the amount of a priori application information available to designers during design time from no information to a general knowledge of the application domain. The results showed that only a small set of applications representative of the anticipated applications' general domains alleviated the design efforts and was sufficient to provide energy savings within 5.6% of the complete, unsubsetted design space. Furthermore, since using a small set of applications was likely to reduce the design space exploration time, we analyze and quantify the impact of a priori applications knowledge on the speedup in the execution time to select the desired configurations. The results revealed that a basic knowledge of the anticipated applications reduced the subset design space exploration time by up to 6.6X.

Keywords: design space exploration; embedded systems; configurable caches; cache tuning; design space subsetting

1. Introduction

Designing a system that meets optimization goals (e.g., performance, energy, energy-delay product (EDP), etc.), is becoming a major goal for system designers in all computing device domains, from high-performance servers to embedded systems. Since hardware requirements, such as the cache memory size, processor speed, front bus speed, number arithmetic logic units, etc., differ for each application, an optimized system for one application is not necessarily an optimized system for another application.

System optimization can be achieved using configurable hardware, which has configurable/tunable parameters [1–4] that can be tuned during runtime to adhere to optimization goals. A set of parameter values constitutes a configuration and all possible configurations compose the complete design space. The configuration that most closely adheres to the application's requirements while achieving optimization goals constitutes the application's best configuration.

Tuning is the process of exploring the design space to determine the best configuration. Many prior tuning methods [5–7] executed applications on different configurations during runtime, measured execution characteristics, such as the cache miss-rate, cycles per instruction, using hardware counters, and then selected the best configuration. However, these methods incurred significant tuning overhead, including energy and performance overheads expended while executing the application in non-best configurations. To reduce the tuning overhead, other methods [8,9] measured the application's

execution characteristics and used analytical models to predict the best configuration based on these characteristics. However, these methods required complex computations and lengthy calculation processing time [8], or high application persistency to amortize the tuning overhead [9].

To eliminate the runtime tuning overhead and complex computations, Wang et al. [10] tuned the configurable hardware offline, prior to system runtime, stored the application's best configurations in a lookup table, and used the information during runtime to tune the hardware to the best configuration. However, offline tuning required a priori knowledge of applications and a lengthy simulation time for large design spaces.

Since one of the major challenges with configurable hardware is efficiently and effectively searching a large design space, prior work [11] empirically determined that the complete design space was not necessary to achieve optimization goals. The authors observed that many configurations provided similar optimization potential, such that the design space contains many near-best configurations. Near-best configurations are configurations that adhere to optimization goals nearly as well as the best configuration and therefore, the design space does not need to contain all configurations to adhere to the optimization goals. Runtime tuning overhead and lengthy simulation time of offline tuning can be reduced if the complete design space is pruned into a subset of configurations containing near-best configurations.

However, since applications have disparate hardware requirements, the near-best configurations are application-specific. Given a general purpose system, the subset must be large enough to contain best or near-best configurations for all applications that are expected to run on the system in order to adhere to optimization goals for every application. A major challenge in selecting this subset is ensuring that the subset is large enough to meet all disparate application requirements, but not so large that the design space exploration time remains long. Therefore, the best tradeoff between the subset size and adherence to optimization goals requires an exhaustive exploration of all subset sizes and subset configuration combinations [11].

To determine the best subset size and configurations, designers must evaluate all of the applications' optimization goal adherence for every combination of subset size and configurations, and then select the highest quality subset. High-quality subsets are subsets that contain a combination of configurations that most closely adhere to optimization goals as compared to the complete design space. Although prior work [11] used a heuristic [12] to speed up the high-quality subset determination, the heuristic still required a priori knowledge of all expected applications in order to evaluate the subset quality, which limited the usage of configuration subsetting in general purpose systems.

The challenge is then, during design time, to determine the subsets that contain best or near-best configurations with very low or no a priori knowledge of the applications. However, since applications are typically required to evaluate the configurations' qualities and select the best subset, varying the a priori knowledge of the application will affect the subset's quality. Furthermore, since configuration/subset quality evaluations require executing the applications, the number of applications evaluated impacts the evaluation time. However, the extent of this impact is not linear, since application execution times vary based on the nature of the application (i.e., using half of the applications does not necessarily result in a 2X speedup in the design space exploration times). To gain an insight into the performance-quality tradeoff, it is imperative to determine the minimum number of applications required to select high-quality subsets.

In this paper, we evaluated and quantified the extent to which a priori knowledge of all anticipated applications affects subset quality and determined that a small training set of applications provides sufficiently high-quality subsets. This observation significantly broadens the usability of the heuristic [12] by alleviating the requirement of a priori knowledge of all applications. We used training application sets to determine the configuration subsets and evaluated the quality of the configuration subsets using a disjoint testing application sets. To evaluate the subset quality, we compared the adherence to the design optimization goals using the subset's configurations as compared to using the complete design space.

Based on these results, we conjectured that if there is a basic knowledge of the anticipated applications' hardware requirements (e.g., a large cache size, deep pipeline, fast clock frequency, etc.), a test set containing applications with similar hardware requirements would improve the subset quality. To test this correlation, we used three methods to select our training application sets: a random method which required no a priori knowledge of anticipated applications, and two classification methods. The first classification method is a basic classification method that classified applications based on the general hardware requirements of the anticipated applications. To gain insights on the classification method's impact on selecting a subset, the second classification method hierarchically classified applications based on the similarity of the applications' hardware requirements.

Additionally, to quantify the impact of a priori knowledge on the speedup of the heuristic [12] to select a high-quality subset, we compared the execution time of our algorithms to prior work [11]. We compared the execution time that our random method required to select a subset using training applications as compared to using the complete application set. Furthermore, since training applications differ for different classes, we compared the execution time of our classification methods to determine the subsets as compared to our random method.

Our results revealed that a priori knowledge of all anticipated applications is not required to obtain high-quality subsets, and rather, a fraction of training applications is sufficient to obtain a configuration subset of high-quality (i.e., a subset with 22.2% of the complete design space achieves an average energy savings within 17.7% of the complete design space). These results suggest that subsets alleviate design efforts for general purpose computing while closely adhering to optimization goals. Our results also showed that the subset quality can be improved with a basic knowledge of the anticipated applications' hardware requirements, by as much as 10.4% on average. Thus, designers can determine application-domain-specific subsets and reuse the subset for any application that belongs to that domain. However, using a domain-specific subset for an application not of that domain greatly depreciated that application's energy savings. Our hierarchical classification revealed that this degradation in energy saving is alleviated if applications are classified using the application's hardware requirements.

Our speedup results showed that the training application set had the most impact on subset design space exploration time, and a basic knowledge of the anticipated applications reduced the subset design space exploration time by up to 6.6X. Additionally, our results revealed that hierarchical classification performed slower than basic classification since hierarchical classification iteratively classified applications based on the applications' hardware requirements.

The remainder of this paper is organized as follows. Section 2 discusses the foundational related work that makes our contributions possible. Since the cache memory hierarchy has a significant impact on the system's performance and energy, Section 3 discusses our cache configurations design space. Section 4 introduces design space exploration as a data mining algorithm, and the heuristic we evaluated in this work. Section 5 describes our training application selection methodologies used in this paper. Section 6 provides our experiment setup environment, and Sections 7 and 8 discuss the results and analysis of the subset quality and speedup, respectively.

2. Related Work

To adhere to the application hardware requirements, much prior research focused on configurable hardware (e.g., [13–23]), configurable caches (e.g., [3,4,24–27]), and design space exploration (e.g., [5–7,9–11,28–32]). Given this expansive prior work, we discuss fundamentals of configurable hardware, followed by specific related work in configurable caches and design space exploration with fundamentals that are directly applicable to our approach.

2.1. Configurable Hardware

Folegnani [22] dynamically resized the instruction queue while disabling the associated instruction wakeup logic. Khan et al. [13] designed a model that predicts the optimum instruction-issue window

size based on the CPU events and the instruction level parallelism. The model suggested the instruction issue-window size that best utilized the CPU. Adegbija et al. [14] used designed a methodology that leverages the configurable issue queue (IQ) and reorder buffer (ROB) to save energy. To optimize the execution, the methodology adjusted the IQ and ROB sizes based on the application's execution. The results revealed 23% saving of energy than when compared to a system with fixed IQ and ROB sizes. Whereas these works provided novel contributions in configurable hardware, the authors did not consider components with larger energy contributions (e.g., cache memory).

In order to increase the total CPU energy savings, extensive research exploited the potential energy savings using dynamic voltage and frequency scaling (DVFS). Previous works have explored DFVS [15] as a solution to save energy, included DVFS as part of online and offline scheduling algorithms [17], increased the DVFS efficiency by allowing different units inside a CPU to operate at different voltage-frequencies [2], enabled a group of components to operate at a different voltage-frequency [18,19], leveraged DVFS to reduce idle energy [20], or evaluated the tradeoff between the discrete values of DVFS and energy savings [21,23]. Whereas these works contributed novel hardware architectures using DVFS, we plan to include DVFS configurations (e.g., as in Reference [19,21]) in our design space in future works.

2.2. Configurable Caches

Due to the cache's large contribution to total CPU energy, much research has focused on configurable caches. Using the Motorola M-CORE M3 processor, Malik et al. [3] varied three configurable parameters: write policy (write through or write back), write buffer (push or store), and way management (direct-mapped or n-way set associative) to measure performance and energy consumption for different applications to meet the applications' specific optimization goals.

To increase cache configurability, much of the prior work developed cache parameters that are configurable. Patel et al. [24] developed a configurable indexing cache that reduced the cache conflict misses. Zhang et al. proposed configurable associativity [4] and line size [25], which reduced memory access energy by 60% and 37%, respectively. The proposed work used an algorithm that takes in an application trace to predict the index of the data placement in the cache. Biglari et al. [26] developed a fine-grained configurable cache architecture for soft processors. The cache was fabricated on an FPGA and allowed extra victim buffers to be instantiated on high conflict misses.

Whereas these prior works provided novel contributions to configurable caches, they required the designer to select and statically determine the best cache configuration during design time (e.g., Reference [10]), or required special hardware support (e.g., Reference [3]) and design space exploration algorithms (e.g., Reference [4]) to quickly search the design space during runtime.

2.3. Design Space Exploration

Chen et al. [5] proposed a runtime reconfiguration management algorithm that searched the design space by exploring one parameter at a time (cache size, line size, then associativity) while holding the other parameters fixed. Palesi et al. [6] used genetic algorithms to find Pareto-optimal configurations using Platune [28], a tuning framework for system-on-a-chip platforms. Zhang et al. [7] proposed a heuristic that searched the design space in order of the cache parameters' impacts on energy to determine Pareto-optimal configurations.

Since during design space exploration, physically executing each explored configuration affects the application's runtime behavior, Gordon-Ross et al. [9] proposed a non-intrusive oracle hardware that executed in parallel with the cache, evaluated all possible cache configurations simultaneously, and determined the best configuration. Even though this oracle hardware eliminated the performance overhead, the Oracle hardware imposed significant power and energy overheads and thus, was only feasible for systems with persistent applications to amortize these overheads. However, these prior design space exploration methods still considered the entire design space before tuning the cache to the best configuration.

To reduce the design space, Viana et al. [11] used a subsetting method which trimmed the design space into a small subset of good and near-good configuration. Viana et al. evaluated the subset size to quality tradeoff exhaustively. The results revealed that a subset of size 4 (out of 18) configurations provided energy savings within 1% and 5% of the complete design space for the instruction and data cache, respectively. Palermo et al. [29] developed a methodology that iteratively eliminated the evaluated configurations from the design space. On each iteration, a set of Pareto configurations was selected. The method continued to select these configuration sets until a certain threshold was met. The threshold depended on the convergence of the configurations and a predefined maximum number of simulations. Thus, the threshold allowed for the stopping of the algorithm before it reached a steady state, avoiding the case of long explorations. However, Viana et al. and Palermo et al.'s methods required a priori knowledge of all anticipated applications, an infeasible requirement for general purpose systems that run applications that are unknown during design time.

3. Selecting and Evaluating Configuration Subsets

Whereas our methodology is applicable to any architectural parameter such as pipeline depth, issue width, etc., our design space comprises of the common configurable cache configuration, and our subset evaluation compares the quality of each subset to the complete, unsubsetted design space.

3.1. Selected Design Space

Large design spaces provide a designer with great design flexibility and enable a wide range of different configurations that are suitable for the system's intended domain and thus, disparate application requirements. For example, high-performance desktop computers typically require larger memory sizes/line sizes/associativities, deeper pipeline depth (PD), wider issue window (IW) sizes, larger reorder buffer (ROB) sizes, etc., as compared to embedded systems, due to typically larger working set sizes. Whereas evaluating the cross product of the configurations for many configurable parameters provides fine-grained subset evaluation, the number of configurations to evaluate would make the evaluation time-prohibitive. For instance, 18 cache $\times$ 24 PD $\times$ 8 IW $\times$ 8 ROB configurations would result in 27,648 total configurations, and using Equation (2), 1.33×10^{513} possible subsets. Whereas our approach can easily be used with other configuration parameters, independently (e.g., PD) or jointly (e.g., IW $\times$ ROB). Thus, we focus on configurable caches only due to the cache's significant impact on energy and performance values.

Table 1 lists the cache configuration design space used in our experiments, which reflects our applications' hardware requirements [3,33,34]. However, our presented methodologies are independent. The table rows represent different cache sizes in Kbytes (k) and associativities (W) in number of ways (e.g., a 2 Kbyte direct-mapped cache is denoted as 2K_1W), and the columns represent the different line sizes in bytes (B) (e.g., a 32 byte line size is denoted as 32B). Each row and column intersection denotes a unique cache configuration c_m. We note that while our methodology evaluates a single-core system with a single level of private cache, our fundamentals are applicable to arbitrary X-core systems with Y levels of private/shared cache. Additionally, evaluating a single-core system with a single level of cache enabled the validation and comparison with prior work [11].

Table 1. The cache configuration design space. Rows represent the cache size in Kbytes (K) and associativity (W). Columns represent the line sizes in bytes (B). Each row and column intersection denotes a unique cache configuration cm (size, associativity, and line size).

	16B	**32B**	**64B**
2K_1W	c_1	c_7	c_{13}
4K_1W	c_2	c_8	c_{14}
4K_2W	c_3	c_9	c_{15}
8K_1W	c_4	c_{10}	c_{16}
8K_2W	c_5	c_{11}	c_{17}
8K_4W	c_6	c_{12}	c_{18}

For comparison and analysis purposes, we designated c_{18} as the base cache configuration, which represents a common cache configuration available in commercial-off-the-shelf (COTS) embedded processors executing similar applications [4].

3.2. Selecting and Evaluating Subsets

Given a set of m cache configurations $C = \{c_1, c_2, \ldots, c_m\}$ and a set of n applications $A = \{a_1, a_2, \ldots, a_n\}$, the problem is to select a high-quality configuration subset S such that $S \subset C$ and $|S| << |C|$. Given $|S|$, the subset design space contains the number of combinations of selecting $|S|$ out of $|C|$ configurations such that

$$N(|S|) = \frac{m!}{|S|!(m-|S|)!}, \tag{1}$$

where N is the number of different subset combinations for a given subset size $|S|$. Given our cache configuration design space (Table 1), each S's quality must be evaluated for all possible subset sizes $1 \leq |S| \leq m$ for $m = 18$:

$$\sum_{|s|=1}^{18} N\begin{pmatrix} 18 \\ |S| \end{pmatrix} = 262,143 \tag{2}$$

We created all 262,143 subsets and determined the qualities of the subsets by comparing each application's energy consumption using the subset's best configuration $s_b \in S$ as compared to the complete design space's best configuration $c_b \in C$. We define $s_b \in S$ and $c_b \in C$ as the best configurations that most closely adhere to the system's design and optimization goals, which for our experiments is the lowest energy configuration. We denote the energy consumption for an application a_i executing with c_b or s_b as $e(c_b, a_i)$ or $e(s_b, a_i)$, respectively. Since $S \subset C$ and $e(s_b, a_i) \geq e(c_b, a_i)$, the energy increase $e_{inc}(s_b, c_b, a_i)$ incurred by executing application a_i with s_b instead of c_b is

$$e_{inc}(s_b, c_b, a_i) = \frac{e(s_b, a_i) - e(c_b, a_i)}{e(c_b, a_i)} \tag{3}$$

Comparing the subsets' qualities requires calculating the average energy increase μ_{inc} across all applications:

$$\mu_{inc} = \frac{\sum_{i=1}^{n} e_{inc}(s_b, c_b, a_i)}{n} \tag{4}$$

This calculation must be done for all possible subsets followed by selecting the smallest subset that most closely adheres to the system's design and optimization goals.

4. Subset Selection Heuristic

Since the design space can be viewed as a data mining problem [11], we provide background on the data mining algorithm [12] which we use to obtain our subsets.

4.1. Background of SWAB

The data mining heuristic goal is to obtain smaller, representative data points of the complete data set. Time series segmentation is a data mining heuristic that makes the data storage, transmission, and computation efficient. Common heuristics used in time series segmentation are the sliding window and bottom-up heuristics, which approximate time series segments based on the average mean square error between a time series and the time series' approximated segments. The bottom-up heuristic operates offline on a known data set, which affords a global view of the data and provides a close approximation of the time series. Alternatively, the sliding window heuristic runs in real time on streaming data, but this flexibility introduces a higher approximation error.

Keogh et al. [12] studied the feasibility and benefits of combining both algorithms into an online algorithm with semi-global approximation. The authors proposed SWAB, which applied both the sliding window and bottom-up algorithms in succession and iteratively, and was able to approximate

a time series in real time with approximated segments of equal average mean square error as the bottom-up heuristic. SWAB can be used in other data mining techniques, such as computer graphics decimation, which reduces the number of colors used in an image (i.e., the configurations in the design space) by merging similar colors (i.e., removing energy-redundant configurations) until a minimal/designated number of nuances remain (i.e., the subset size).

4.2. Design Space Exploration, a Data Mining Problem

Since design space contains the best and near-best configurations, the design space can be analogous to a computer image. A computer image comprises many color nuances, with higher degrees of nuances corresponding to higher image quality (e.g., 32-bit provides higher quality images than 16-bit). However, not all nuances are needed to depict a high-quality image and compression methods removed nuances which depict identical, or near-identical nuances [12].

Comparing design spaces to a computer image, whereas configurations correspond to color nuances, energy savings corresponds to an image quality, and a subset corresponds to a minimal number of color nuances, a design space comprises configurations which are redundant. Since prior work demonstrated SWAB's quality in computer image decimation, SWAB is applicable for design space exploration.

4.3. Using a Heuristic in Design Space Exploration

Using Table 1 as a sample configuration design space for illustrative purposes, SWAB reads in the complete cache configurations of design space C and the applications' energy values for each configuration. Then, using the bottom-up heuristic for every configuration pair (the 54 given in Table 1), SWAB calculates the energy change $\mu_\Delta(c_j, c_k)$ incurred by executing application a_i with configuration c_j instead of configuration c_k:

$$\mu_\Delta(c_j, c_k) = \frac{1}{n}\sum_{i=1}^{n} e_\Delta(c_j, c_k, a_i). \tag{5}$$

where n is the number of applications, c_j and c_k are the row- and column-adjacent configuration pairs in Table 1, and

$$e_\Delta(c_j, c_k, a_i) = \frac{e(c_j, a_i) - e(c_k, a_i)}{e(c_k, a_i)} \tag{6}$$

Algorithm 1 depicts the SWAB algorithm to subset a configurable cache design space. SWAB takes as its input, the complete design space C, an application-configuration energy matrix E (columns correspond to the applications and the rows correspond to the configurations), and a user-defined subset size x, and outputs the configuration subset S. SWAB initializes the subset S to the entire design space C (line 1); calculates μ_Δ for all adjacent configuration pairs (lines 2 through 7); determines the configuration pair with the smallest μ_Δ (line 8); merges these configurations (line 9) by eliminating the configuration that if removed, minimizes the average increase; and removes the configuration that results in the smallest μ_Δ from the design space. For example, given Equations (5) and (6), if the adjacent configuration pair with the smallest μ_Δ is (c_j c_k), then $e_{inc}(c_j,c_k) < e_{inc}(c_k,c_j)$ and c_j is the configuration removed from S. SWAB repeats lines 4 through 13 until $|S| = x$.

5. Training Set Creation Method

To evaluate the criticality of a priori knowledge of applications, we created subsets using three methods with various levels of application knowledge requirements and design effort.

5.1. Random Method Training Sets

Given an application suite A of n applications—34 in our experiment—we denoted a training set T of size i as T_i, where T_i contains i randomly-selected applications from A. We evaluate T_i's subset quality using an associated test set Q_i that contains all remaining applications and $|Q| = n - i$. Since a

single random selection of i applications from A may bias the subset quality to those particular i applications, we generalized our results by evaluating all unique combinations ω of i applications for each training set size i, resulting in

$$\omega_i = N\begin{pmatrix} n \\ T_i \end{pmatrix} \tag{7}$$

combinations for each subset size i, and a total of

$$\omega = \sum_{T_{i=1}}^{n} N\begin{pmatrix} n \\ T_i \end{pmatrix} = 17,179,869,183 \tag{8}$$

combinations.

Since the exhaustive evaluation of the complete ω is prohibitive, we evaluated T_1 through T_{10} exhaustively and extrapolated the findings to larger sizes using the correlation between the training set size and the subset's quality. We verified this extrapolation by evaluating two larger training sets T_{17} and T_{34}. T_{17} has equal-sized training and test sets and provides insights on the diminishing returns of training versus test set sizes. T_{34} includes all n anticipated applications, which leaves no remaining applications to create disjoint testing sets. For this case, the associated Q_{34} contains all n anticipated applications A, which provides an exact comparison with prior work [11].

To distinguish between the unique combinations of i applications comprising T_i, we denoted ρ_{Ti} as a single, unique combination of training applications. We used Algorithm 1 to select S for each ρ_{Ti}, and calculated the average energy Ψ_i for each test set Q_i containing application $a \notin \rho_{Ti}$:

$$\Psi_i = \frac{\sum_{\rho_{Ti=1}}^{\omega_i} e_{\rho_{Ti}}(a, s_b)}{\omega_i} \tag{9}$$

where $e_\rho(a, s_b)$ is the energy for application a executing with $s_b \in S$, created using the training set combination ρ_{Ti}.

Algorithm 1. SWAB for the cache configuration design space exploration.

Input: Complete cache configuration design space: C;
Application-configuration energy matrix: E;
User-defined subset size: x
Output: Subsetted configuration space: S

```
1  Begin   S = C;                          // start with the complete configuration space
2  While   ( | S | > x)                    // repeat until x configurations remain
3            For all adjacent pairs (cj,ck)
4                  For all applications ai
5                        find_μΔ;          // evaluate average energy change
6                  End
7            End
8            Min_pair = find_Pair_min_μΔ_  // return pair with minimum μΔ
9            merge_pair(Min_pair);
10         S = S − ci;                     // remove merged configuration
11    End while
12  Return (S);
13  End
```

5.2. Classification Method Training Sets

Since applications with similar data cache miss-rates have similar data cache requirements [32], we formulated our hypothesis: that a high-quality subset for a set of applications is a high-quality subset for other, disjoint applications that have similar data cache requirements, and then we extended

our hypothesis to the instruction cache. A cache miss results in energy consumption due to off-chip access and processor-stall, both of which expend dynamic and static energy. A reduction in the miss-rate contributes fewer cycles spent fetching data and reduces both dynamic and static energy. Our energy model accounts for access time and, we tested our approach using the cache miss-rate (complete details in Section 6).

However, measuring the similarity between applications is challenging to formulate, given the disparate applications' instruction and data access patterns. Our approach to solving this problem is to classify applications into groups that have similar miss-rates and measure the quality of configuration subsets obtained for each group of these applications. We assumed that we can obtain high-quality data cache subsets by creating training sets consisting of applications that have similar data cache miss-rates as the anticipated applications and that this conjecture extends to the instruction cache. However, this cache miss-rate classification requires design time efforts, such as determining the class size (i.e., the number of applications per class necessary to provide an accurate expected cache miss-rate for that class) and the class's minimum and maximum cache miss-rates. To evaluate the tradeoffs between classification effort and subset quality, we evaluated two classification methods: a basic, simple classification method and a more advanced, hierarchical classification method. We note that this classification method does not alleviate the requirement to profile new, unknown applications, however, it removes the requirement to profile the application for the entire design space. The base configuration, rather than all configurations, can be used to profile the application to obtain an approximate miss-rate and classify the application into the appropriate class.

5.2.1. Basic Classification

The basic classification (*BC*) method alleviates classification efforts by allowing the designer to select abstract (i.e., non-empirical) minimum and maximum miss-rates for each class based on other design constraints (e.g., required performance).

Figure 1 depicts the applications sorted in ascending order from left to right based on the applications' cache miss-rates averaged over all m configurations. The applications are demarcated into three equal-sized miss-rate classes: low, mid-range, and high.

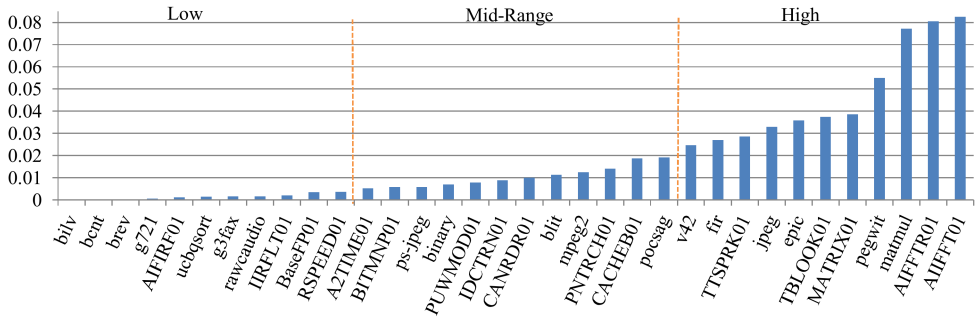

Figure 1. The applications sorted in ascending order left to right by data cache miss-rate and split into three miss-rate groups: low, mid-range, and high.

We created a training set T_{BC} for each miss-rate group. Since the results using random training sets (Section 7.1) showed that a training set of size T_3 was the smallest training set size for which every application's Ψ resulted in lower energy than executing the application with the base configuration c_{18}, each T_{BC} contained three applications, denoted as $T_{BC,3}$. To ensure a range of miss-rates and thus, cache requirements were captured by each $T_{BC,3}$, each $T_{BC,3}$ contained the lowest, mid-range, and highest miss-rate applications from the application's respective miss-rate group and the associated Q_i contained the remaining applications in the same miss-rate group. For each $T_{BC,3}$, we selected S

with Algorithm 1, determined $s_b \in S_{BC}$ from $T_{BC,3}$, and calculated e_Δ for each application with respect to Ψ_3 from T_3:

$$e_\Delta(s_b, \Psi, a) = \frac{e(s_b, a) - e(\Psi, a)}{e(\Psi, a)} \tag{10}$$

Since each $T_{BC,3}$ results in a subset S, one pitfall to the basic classification is an inflexible number of subsets for designer consideration, which limits system flexibility. For example, to fully utilize a configurable multicore system, designers can only select as many subsets as the number of cores, such that each core offers a disparate subset of configurations that adhere to a set of application hardware requirements [1]. If the number of subsets is fewer than the number of cores, a designer must replicate the subsets on the cores, which results in redundant configurations across these cores. Alternatively, if the number of subsets is greater than the number of cores, the designers must eliminate some subsets. Thus, the number of subsets must be as flexible as possible in order to enable designer's flexibility when selecting the number of system cores. Therefore, we developed the hierarchical classification (Section 5.2.2) of applications, which provides designers with the flexibility to determine the number of classes, which subsequently dictates the number of configuration subsets.

5.2.2. Hierarchical Classification

In order to remove the three-class restriction of basic classification, hierarchical classification (*HC)* allows a range of classes. Hierarchical classification starts with one class per application (34 in our case), which is called the *first level* and ends with one class which contains all applications, called the *final level*. Hierarchical classification is based on an analysis of the difference (distance) between the applications' hardware requirements, and dynamically creates an appropriate number of classes based on these distances.

Algorithm 2 depicts our hierarchical classification algorithm. The algorithm takes as its input an application miss-rate matrix X (applications identified by name or ID), a user-defined range of classes R (number of classes), and a classification metric I (Euclidian, city-block, Chebychev, etc.), and classifies the application into R-classes as follows.

Algorithm 2. The hierarchical classification algorithm.

Input: Application-ID miss-rate matrix, X;
User-defined range of classes, R;
Classification metric, I
Output: Matrix of encoded trees of hierarchical clusters, Z

```
1  Begin:
2  Define   topLevel = size(X, applications)–R+1
3  While    classificationLevel != topLevel
4               For all class- | application-pair
5               Calculate the miss-rate's I-distance //Euclidean distance is the default
6               End
7               Merge pairs with smallest d into one class
8               Assign miss-rate to class          //single distance
9               classificationLevel++
10 End
11 Return (Z)
```

To analyze the distances, the algorithm begins by calculating the pair-wise I-distance of the application's average miss-rate over m configurations. Given the low dimensions of our experiment, we default I to the Euclidean distance

$$d(a_i, a_j) = \sqrt{(miss\ rate_i - miss\ rate_j)^2}, \tag{11}$$

wherein a_i and a_j are two applications out of n applications with corresponding *miss rate*$_i$ and *miss rate*$_j$, respectively. The total number of distances measured for the first level of classification is:

$$\frac{n(n-1)}{2} \tag{12}$$

distance pairs. The applications in the pair with the smallest distance are merged into one class and this merge indicates a classification level (e.g., level 2). To analyze the similarity of the hardware requirement with respect to the newly merged class, the hierarchical classification must assign a miss-rate value to the class.

To classify the applications based on the applications' hardware requirement similarity, the smallest miss-rate of the two application miss-rate is assigned as this class's miss-rate (also known as *single distance*). For subsequent levels, the new pair-wise distances between this class and the other applications are calculated using Equation (11). This process iteratively repeats until all applications and classes are classified into R classes (top level) (e.g., $R = 3 \rightarrow$ top level is 32). The result is a hierarchical classification of classes (in n levels), each of which has a different number of applications based on the distance of the applications.

Figure 2 depicts a dendrogram representation of our 34 experimental applications classified using hierarchical classification. The applications are classified into classes based on the average miss-rate of the data cache across m configurations. The y-axis is the miss-rate differences as calculated using Equation (11), in 10^{-3} scale. For brevity and clarity, we truncated the figure between distances 6 and 14 (denoted using vertical dotted lines between 5 and 16). The larger the distance, the less similar the application's hardware requirements are.

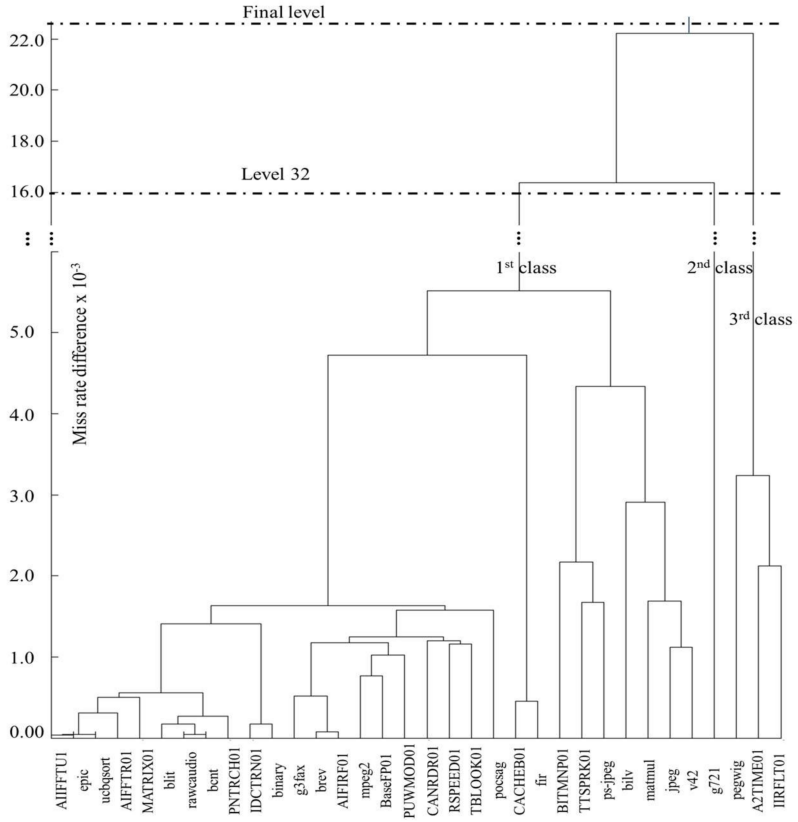

Figure 2. A dendrogram representation of hierarchical classification on the applications' data cache miss-rate difference.

To compare the results of hierarchical clustering to basic clustering, we select a level of the hierarchy, level 32, that has three classes (denoted with the horizontal dotted line in Figure 2, and used training set applications of the same size $i = 3$. These classes are denoted as the 1st, 2nd, and 3rd classes. For each class, we determined a subset of configurations using a training set of three applications $T_{HC,3}$ from that class, used the remaining applications as test applications Qi, used Equation (9) to calculate the average energy Ψ_i for $i = 3$, and compared the results to those obtained from $T_{BC,3}$ using Equation (10). However, we note that some classes had three or fewer applications, which left no remaining applications to use as test applications. For these classes, the training sets must be the same as the test sets and the quality of the subsets obtained using the training sets of for these small classes reflects the tradeoffs between basic and hierarchical classification.

6. Experimental Setup

To provide a fair comparison, we modeled our experimental setup similarly to prior work [11]. We evaluated 34 embedded system applications: sixteen from the EEMBC automotive application suite [33], six from Mediabench I [34], and twelve from Motorola's Powerstone applications [3]. Table 1 depicts our configuration design space where c_{18} is the base configuration (Section 3.1). We used Algorithm 1 to select subsets S with $|S| = 4$ [11] for random training set sizes 1–10, 17, and 34 (Section 5.1) and the basic and hierarchical classification training sets containing three applications (Section 5.2.1). For the hierarchical classification, we used Algorithm 2, M as 2-D (34 applications, average miss-rate), $R = 3$, and $I =$ "Euclidean". We set the training set size to 3 (Section 5.2.2) for each class in Z.

To evaluate the subsets' qualities of the random training sets, we compared Ψ for each application to the application's $c_b \in C$ and to the base configuration c_{18} by normalizing Ψ to c_b and c_{18}, respectively. For the basic classification training sets, we compared the test set's Q_i's applications' energy consumptions for $s_b \in S$—selected using $T_{BC,3}$—to the energy consumptions of Q_i for $s_b \in S$—selected using T_3. We compared these energy consumptions by normalizing the energy consumption of Q_i for $s_b \in S$—selected using $T_{BC,3}$—to $c_b \in C$. To compare the subset quality between hierarchical and basic classification, we normalized the energy consumption of the test applications using $s_b \in S$—selected using $T_{HC,3}$—to $s_b \in S$ (selected using T_{BC}). To quantify the tradeoff between energy savings and hardware requirements (i.e., the number of configurations), we normalized the energy consumption of Q_i for $s_b \in S$—selected using $T_{BC,3}$—to the energy consumption of Q_i for $s_b \in S$ (selected using $T_{HC,3}$).

Furthermore, to evaluate the design-time overhead tradeoff, we use the execution time of Algorithm 1 to determine S for random, basic classification, and hierarchical classification. We note that the execution time for Algorithm 2 is negligible compared to that of Algorithm 1 and thus, we exclude this comparison from our analysis.

Figure 3 depicts our cache hierarchy energy model used for the instruction and data caches, which considered the dynamic and static energies [11]. We obtained the dynamic energy using CACTI [35] for 0.18-micron technology, estimated the static energy as 10% of the dynamic energy (a valid approximation for 0.18-micron technology [9]), and calculated the CPU stall energy to be approximately 20% of the active energy [9]. Since our modeled system did not have a shared level two cache, and thus no dependencies between the instruction and data cache behavior, we evaluated separately the private level one instruction and data caches and used SimpleScalar [36] in order to obtain the cache accesses/hits/misses for each configuration for each application. We obtained off-chip access energy from a standard low-power Samsung memory, estimated that a fetch from main memory took forty times longer than a level one cache fetch, and the memory bandwidth was 50% of the miss penalty (44 cycles) [9].

E(total) = *E(sta) + E(dyn)*
E(dyn) = *cache-hits * E(hit) + cache-misses * E(miss)*
E(miss) = *E(off chip access) + miss-cycles * E(CPU stall) + E(cache fill)*
Miss Cycles = *cache-misses * miss-latency + (cache-misses * (line size/16)) x memory band width)*
E(sta) = *total-cycles * E(static-per-cycle)*
E(static-per-cycle)) = *E(per-Kbyte)*x cache-size-in-Kbytes*
E(per Kbyte) = *(E(dyn of base cache) * 10%) / (base cache size in Kbytes)*

Figure 3. A cache hierarchy energy model for the level one instruction and data caches.

7. Subset Quality Results and Analysis

We extensively evaluated the random, basic, and hierarchical classification training sets subsets' qualities, and quantified the criticality of a priori knowledge of the anticipated applications by comparing these training sets subsets' qualities. We compared the subsets selected using T_3 (created using random training sets) and $T_{BC,3}$ (created using basic classification training sets) and compared the subsets' qualities by normalizing the associated test set's Q_i, applications' energy consumptions for using $s_b \in S$–selected using T_3–to the energy consumptions for using $s_b \in S$ (selected using $T_{BC,3}$). Similarly, we analyzed the classification method's impact on selecting subsets by comparing the subsets selected using $T_{BC,3}$ to the subsets selected using $T_{HC,3}$.

7.1. Random Method Subset Quality

Figure 4a,b depict the instruction and data cache energy consumptions averaged over all test applications for the random application training sets of varying sizes and a subset size of four normalized to the best configuration in the complete design space and the base configuration, respectively. For brevity, each bar corresponds to the average (instead of per-application) energy consumption using $s_b \in S$ for all Q_i applications for a given T_i, however, we discuss and evaluate the per-application energy consumption. In Figure 4a, 1.0 corresponds to the baseline energy consumption for the best configuration $c_b \in C$ and subsets with normalized energy consumptions closer to 1.0 represent higher quality subsets. In Figure 4b, 1.0 corresponds to the baseline energy for the base configuration $c_{18} \in C$.

The instruction and data caches showed similar energy consumption trends as the training set size increased. However, since these training sets contained randomly-selected applications, on a per-application basis for the instruction cache, larger training sets did not necessarily equate to higher quality subsets, even though the moving average of the energy increase over all applications decreased as the training set size increased. For example, T_3 provided energy consumption within 16.2% of the complete design space for the instruction cache, while T_4 *increased* the energy to 20.4%. The instruction cache results showed that the largest training set T_{34} produced the highest quality subsets (the smallest average energy increase was 7.5%), while for the data cache, T_{17} produced the highest quality subsets (the smallest average energy increase was 8.0%). These results suggest that, since the data caches generally exhibit lower inherent spatial and temporal locality and thus, a greater miss-rate and cache-requirement variation, it is more challenging to capture enough cache parameter variation in a small subset created from randomly-selected training applications.

Figure 4b reveals that the random training sets' subsets always resulted in energy savings as compared to the base configuration, with a general increase in savings as the training set size increased. Even the smallest training set T_1 yielded average energy savings of 11.3% and 24.2% as compared to the base configuration for the instruction and data caches, respectively. Per-application analysis revealed that T_3 was the smallest training set size where 94.1% of the test applications in Q_3 had lower energy consumption, on average, as compared to the base configuration. For T_3, the best-case application's instruction cache had energy savings of 57.5% and 57.6%, compared to the base configuration, which

was only a 3.18% and 3.8% energy increase compared to the best configuration in the complete design space, for the instruction and data caches, respectively. On average, for the instruction and data caches, the energy increase with respect to the complete design space was 19.8% and 17.7%, respectively, and the average energy savings with respect to the base configuration was 31.3% and 29.0%, respectively.

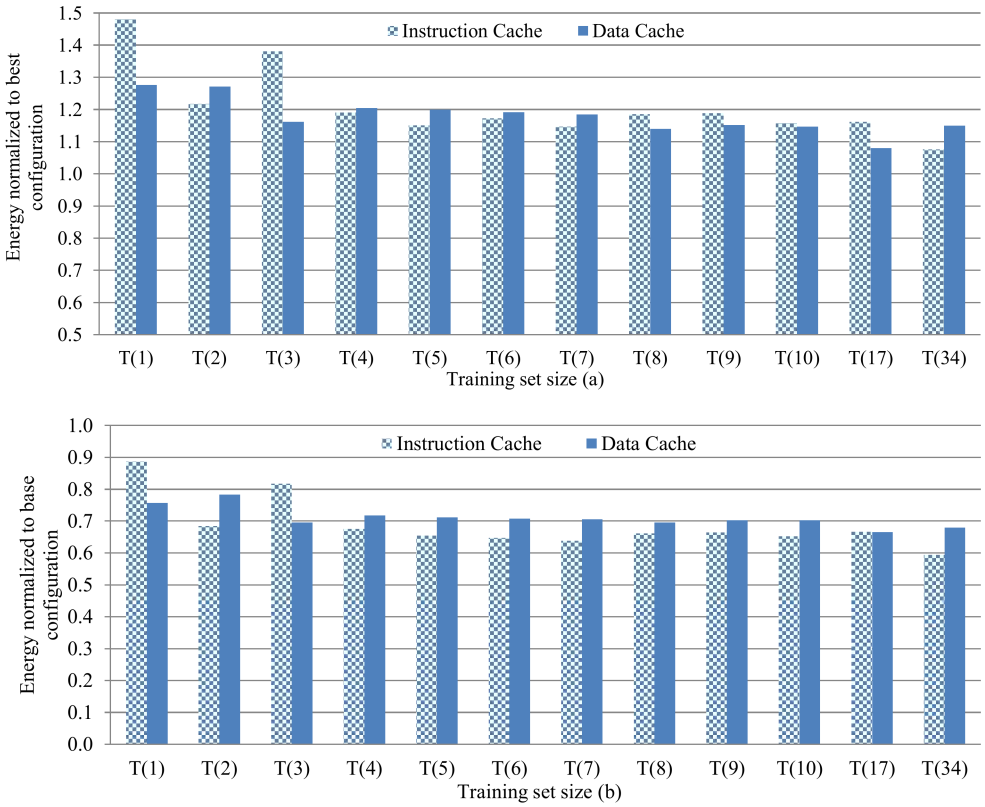

Figure 4. The average cache energy consumption for all random application test sets of varying training set sizes and a subset size of four, normalized to the best configuration in the complete design space (**a**) and to the base configuration (**b**).

Analysis of training set size versus energy savings with respect to the base configuration revealed that T_3 and T_8 had the highest average energy savings of 30.3% and 30.5%, respectively. Although T_8's energy savings was 0.5% more than T_3, T_8 required a priori knowledge of eight applications, instead of three, which imposes significantly more design-time effort due to a 3034X increase in the subset design space. Thus, T_3 provided the best tradeoff between training set size and energy savings.

7.2. Criticality of A Priori Knowledge of Applications

7.2.1. Basic Knowledge Criticality

Figure 5a,b depict the instruction and data cache energy consumptions, respectively, for test set applications for the low, mid-range, and high miss-rate groups (ordered similarly to Figure 1) normalized to Ψ, which is the average of all of the applications' energies using $s_b \in S$–selected using T_3–for all subset combinations for T_3. 1.0 denotes T_3's baseline, where normalized energies below 1.0 represent an energy improvement (i.e., an increased subset quality) as compared to T_3. On average over all miss-rate groups, the basic classification-based training sets increased the energy savings by 10.4% and 3.1% compared to T_3 for the instruction and data caches, respectively. Analyzing the

instruction and data caches' normalized energy consumption variances revealed that the instruction cache energy consumption had a higher variance than the data cache, 0.058 and 0.013, respectively.

The data cache's lower variance is expected since applications with similar data cache miss-rates indicate similar cache requirements. Thus, grouping anticipated applications into basic classification-based groups yields less varied energy savings.

We extended the analysis of Figure 5 to determine the subset quality if the subsets and test set applications belonged to different miss-rate groups. Using similar normalization as in Figure 5, we analyzed the quality of a subset selected using applications from one miss-rate group and tested it with applications from a different miss-rate group. The data cache results revealed significant increases in energy consumption for the test set applications as compared to T_3. For example, executing high miss-rate test applications using a subset selected using low and mid-range miss-rate training applications *increased* the energy consumption by as much as 640% and 190%, respectively, as compared to T_3, while executing high miss-rate test applications with a subset selected using high miss-rate training applications *reduced* energy consumption by 20% as compared to T_3. The instruction cache results revealed a smaller increase in energy consumption. For example, executing high miss-rate test applications with a subset selected using low and mid-range miss-rate training applications increased the energy consumption by as much as 290% and 130%, respectively, as compared to T_3.

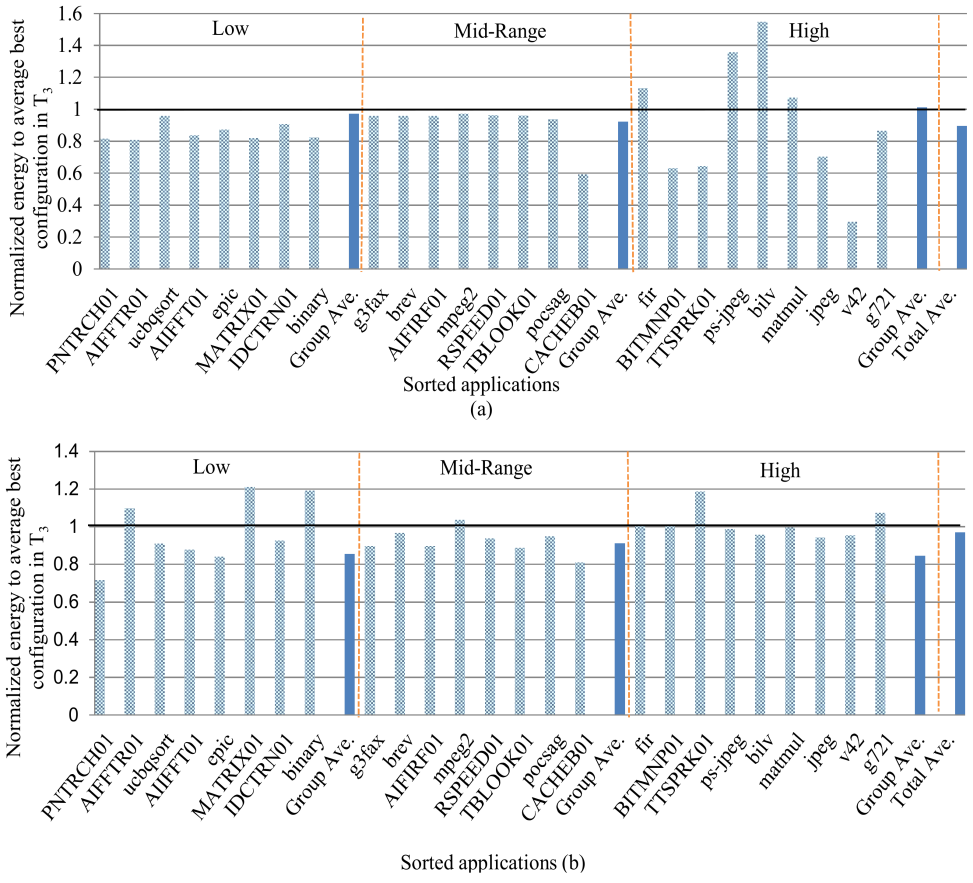

Figure 5. The instruction (**a**) and data (**b**) cache energy consumptions normalized to the average best configuration for each test application in T_3. The applications are sorted in ascending cache miss-rate order from left to right with demarcations showing the low, mid-range, and high miss-rate groups.

Figure 6 depicts the applications' energy consumptions using $s_b \in S$–selected using $T_{BC,3}$–and normalized to $c_b \in C$ with the baseline at 1.0. The applications are sorted in ascending cache miss-rate order from left to right with demarcations showing the low, mid-range, and high miss-rate

groups. The averages are shown for each group (group average) and across all groups (total average). A normalized energy over 1.0 is an increase in energy consumption as compared to the best configuration. For the low and mid-range miss-rate groups, the average energy increases were similar, 0% and 0.05%, respectively, but the energy increase was much higher for the high miss-rate group, 14.8%. The data cache high miss-rate group's average was similar, 15%, but the low and mid-range miss-rate groups' averages were much higher, 9.7% and 4.9%, respectively. Although the total average energy increase for the instruction cache was smaller than for the data cache—5.6% and 10.2%, respectively—the instruction cache's average energy range was 11% higher than the data cache's energy range: 0–62% versus 0–51%, respectively.

Additionally, we compared the average energy increases with subsets selected using $T_{BC,3}$ and T_3 as compared to $c_b \in C$. The analysis revealed that, on average, the subsets selected using $T_{BC,3}$ revealed 32.5% and 6.5% more energy savings when compared to the subsets selected using T_3, for the instruction and data caches, respectively.

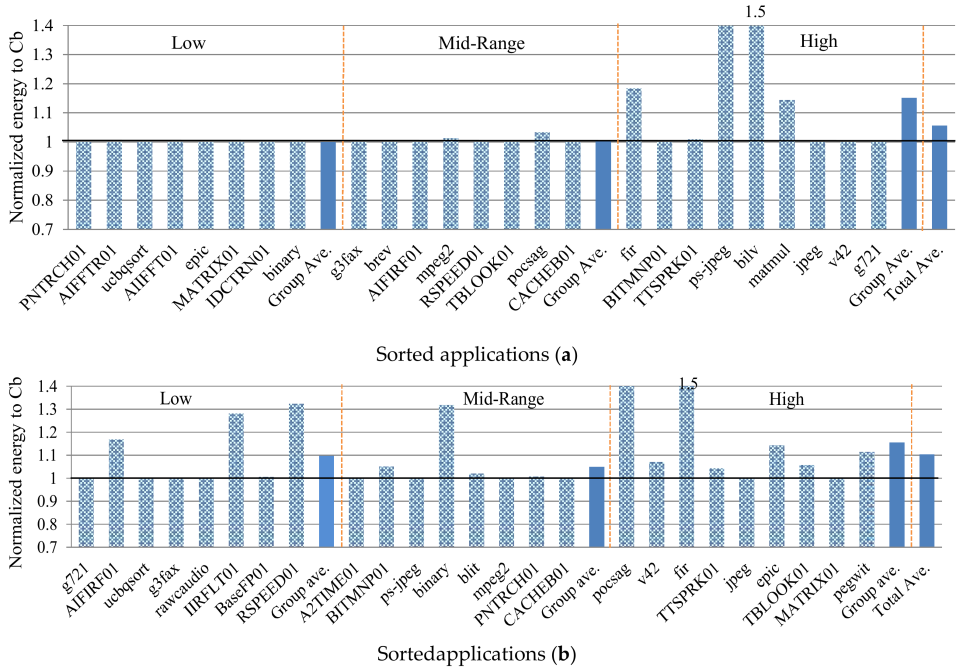

Figure 6. The instruction (**a**) and data (**b**) cache energy consumption normalized to the best configuration in the complete design space *C*. The applications are sorted in ascending cache miss-rate order from left to right with demarcations showing the low, mid-range, and high cache miss-rate groups.

7.2.2. Classification Method Impact

Since we used three classes in hierarchical classification to provide a fair comparison with basic classification, we compared the subset quality for each method using the subsets from these classes. Figure 7a,b depict the average energy consumption of the test applications *Qi* using $s_b \in S$–selected using $T_{HC,3}$–and normalized to the energy consumption of the test applications using $s_b \in S$–selected using $T_{BC,3}$–for the instruction and data caches, respectively. A value of 1.0 denotes the baseline for $T_{BC,3}$, where normalized energy below 1.0 represents an energy improvement as compared to $T_{BC,3}$.

The hierarchical classification's subsets both improved and degraded the energy consumption across different applications. However, on average, the energy consumption degraded by 21.3% and 13.1% for the instruction and data caches, respectively. For the three-class level in the dendrogram in Figure 2, the hierarchical clustering classified thirty applications in one class compared to the basic classification, which classified eleven of these applications in the low miss-rate class, eleven in the mid-range miss-rate class, and the remaining eight in the high miss-rate class. Hierarchical clustering

resulted in one subset adhering to thirty applications' disparate hardware requirements, while the basic classification resulted in three subsets to achieve similar adherence.

To validate the class size's impact on subset quality, we evaluated the quality of a subset of a small class using disjoint applications from a larger class. We evaluated the subset S—selected using $T_{BC,3}$—for the low miss-rate class using disjoint test applications from the 1st class of $T_{HC,3}$ (i.e., the 27 disjoint applications). Figure 8a,b depict the instruction and data cache's energy consumption, respectively, for these test applications using $s_b \in S$–selected using $T_{HC,3}$–and normalized to the energy consumption of $s_b \in S$–selected using $T_{BC,3}$–for the low-range class. On average, the hierarchical classification subset resulted in a 2.9% and 17.9% improvement in the energy consumption for the instruction and data caches, respectively. Since the basic classification used only three training applications to determine the S–selected using $T_{BC,3}$–for the low miss-rate class, the quality of this subset is high only for the test application from the low miss-rate class.

Additionally, comparing this analysis to our analysis in Section 7.1, we observe that the subset quality does not necessarily deteriorate when used for larger application classes. A specific training set (e.g., the three training applications when using basic classification) resulted in a high-quality subset for test applications Q_i of the same class, but in very low quality subsets for test applications Q_i from other classes (e.g., the mid-range or high miss-rate classes). However, a subset determined by hierarchical classification for the 1st class was of higher quality for a larger number of applications (30 applications). This is due to the hierarchical classification's intrinsic of grouping applications with similar hardware requirements (i.e., '*single distance*'). Thus, the classification method that dictates the similarity of application hardware requirements, rather than the class size, has a higher impact on subset quality. Basic classification results in high-quality subsets per class, for small classes (e.g., the ten-application class). However, hierarchical classification produced a higher quality subset for a larger class (e.g., the thirty-application class), compared to the basic classification.

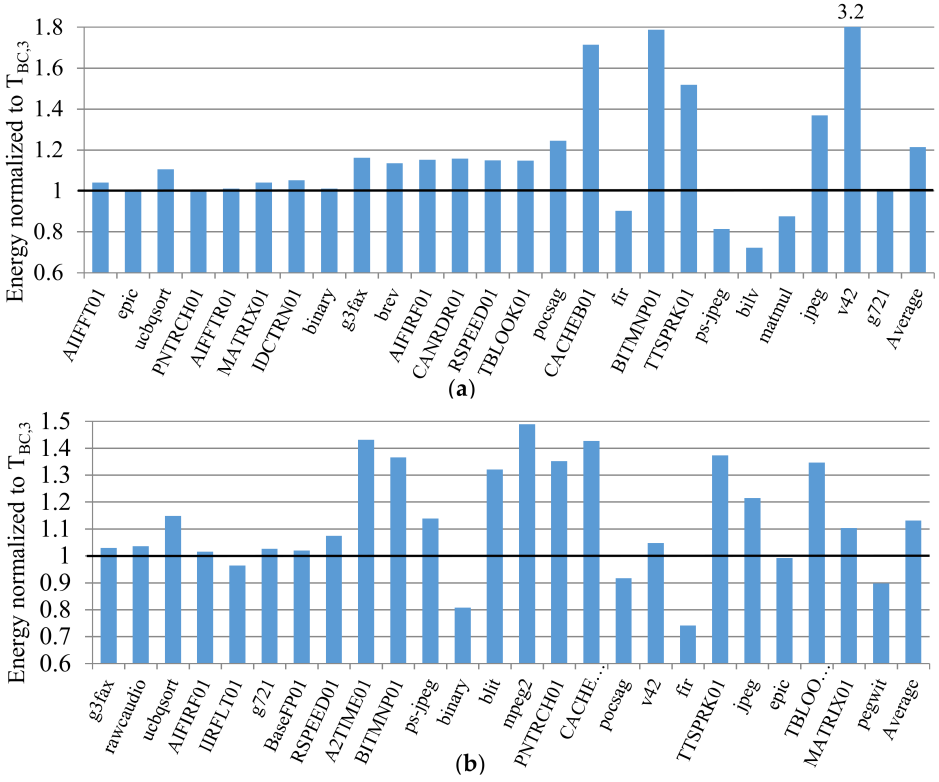

Figure 7. The average energy consumption for the *Qi* of test application using $s_b \in S$–selected using $T_{HC,3}$–and normalized to the energy consumption of the test applications using $s_b \in S$–selected using $T_{BC,3}$–for the instruction (**a**) and data cache (**b**).

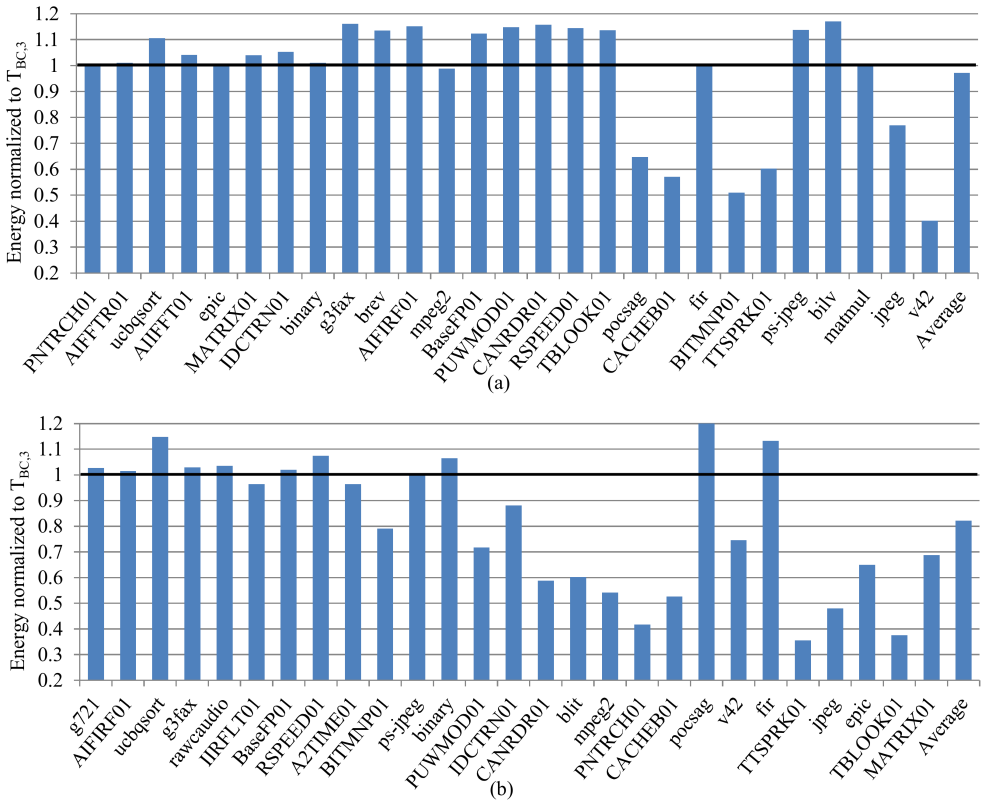

Figure 8. The instruction (**a**) and data (**b**) cache's energy consumption, for the test applications using $s_b \epsilon S$–selected using $T_{HC,3}$–and normalized to the energy consumption of $s_b \epsilon S$–selected using $T_{BC,3}$–for the low miss-rate class.

8. Speedup Results and Analysis

To quantify the design-time effort, we evaluated the speedup of our subset selection algorithm, as compared to prior work (T_{34}), the impact of the subset size on the performance of our algorithms, the overhead associated with our classification algorithms, and the subset quality-speedup tradeoff. We performed our speedup analysis of our algorithms for the data and instruction caches. However, we obtained identical results, since the execution times of our algorithms are dependent on the training application set size and not the type of cache (instruction or data). For brevity, we discuss the results of our algorithm's speedup analysis of the data cache.

8.1. Subset Selection Speedup

Since T_3 provided the best tradeoff between the training set size and energy savings, we performed our speedup analysis on subsets of the size 3: ρT_3, $T_{BC,3}$, and $T_{HC,3}$. To compare the performance of Algorithm 1 to prior work, we used the execution time of Algorithm 1 to obtain the configuration subsets for ρT_3, $T_{BC,3}$, and $T_{HC,3}$ and normalized the values to the execution time of prior work. Figure 9 depicts the normalized results, where a value of 1.0 denotes the baseline execution time and the normalized execution times above 1.0 represent a speedup, compared to prior work. The results revealed that using Algorithm 1 explored the subset design space and determined a subset as much as 6.6, 6.5, and 6.1X faster for ρT_3, $T_{BC,3}$, and $T_{HC,3}$, respectively, compared to prior work. Since Algorithm 1 used an application-configuration energy matrix as an input and calculated the average energy increase μ_Δ for all applications in that matrix, the number of training applications evaluated had a large impact on Algorithm 1's execution time.

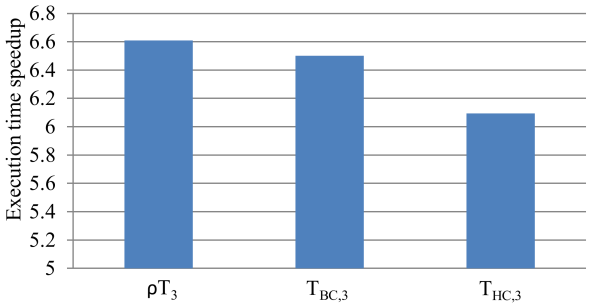

Figure 9. The execution time of Algorithm 1 for ρT_3, $T_{BC,3}$, and $T_{HC,3}$, normalized to the execution time of prior work.

Furthermore, to evaluate the performance variations of Algorithm 1 for ρT_3, $T_{BC,3}$, and $T_{HC,3}$, we normalized the execution times of Algorithm 1 for $T_{BC,3}$ and $T_{HC,3}$ to the execution time of Algorithm 1 for ρT_3. The results revealed that Algorithm 1 explored the subset design space 1.69% and 8.47% slower for $T_{BC,3}$ and $T_{HC,3}$, respectively, compared to ρT_3. Our code instrumentation accounted the discrepancies to the overhead required by Algorithm 1 to navigate the application-configuration energy matrix to select the training set applications. Although Algorithm 1 needed three applications for each execution, randomly selecting three training applications required less time than specified applications in a given class/group. We emphasize that these variations are negligible compared to the speedup obtained, as compared to prior work.

To further evaluate the training set size impact on the speedup, we analyzed the execution time of Algorithm 1 to select subsets of various sizes. Since the speedups of ρT_3, $T_{BC,3}$, and $T_{HC,3}$ are similar, analyzing the speedup of all $\rho T_{(i)}$, where $i \in [1–10, 17]$ also provides insight into the speedup for $T_{BC,i}$ and $T_{HC,i}$. We normalized the execution times of Algorithm 1 with $\rho T_{(i)}$ for $i \in [1–10, 17]$ to prior work and evaluated the correlation between the training application set size and speed up.

Figure 10 depicts the speedups for Algorithm 1, using varying training set sizes normalized to the execution time of prior work. For sets with one application (T_1 and 2nd class in $T_{HC,3}$), Algorithm 1 used one application and explored the design space 7.8X faster as compared to using the complete application set A. Alternatively, for sets with 17 applications, Algorithm 1 explored the subset design space and determined a subset as much as 1.5X faster, compared to using the complete application set A. Furthermore, the correlation analysis revealed a -0.90 correlation factor between the set size and the speedup; the smaller the size, the higher the speedup; and that the correlation is not perfectly linear and saturates for training sets with sizes above 17. This result suggests that for a potentially very large design space, a significant speedup can be obtained for training sets that are less than half the size of the entire application set. However, using a very small training set size can result in lower quality configuration subsets (Section 7.2).

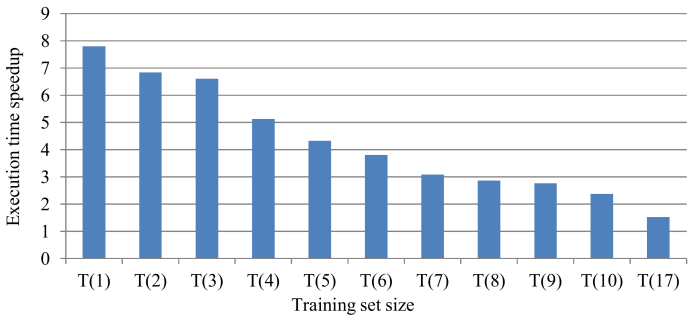

Figure 10. The execution time of Algorithm 1 for ρT_i, for $i \in [1–10, 17]$, normalized to the execution time of prior work.

8.2. Classification Algorithm Speedup

Since prior work did not do any classification, we juxtaposed the time it required for our basic and hierarchical (Algorithm 2) classification algorithms to execute and classify the applications. Figure 11 depicts the raw average execution times in milliseconds for the basic classification and hierarchical classification to select three training applications that are used in Algorithm 1 as the training application. We used a machine with an x86 core that ran with a clock speed of 1.2 GHz and performed the measurement ten times and calculated the average execution time. The results revealed that, on average, Algorithm 2 required 62 ms to execute while the basic classification required 10 ms. Comparatively, hierarchical classification is 6.2X slower than basic classification. Although both algorithms select 3 applications out of 34, the basic classification algorithm required a shorter execution time since basic classification only sorts the applications once, while Algorithm 2 iteratively merges applications into classes of the closest Euclidean distance.

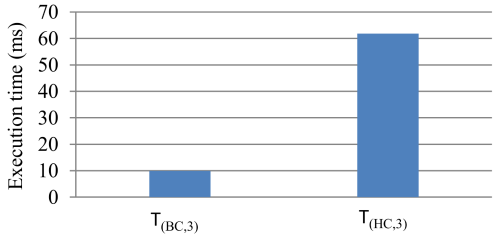

Figure 11. The execution time in milliseconds for the basic and hierarchical classification algorithms to select three training applications.

This result suggested that the distance/classification metric (Euclidean, city-block, Chebychev, etc.) had an impact on Algorithm 2. Subsequently, Algorithm 2 is likely to execute slower for a design space with higher dimensions (application performance, code density, compatibility, etc.) that require a high-dimension distance metric.

8.3. Classification Methods Speedup versus Subset Quality

Our experiments also proved that a design space subset can be characteristically determined, rather than application-specifically determined, thus, easing subset selection based on generalized characteristics rather than application specifics. Alternatively, hierarchical clustering required the application miss-rates in order to calculate the distance between application hardware requirements and subsequently classify the applications. However, hierarchical classification can determine the cutoff miss-rate for each class using application traces, mini-applications, kernel applications, etc. and thus, no a priori knowledge of *exact* applications is required.

These basic classification training set results illustrate that a priori knowledge of the general application domain's miss-rate characteristics can significantly enhance subset quality. We emphasize that exact miss-rate analysis is not required and a simple miss-rate range classification (i.e., low, mid-range, high) shows appreciable subset quality improvements with greater improvement potential if the designer has more anticipated application characteristic information. Algorithm 2, for instance, demonstrated a higher subset quality, with an average of 17.9% (Figure 8), at the cost of a slower execution time, as much as 6.2X (Figure 11), as compared to the basic classification algorithm.

Since classification metrics have the largest impact on Algorithm 2 (Section 8.2), Algorithm 2 will require a longer execution time for design spaces with high dimensions, compared to the basic classification algorithm. However, high dimensional design spaces represent more disparate hardware requirements that could not be realized with the basic classification algorithm and hierarchical classification produced a higher quality subset for a large set of applications that had disparate hardware requirements compared to the basic classification (Section 7.2.2).

The subset-quality-execution time tradeoff provides the designer with options to meet the required design constraints.

9. Conclusions and Future Work

Configurable caches enable the cache hierarchy to adapt to varying application-specific requirements to reduce energy consumption. However, highly configurable caches which provide a myriad of cache configurations for highly disparate application requirements, require potentially prohibitive design space exploration time. To reduce the design space exploration time, we evaluated cache configuration design space subsetting using a set of training applications and evaluated the subsets' qualities in terms of energy savings potential using a disjoint test set rather than the complete, unsubsetted design space. Our results showed that randomly-selected application training sets can significantly reduce the design space to small, high-quality configuration subsets, and using basic classification-based training sets can increase the subsets' qualities. These analyses enable designers to harness the large energy savings potential of configurable caches for both general and domain-specific application requirements, with minimal design-time effort and only a general knowledge of the anticipated applications.

Our results also revealed that hierarchical classification (Algorithm 2) requires longer execution time. However, it is favorable when hardware resources are scarce; one subset can adhere to more application's hardware requirements, compared to the basic classification. Whereas both classification methods require application miss-rates to determine the application's class, both algorithms alleviate the necessity of profiling new, unknown applications for all configurations in the design space; a simple profiling on a base configuration, for instance, provides a basic knowledge of the application's hardware requirements, and thus the application's class.

Future work includes investigating different classification methods that consider application behavior, such as computation and data movements (linear, n-body simulation, spectral methods, and so on) and additional application statistics, such as static code analysis, instruction count, clock cycles, and so on. Since the order of which the algorithm traverses the design space dictates the pruned configurations, using these additional statistics in a multidimensional space (i.e., statistic per dimension space), we plan to investigate the impact of design space exploration algorithms in determining Pareto optimal subsets.

Acknowledgments: This work was support in part by the National Science Foundation (CNS-0953447). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

Author Contributions: F Mohamad Hammam Alsafrjalani and Ann Gordon-Ross conceived of and designed the experiment. Mohamad Hammam Alsafrjalani performed the experiments and analyzed the results. Both authors wrote the article and approved the final manuscript.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Alsafrjalani, M.H.; Gordon-Ross, A. Dynamic Scheduling for Reduced Energy in Configuration-Subsetted Heterogeneous Multicore Systems. In Proceedings of the 2014 12th IEEE International Conference on Embedded and Ubiquitous Computing, Milano, Italy, 26–28 August 2014; pp. 17–24.
2. Albonesi, D.H. Dynamic IPC/clock rate optimization. In Proceedings of the 25th Annual International Symposium on Computer Architecture (Cat. No. 98CB36235), Barcelona, Spain, 27 June–2 July 1998; pp. 282–292.
3. Malik, A.; Moyer, B.; Cermak, D. A low power unified cache architecture providing power and performance flexibility. In Proceedings of the 2000 International Symposium on Low Power Electronics and Design, ISLPED '00, Rapallo, Italy, 25–27 July 2000; pp. 241–243.

4. Zhang, C.; Vahid, F.; Najjar, W. A highly configurable cache architecture for embedded systems. In Proceedings of the 30th Annual International Symposium on Computer Architecture, San Diego, CA, USA, 9–11 June 2003; pp. 136–146.
5. Chen, L.; Zou, X.; Lei, J.; Liu, Z. Dynamically Reconfigurable Cache for Low-Power Embedded System. In Proceedings of the Third International Conference on Natural Computation (ICNC 2007), Haikou, China, 24–27 August 2007; pp. 180–184.
6. Palesi, M.; Givargis, T. Multi-objective design space exploration using genetic algorithms. In Proceedings of the Tenth International Symposium on Hardware/Software Codesign. CODES 2002 (IEEE Cat. No. 02TH8627), Estes Park, CO, USA, 8 May 2002; pp. 67–72.
7. Zhang, C.; Vahid, F. Cache configuration exploration on prototyping platforms. In Proceedings of the 14th IEEE International Workshop on Rapid Systems Prototyping, San Diego, CA, USA, 9–11 June 2003; pp. 164–170.
8. Ghosh, A.; Givagis, T. Cache optimization for embedded processor cores: An analytical approach. In Proceedings of the ICCAD-2003. International Conference on Computer Aided Design (IEEE Cat. No. 03CH37486), San Jose, CA, USA, 9–13 November 2003; pp. 342–347.
9. Gordon-Ross, A.; Viana, P.A.; Vahid, F.; Najjar, W.; Barros, E. A One-Shot Configurable-Cache Tuner for Improved Energy and Performance. In Proceedings of the 2007 Design, Automation & Test in Europe Conference & Exhibition, Nice, France, 16–20 April 2007; pp. 1–6.
10. Wang, W.; Mishra, P. Dynamic Reconfiguration of Two-Level Caches in Soft Real-Time Embedded Systems. In Proceedings of the 2009 IEEE Computer Society Annual Symposium on VLSI, Tampa, FL, USA, 13–15 May 2009; pp. 145–150.
11. Viana, P.; Gordon-Ross, A.; Keogh, E.; Barros, E.; Vahid, F. Configurable cache subsetting for fast cache tuning. In Proceedings of the 2006 43rd ACM/IEEE Design Automation Conference, San Francisco, CA, USA, 24–28 July 2006; pp. 695–700.
12. Keogh, E.; Chu, S.; Hart, D.; Pazzani, M. An online algorithm for segmenting time series. In Proceedings of the 2001 IEEE International Conference on Data Mining, San Jose, CA, USA, 29 November–2 December 2001; pp. 289–296.
13. Khan, O.; Kundu, S. A model to exploit power-performance efficiency in superscalar processors via structure resizing. In Proceedings of the 20th ACM Great Lakes Symposium on VLSI (GLSVLSI '10), New York, NY, USA, 16–18 May 2010; pp. 215–220.
14. Adegbija, T.; Gordon-Ross, A. Phase-Based Dynamic Instruction Window Optimization for Embedded Systems. In Proceedings of the 2016 IEEE Computer Society Annual Symposium on VLSI (ISVLSI), Pittsburgh, PA, USA, 11–13 July 2016; pp. 397–402.
15. Herbert, S.; Garg, A.; Marculescu, D. Exploiting Process Variability in Voltage/Frequency Control. *IEEE Trans. Large Scale Integr. (VLSI) Syst.* **2012**, *20*, 1392–1404. [CrossRef]
16. Ernst, D.; Kim, N.S.; Das, S.; Pant, S.; Rao, R.; Pham, T.; Ziesler, C.; Blaauw, D.; Austin, T.; Flautner, K.; et al. Razor: A low-power pipeline based on circuit-level timing speculation. In Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-36, San Diego, CA, USA, 3–5 December 2003; pp. 7–18.
17. Rangan, K.; Wei, G.; Brooks, D. Thread motion: Fine-grained power management for multi-core systems. *SIGARCH Comput. Archit. News* **2009**, *37*, 302–313. [CrossRef]
18. Semeraro, G.; Magklis, G.; Balasubramonian, D.; Albonesi, S.; Dwarkadas, H.; Scott, M. Energy-efficient processor design using multiple clock domains with dynamic voltage and frequency scaling. In Proceedings of the Eighth International Symposium on High Performance Computer Architecture, Boston, MA, USA, 2–6 February 2002; pp. 29–40.
19. Silvano, C.; Palermo, G.; Xydis, S.; Stamelakos, I. Voltage island management in near threshold manycore architectures to mitigate dark silicon. In Proceedings of the 2014 Design, Automation & Test in Europe Conference & Exhibition (DATE), Dresden, Germany, 24–28 March 2014; pp. 1–6.
20. Jejurikar, R.; Pereira, C.; Gupta, R. Leakage aware dynamic voltage scaling for real-time embedded systems. In Proceedings of the 41st Design Automation Conference, San Diego, CA, USA, 7–11 July 2004; pp. 275–280.
21. Mantovani, P.; Cota, E.G.; Tien, K.; Pilato, C.; Di Guglielmo, G.; Shepard, K.; Carloni, L.P. An FPGA-based infrastructure for fine-grained DVFS analysis in high-performance embedded systems. In Proceedings of the 53rd ACM Annual Design Automation Conference (DAC '16), Austin, TX, USA, 5–9 June 2016.

22. Folegnani, D.; Gonzalez, A. Energy-effective issue logic. In Proceedings of the 28th Annual International Symposium on Computer Architecture, Goteborg, Sweden, 30 June–4 July 2001; pp. 230–239.
23. Ishihara, T.; Yasuura, H. Voltage scheduling problem for dynamically variable voltage processors. In Proceedings of the 1998 International Symposium on Low Power Electronics and Design (IEEE Cat. No. 98TH8379), Monterey, CA, USA, 10–12 August 1998; pp. 197–202.
24. Patel, K.; Benini, L.; Macii, E.; Poncino, M. Reducing Conflict Misses by Application-Specific Reconfigurable Indexing. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* **2006**, *25*, 2626–2637. [CrossRef]
25. Zhang, C.; Vahid, F.; Najjar, W. Energy benefits of a configurable line size cache for embedded systems. In Proceedings of the IEEE Computer Society Annual Symposium on VLSI, Tampa, FL, USA, 20–21 February 2003; pp. 87–91.
26. Biglari, M.; Barijough, K.M.; Goudarzi, M.; Pourmohseni, B. A fine-grained configurable cache architecture for soft processors. In Proceedings of the 2015 18th CSI International Symposium on Computer Architecture and Digital Systems (CADS), Tehran, Iran, 7–8 October 2015; pp. 1–6.
27. Chiou, D.; Devadas, S.; Rudolph, L.; Ang, B. Dynamic Cache Partitioning via Columnization. In Proceedings of the MIT Computation Structures Group Memo 430, Cambridge, MA, USA, November 1999.
28. Givargis, T.; Vahid, F. Platune: A tuning framework for system-on-a-chip platforms. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* **2002**, *21*, 1317–1327. [CrossRef]
29. Palermo, G.; Silvano, C.; Zaccaria, V. ReSPIR: A response surface-based Pareto iterative refinement for application-specific design space exploration. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* **2009**, *28*, 1816–1829. [CrossRef]
30. Gordon-Ross, A.; Vahid, F.; Dutt, N. Automatic tuning of two-level caches to embedded applications. In Proceedings of the Design, Automation and Test in Europe Conference and Exhibition, Paris, France, 16–20 February 2004; pp. 208–213.
31. Gordon-Ross, A.; Vahid, F. Fast Configurable-Cache Tuning with a Unified Second-Level Cache. *IEEE Trans. Large Scale Integr. (VLSI) Syst.* **2009**, *17*, 80–91. [CrossRef]
32. Rawlins, M.; Gordon-Ross, A. An application classification guided cache tuning heuristic for multi-core architectures. In Proceedings of the 17th Asia and South Pacific Design Automation Conference, Sydney, Australia, 30 January–2 February 2012; pp. 23–28.
33. EEMBC. The Embedded Microprocessor Benchmark Consortium. Available online: https://www.eembc.org/benchmark/automotive_sl.php (accessed on 19 December 2017).
34. Mediabench Consortium. Available online: http://euler.slu.edu/~fritts/mediabench/ (accessed on 19 December 2017).
35. Reinman, G.; Jouppi, N.P. COMPAQ Western Research Lab: CACTI2.0: An Integrated Cache Timing and Power Model. 1999. Available online: http://www.hpl.hp.com/research/cacti/cacti2.pdf (accessed on 19 December 2017).
36. Burger, D.; Austin, T.M.; Bennet, S. *Evaluating Future Microprocessors: The Simplescalar Tools Set*; Technical Report CS-TR-1996-1308, CS; University of Wisconsin-Madison, Computer Sciences Department: Madison, WI, USA, 1996.