A.2 Full Implementation Explanation

A.2.1 Command-Line Interface Implementation

To allow the measurement of benchmarks, we implement a way to communicate with the program with the help of a Command-Line Interface (CLI).

```
1 public static class CsharpRAPLCLI {
2     public static Options Options { get; private set; } = new();
3     private static Action<Analysis.Analysis> _analysisCallback =
      ↪  Analyse;
4     private static Action<string>? _plotCallback;
5     ...
6 }
```

Listing 31: CLI Class Fields.

In Listing 31 we see the declaration of the `CsharpRAPLCLI` class containing the logic for communicating with the CLI. We see the fields that exists in this class. The `Options` field contains the options that the user can choose when running the program, for example removing old results, analysing previous results instead of running the benchmarks, analysing new results, etc. The `_analysisCallback` field contains an `Action` that is executed after all benchmarks have been run. This callback allows the user to set up custom automatic analysis, besides the automatic analysis shown in Section 5.3. The `_plotCallback` field contains an `Action` that is executed after all benchmarks has been executed. The `_plotCallback` is created to allow the user to set up custom plots, like bar graphs, pie charts, line graphs etc.

```
1 public static class CsharpRAPLCLI {
2     ...
3     public static Options Parse(string[] args, int
      ↪  maximumTerminalWidth = 0) {
4         var parser = new Parser(settings => {
5             settings.CaseSensitive = false;
6             settings.HelpWriter = Console.Out;
7             if (maximumTerminalWidth != 0) {
```

```
8                   settings.MaximumDisplayWidth =
                    ↪ maximumTerminalWidth;
9               }
10          });
11          parser.ParseArguments<Options>(args).
            ↪ WithParsed(RunOptions).WithNotParsed(HandleParseError);
12          ...
13      }
14      ...
15  }
```

Listing 32: Setting up the parser in the `Parse` Method.

In Listing 32 we can see the `Parse` method. Specifically, we can see how we setup a `Parser` from the CommandLine library [134]. We set the help writer to `Console.Out` to ensure the help command can be written to the user. After applying the settings, we give the arguments provided to the parser. If the parser succeeds in parsing the arguments, the `RunOptions` method is called, otherwise the `HandleParseError` method is called. These methods are seen in Listing 35 and Listing 36.

```
1   public static class CsharpRAPLCLI {
2       ...
3       public static Options Parse(string[] args, int
        ↪ maximumTerminalWidth = 0) {
4           ...
5           if (Options.OnlyPlot) {
6               if (_plotCallback == null) {
7                   BenchmarkPlot.
                    ↪ PlotAllResultsGroupsFromFolder(Options.OutputPath);
8               }
9               else {
10                  _plotCallback.Invoke(Options.OutputPath);
11              }
12              Options.ShouldExit = true;
13              return Options;
14          }
15          if (Options.OnlyAnalysis) {
```

```
16              StartAnalysis(Options.
                ↪  BenchmarksToAnalyse.ToArray());
17              Options.ShouldExit = true;
18          }
19          return Options;
20      }
21      ...
22  }
```

Listing 33: Checking specific options in the `Parse` Method. Continued from Listing 32.

In Listing 33 we can see the rest of the `Parse` method. This part of the `Parse` method serves to check if the options `OnlyPlot` or `OnlyAnalysis` has been set to `true`. If `OnlyPlot` is `true` the benchmarks results are used for creating plots. In the same manner if `OnlyAnalysis` is `true` the results are used for analysis. If neither is set to `true`, neither method is called and we go straight to the end of the method. If neither is set to `true`, it means that the `ShouldExit` variable is not set to true and therefore the benchmarks are run and we will gather new results. The `Options` object is returned, as this contains data for further usage.

```
1   public class Options {
2       ...
3       [Option('r', nameof(RemoveOldResults), Required = false,
    ↪   HelpText = "If set removes all files from the output folder
    ↪   and the plot folder.")]
4       public bool RemoveOldResults { get; set; }
5       ...
6       [Option(nameof(OnlyAnalysis), Required = false, HelpText =
    ↪   "Analysis the results in the output path.")]
7       public bool OnlyAnalysis { get; set; }
8       ...
9       public bool ShouldExit;
10  }
```

Listing 34: The `Options` Class.

In Listing 34 we can see a part of the `Options` class. This is a data class that includes fields that are changed by the `parser` in Listing 32. Addition-

ally, the class includes fields used by the program to keep track of default values and the state of the program. As the `parser` parses arguments, this means that the user can decide the values of some fields in the `Options` class. The fields which are mutable by the `parser` have an attribute called `Option`, containing relevant info for the field. The `ShouldExit` field is only used if an error occurs, or if the user wants to plot or analyse the results without running the benchmarks.

```
public static class CsharpRAPLCLI {
    ...
    private static void RunOptions(Options opts) {
        Options = opts;
        Options.OutputPath = Options.OutputPath.Replace("\\",
        ↪  "/");
        if (!Options.OutputPath.EndsWith("/")) {
            Options.OutputPath += "/";
        }
        Options.PlotOutputPath =
        ↪  Options.PlotOutputPath.Replace("\\", "/");
        if (!Options.PlotOutputPath.EndsWith("/")) {
            Options.PlotOutputPath += "/";
        }
        if (!Options.RemoveOldResults) {
            return;
        }
        if (Directory.Exists(Options.PlotOutputPath)) {
            Directory.Delete(Options.PlotOutputPath, true);
        }
        if (Directory.Exists(Options.OutputPath)) {
            Directory.Delete(Options.OutputPath, true);
        }
    }
    ...
}
```

Listing 35: The `RunOptions` method that is called if the arguments are parsed.

In Listing 35 we see the method that is called when the arguments can be parsed by the `parser`. Here the options arguments are sanitized so they

work properly despite having two backslashes in the paths. Furthermore, old results are deleted if the *RemoveOldResults* option has been set to `true`.

```
public static class CsharpRAPLCLI {
    ...
    private static void HandleParseError(IEnumerable<Error>
    ↪ errs) {
        foreach (Error error in errs) {
            if (error.Tag is ErrorType.HelpRequestedError or
            ↪ ErrorType.VersionRequestedError) {
                Options.ShouldExit = true;
            }
            else {
                throw new
                ↪ NotSupportedException(ParseError(error));
            }
        }
    }
    ...
}
```

Listing 36: The `HandleParseError` method that is called if the arguments can not be parsed.

In Listing 36 we can see the method that is called if the arguments cannot be parsed. Here we go through the errors that are found and check what type of error it is. If the error has the tag `HelpRequestedError` or `VersionRequestedError`, we do not need to do anything besides exit the program, meaning we set the `ShouldExit` field to `true`. The `HelpRequestedError` tag is set when using the `help` command and the `VersionRequestedError` tag is set when using the `version` command. If the error does not have this tag, we throw a `NotSupportedException` with a string returned by the `ParseError` method.

```
public static class CsharpRAPLCLI {
    ...
    private static string ParseError(Error error) {
```

```
        switch (error) {
            case BadFormatTokenError badFormatTokenError:
                return $"Token '{badFormatTokenError.Token}' is
                ↪ not recognized.";
            ...
        }
        throw new ArgumentOutOfRangeException($"{error.Tag} is
        ↪ out of range.");
    }
    ...
}
```

Listing 37: The `ParseError` method that is called to parse an error to a string.

In Listing 37 we can see how errors are parsed to a `string`. In the `switch` statement, the type of the error is checked and the corresponding `string` is returned. If none of the errors fit the type, an `ArgumentOutOfRangeException` is thrown.

```
public static class CsharpRAPLCLI {
    ...
    public static void StartAnalysis(Dictionary<string,
    ↪ List<IBenchmark>> benchmarksWithGroups) {
        foreach (string group in benchmarksWithGroups.Keys) {
            ...
            for (int i = 0; i <
            ↪ benchmarksWithGroups[group].Count; i++) {
                for (int j = i + 1; j <
                ↪ benchmarksWithGroups[group].Count; j++) {
                    var analysis =
                        new Analysis.Analysis(
                        ↪ benchmarksWithGroups[group][i],
                        ↪ benchmarksWithGroups[group][j]);
                    _analysisCallback.Invoke(analysis);
                }
            }
        }
    }
```

```
    ...
}
```

Listing 38: The `StartAnalysis` method which sets up the results of the benchmarks to be analysed using the `analysisCallback`.

In Listing 38 we can see how an analysis is started with the analysis callback. We go through all of the groups we have gathered results from and invoke the analysis callback, which can be defined by the user, on every pair of benchmarks in the group. This does not include the analysis done automatically by the program after running the benchmarks.

```
public static class CsharpRAPLCLI {
    ...
    public static void StartAnalysis(string[] thingsToAnalyse) {
        ...
        foreach (string[] chunk in thingsToAnalyse.Chunk(2)) {
            string firstPath = chunk[0];
            string secondPath = chunk[1];
            if (!firstPath.EndsWith(".csv")) {
                firstPath = GetMostRecentFile(firstPath);
            }
            if (!secondPath.EndsWith(".csv")) {
                secondPath = GetMostRecentFile(secondPath);
            }
            ...
            var analysis = new Analysis.Analysis(firstPath,
            ↪  secondPath);
            _analysisCallback.Invoke(analysis);
        }
    }
    ...
}
```

Listing 39: The `StartAnalysis` method that sets up the results of benchmarks that has previously been run to be analysed using the `analysisCallback`.

In Listing 39 we can see that an analysis can also be called on results saved in a *.csv* file. This is done by calling the `StartAnalysis` method with an array of paths to *.csv* files, this array is then chunked into pairs of 2 so they can be analysed against each other. The most recent *.csv* files in the paths that have been sent are found and used to create an `Analysis` object, which is then used on the analysis callback.

```
1  public static class CsharpRAPLCLI {
2      ...
3      private static string GetMostRecentFile(string path) {
4          const string pattern = "*.csv";
5          var dirInfo = new
               ↪ DirectoryInfo($"{Options.OutputPath}/{path}");
6          FileInfo file = (from f in dirInfo.GetFiles(pattern)
               ↪ orderby f.LastWriteTime descending select f)
7              .First();
8          return $"{path}/{file.Name}";
9      }
10     ...
11 }
```

Listing 40: The `GetMostRecentFile` method that gets the newest .csv file from a path.

In Listing 40 we can see how we get the latest *.csv* file from a path. Specifically, on line 5 we order all the files in the directory sent to the method by `LastWriteTime` and select the first file. This is done so we get the relevant result even when not deleting the previous results from the results folder.

A.2.2 Measurement Implementation

To run benchmarks and measure the runtime, package and DRAM energy we build upon previous work done in [9]. Specifically, we build on their Running Average Power Limit (RAPL) library for C#.

Running All Benchmarks

To make it possible to run benchmarks without having to run each benchmark individually, we create a class called `BenchmarkSuite` to setup the

prerequisites for running benchmarks, and then running them.

```
public class BenchmarkSuite {
    public int Iterations { get; }
    public int LoopIterations { get; }
    private List<IBenchmark> Benchmarks { get; } = new();
    private readonly HashSet<Type> _registeredBenchmarkClasses =
    ↪ new();
    private const string LoopIterationsName =
    ↪ nameof(LoopIterations);
    private const string IterationsName = nameof(Iterations);
    ...
}
```

Listing 41: `BenchmarkSuite` class fields.

In Listing 41, we see the fields in the `BenchmarkSuite` class. The first two fields, `Iterations` and `LoopIterations`, determines how many times a benchmark runs, and how many times the code inside the loop runs, respectively. For example, we may need a benchmark to be run 30 times to get a significant result, while the code in the benchmark may need to be run 1.000.000 times to run for longer than 0,25 seconds.

The field `Benchmarks` is a list that contains the benchmarks that should be run. The field `_registeredBenchmarkClasses` contains the classes that have the benchmarks. This is to make it possible to set the `Iterations` and `LoopIterations` fields in these classes.

The last two fields are used to set the fields with the correct name in the benchmark classes, as the benchmark classes themselves have a field to keep track of `Iterations` and `LoopIterations`, as seen in Listing 43.

```
public class BenchmarkSuite {
    ...
    public void RegisterBenchmark<T>(string? group, Func<T>
    ↪ benchmark, int order = 0) {
        ...
        if (!_registeredBenchmarkClasses.Contains(
        ↪ benchmark.Method.DeclaringType!)) {
            RegisterBenchmarkClass(
            ↪ benchmark.Method.DeclaringType!);
```

```
7           }
8           Benchmarks.Add(new Benchmark<T>(benchmark.Method.Name,
            ↪  Iterations, benchmark, true, group, order));
9       }
10      ...
11  }
```

Listing 42: Registering generic benchmarks in the `RegisterBenchmark` method.

In Listing 42 we see the `RegisterBenchmark` method. This method is generic, which allows benchmarks to have different return types, instead of being constrained to a single return type.

We can see that if the `_registeredBenchmarkClasses` HashSet does not contain the class that declares the benchmark, we call the `RegisterBenchmarkClass` method with the class.

After this, we create a `Benchmark` object with the relevant information and add it to the `Benchmarks` list.

```
1   public class BenchmarkSuite {
2       ...
3       private void RegisterBenchmarkClass(Type benchmarkClass) {
4           TrySetField(benchmarkClass, IterationsName, Iterations);
5           TrySetField(benchmarkClass, LoopIterationsName,
            ↪  LoopIterations);
6           _registeredBenchmarkClasses.Add(benchmarkClass);
7       }
8       ..
9   }
```

Listing 43: Registering a new benchmark class in the `RegisterBenchmarkClass` method.

In Listing 43 we see the `RegisterBenchmarkClass` method. This method sets up and registers a class that has benchmarks in it. It does this by calling the `TrySetField` method with the `Iterations` and `LoopIterations`, and then afterwards add the class to the HashSet of classes containing benchmarks.

```
1   public class BenchmarkSuite {
2       ...
```

```
3       public static bool TrySetField(Type benchmarkClass, string
        ↪  name, int value) {
4           FieldInfo? fieldInfo =
            ↪  benchmarkClass.GetFields().FirstOrDefault(info =>
            ↪  info.Name == name);
5           if (fieldInfo == null) {
6               return false;
7           }
8           if (!fieldInfo.IsStatic) {
9               throw new NotSupportedException($"Your {name} field
                ↪  must be static.");
10          }
11          benchmarkClass.GetField(name, BindingFlags.Public |
            ↪  BindingFlags.Static)?.SetValue(null, value);
12          return true;
13      }
14      ...
15  }
```

Listing 44: Setting a field in a class containing benchmarks with the `TrySetField` method.

In Listing 44 we can see the `TrySetField` method. This method tries to set a field in the provided class that contains benchmarks. We do this by trying to find the field with the name provided, checking if the field exists and is static. Afterwards, the value is set to the provided parameter. If the field does not exist, `false` is returned to show that it has failed in setting the field. If the field does exist but is not static, an `Exception` is thrown, as the field must be static. After the field has been set, `true` is returned to show that it succeeded in setting the field.

```
1   public class BenchmarkSuite {
2       ...
3       public void RunAll() {
4           if (Environment.OSVersion.Platform != PlatformID.Unix) {
5               throw new NotSupportedException("Running the
                ↪  benchmarks is only supported on Unix.");
6           }
```

```
7           List<IBenchmark> benchmarks =
            ↪ Benchmarks.OrderBy(benchmark =>
            ↪ benchmark.Order).ToList();
8           Warmup();
9           foreach ((int index, IBenchmark bench) in
            ↪ benchmarks.WithIndex()) {
10              bench.Run();
11          }
12          PlotGroups();
13          if (CsharpRAPLCLI.Options.ZipResults) {
14              ZipResults();
15          }
16      }
17      ...
18  }
```

Listing 45: Running all the benchmarks with the `RunAll` method.

In Listing 45 we can see the method that runs all of the registered benchmarks. At the start of this method, we check if the platform is Unix. If it is not Unix then RAPL is not currently supported, meaning we cannot measure the benchmarks and a `NotSupportedException` is thrown. Afterwards, the benchmarks are put into a list and the `Warmup` method is called.After this, the `Run` method (Seen in Listing 54) is called on every benchmark, which runs the benchmarks and collects measurements. After the benchmarks have been run, the `PlotGroups` method (Explained in Section 5.3) is called and at last the results are put into a *zip* file if the `ZipResults` CLI option is `true`.

```
1   public class BenchmarkSuite {
2       ...
3       private static void Warmup() {
4           int warmup = 0;
5           for (int i = 0; i < 10; i++) {
6               while (warmup < int.MaxValue) {
7                   warmup++;
8                   ... // Write percentage completion to console
9               }
```

```
10              warmup = 0;
11          }
12      }
13      ...
14  }
```

Listing 46: Warming up to decrease number of outliers, with the `Warmup` method.

In Listing 46 we can see the warmup function. This serves to get the hardware ready to run benchmarks without having to cold-start, which helps to remove outliers (As seen in Section A.1.6). We do this by creating a variable named `warmup` which we increment to the max value of `int` 10 times. To ensure that the compiler does not optimize the code away when running, we write the percentage completion to console.

```
1   public class BenchmarkSuite {
2       ...
3       private static void ZipResults() {
4           using var zipToOpen = new FileStream("results.zip",
            ↪  FileMode.Create);
5           using var archive = new ZipArchive(zipToOpen,
            ↪  ZipArchiveMode.Update);
6           foreach (string file in
            ↪  Helpers.GetAllCSVFilesFromOutputPath()) {
7               archive.CreateEntryFromFile(file, file);
8           }
9           foreach (string file in Helpers.GetAllPlotFiles()) {
10              archive.CreateEntryFromFile(file, file);
11          }
12      }
13      ...
14  }
```

Listing 47: Zipping the results so they fill less space, using the `ZipResults` method.

In Listing 47 we can see how we create a zip file with our results from running the benchmarks. We do this by creating a `FileStream` in the `Create FileMode`. We then create a `ZipArchive` [135] using this file set in

`ZipArchiveMode.Update` to update the file, after which we write all of the `.csv` files and plots into the archive.

Collecting Benchmarks

Now that we have created a method to register and run the benchmarks, we create a method to collect the benchmarks automatically so the user does not need to manually send the benchmarks to the relevant methods. To do this, we create a class called `BenchmarkCollector` that inherits from `BenchmarkSuite`.

```
1 public class BenchmarkCollector : BenchmarkSuite {
2     private readonly Dictionary<Type,
      ↪  RegisterBenchmarkVariation>
      ↪  _registeredBenchmarkVariations = new();
3     private const BindingFlags RegisterBenchmarkFlags =
      ↪  BindingFlags.Instance | BindingFlags.Public |
      ↪  BindingFlags.InvokeMethod;
4     private static readonly MethodInfo
      ↪  RegisterBenchmarkGenericMethod =
      ↪  typeof(BenchmarkSuite).GetMethods(RegisterBenchmarkFlags).First(info
      ↪  => info.Name == nameof(RegisterBenchmark) &&
      ↪  info.GetParameters().Length == 3);
5     ...
6 }
7 internal sealed record RegisterBenchmarkVariation(MethodInfo
  ↪  GenericAddBenchmark, Type FuncType);
```

Listing 48: BenchmarkCollector class Fields.

The `BenchmarkCollector` class has 3 fields. The field `_registeredBenchmarkVariations` is a `Dictionary` that maps return types to a `RegisterBenchmarkVariation`. This makes it possible for benchmarks to have different return types instead of being constrained to just one. For example, you may have a benchmark that does operations on a string, in which case you may want to return a string, but another benchmark does operations on an integer, in which case you may return an integer.

The field `RegisterBenchmarkFlags` holds flags that makes it easier to find the method `RegisterBenchmark` for the `RegisterBenchmarkGenericMethod`.

The field `RegisterBenchmarkGenericMethod` contains the `MethodInfo` [136] for the `RegisterBenchmark` method, making it possible to invoke this method at a later stage when we have collected the benchmarks.

The `RegisterBenchmarkVariation record` serves to keep track of the `RegisterBenchmark` method with the correct return type, so it can be invoked correctly.

```
public class BenchmarkCollector : BenchmarkSuite {
    ...
    private void CollectBenchmarks(Assembly assembly) {
        foreach (MethodInfo benchmarkMethod in
        ↪ assembly.GetTypes().SelectMany(type =>
        ↪ type.GetMethods().Where(info =>
        ↪ info.GetCustomAttribute<BenchmarkAttribute>() !=
        ↪ null))) {
            var benchmarkAttribute =
            ↪ benchmarkMethod.GetCustomAttribute<BenchmarkAttribute>()!;
            if (benchmarkAttribute.Skip) {
                continue;
            }
            CheckMethodValidity(benchmarkMethod);
            if (!_registeredBenchmarkVariations.
            ↪ ContainsKey(benchmarkMethod.ReturnType)) {
                RegisterAddBenchmarkVariation(benchmarkMethod);
            }
            (MethodInfo genericRegisterBenchmark, Type funcType)
            ↪ =
            ↪ _registeredBenchmarkVariations[benchmarkMethod.ReturnType];
            Delegate benchmarkDelegate =
            ↪ benchmarkMethod.IsStatic ?
            ↪ benchmarkMethod.CreateDelegate(funcType) :
            ↪ benchmarkMethod.CreateDelegate(funcType,
            ↪ Activator.CreateInstance(benchmarkMethod.DeclaringType!));
            genericRegisterBenchmark.Invoke(this, new object[]
            ↪ {benchmarkAttribute.Group!, benchmarkDelegate,
            ↪ benchmarkAttribute.Order});
        }
    }
```

```
18      ...
19  }
```

Listing 49: Collecting the benchmarks using reflection, with the `CollectBenchmarks` method.

In Listing 49, the `CollectBenchmarks` method can be seen. This is the main method for collecting benchmarks automatically. To collect benchmarks automatically, we utilize `reflection` [137] and `attributes` [138]. We require the benchmarks to have an `attribute` with the name `Benchmark`.

We go through all the methods in all classes in the assembly to find the methods with the attribute `Benchmark`. These methods are then gone through in the `foreach` loop, where we first find the attribute itself.

We check if the skip field has been set, in which case we skip the benchmark. After this, we check if the method is valid by calling the `CheckMethodValidity` method with the benchmark. We then check if there is a `RegisterBenchmark` method with the correct return type in our `Dictionary`, if not, we call the `RegisterAddBenchmarkVariation` method to add a `RegisterBenchmark` method with the correct return type.

After adding the `RegisterBenchmark` with the correct return type to our `Dictionary`, we get the correct `RegisterBenchmark` method to invoke.

Afterwards, we check if the benchmark is static or not, to create the correct `delegate`.

Lastly, we invoke the `genericRegisterBenchmark` method with the correct return type with the relevant parameters to register the benchmark in the `HashSet` in the `BenchmarkSuite` class.

```
1  [AttributeUsage(AttributeTargets.Method)]
2  public class BenchmarkAttribute : Attribute {
3      public string? Group { get; }
4      public string Description { get; }
5      public int Order { get; }
6      public bool Skip { get; }
7      public BenchmarkAttribute(string? group, string description,
       ↪  int order = 0, bool skip = false) {
8          Group = group;
9          Description = description;
10         Order = order;
```

```
11          Skip = skip;
12      }
13  }
```

Listing 50: The `BenchmarkAttribute` class.

In Listing 50 we see the `BenchmarkAttribute` class. This class determines what fields can be set in the attribute. We can see that the attribute can only be set on methods, because the class has the attribute `AttributeUsage` set to `AttributeTargets.Method`. A benchmark can have a group for comparing with other group members, a description for understanding, an order for when benchmarks are to be run and a boolean to determine if it should be skipped.

```
1  public class BenchmarkCollector : BenchmarkSuite {
2      ...
3      private static void CheckMethodValidity(MethodInfo
       ↪  benchmark) {
4          if (!benchmark.IsPublic)
5              throw new NotSupportedException("The benchmark
               ↪  attribute is only supported and supposed to be
               ↪  used on public methods.");
6          if (benchmark.ReturnType == typeof(void))
7              throw new NotSupportedException("The benchmark
               ↪  attribute is only supported and supposed to be
               ↪  used on methods with a non void return type.");
8      }
9      ...
10 }
```

Listing 51: Checking Validity of Methods.

In Listing 51 we can see the `CheckMethodValidity` method. This method checks if the benchmark has the correct visibility and ensures that the benchmark has a return type.

```
1  public class BenchmarkCollector : BenchmarkSuite {
2      ...
```

```
3       private void RegisterAddBenchmarkVariation(MethodInfo
        ↪ benchmark) {
4           Type funcType =
            ↪ typeof(Func<>).MakeGenericType(benchmark.ReturnType);
5           MethodInfo genericAddBenchmark =
            ↪ RegisterBenchmarkGenericMethod.MakeGenericMethod
            ↪ (benchmark.ReturnType);
6           _registeredBenchmarkVariations.Add(benchmark.ReturnType,
            ↪ new RegisterBenchmarkVariation(genericAddBenchmark,
            ↪ funcType));
7       }
8       ...
9   }
```

Listing 52: Registering `RegisterBenchmark` variations with different return types.

In Listing 52 we can see the `RegisterAddBenchmarkVariation` method. This method serves to create a version of `RegisterBenchmark` with the correct return type.

First we find the return type of the benchmark that has been sent to this method, and make it generic. After this, we create a variation of `RegisterBenchmark` with the same return type, and then add that to the `_registeredBenchmarkVariations Dictionary`, so we can access it later.

Individual Benchmarks

Now that we have set up a way to collect benchmarks automatically and call the run method individually on them, we look at the `Benchmark` class which contains the functionality of actually running the benchmarks.

```
1   public class Benchmark<T> : IBenchmark {
2       public int Iterations { get; private set; }
3       public string Name { get; }
4       public string? Group { get; }
5       public int Order { get; }
6       public bool HasRun { get; private set; }
7       public double ElapsedTime { get; private set; }
8       private const int MaxExecutionTime = 2700; //In seconds
```

```
9       private const int TargetLoopIterationTime = 250; //In
        ↪  milliseconds
10      private readonly TextWriter _benchmarkOutputStream = new
        ↪  StreamWriter(Stream.Null);
11      private readonly TextWriter _stdout;
12      private readonly Func<T> _benchmark;
13      private readonly List<BenchmarkResult> _rawResults = new();
14      private readonly List<BenchmarkResult> _normalizedResults =
        ↪  new();
15      private readonly FieldInfo? _loopIterationsFieldInfo;
16      private RAPL? _rapl;
17      private string? _normalizedReturnValue;
18      ...
19  }
```

Listing 53: The `Benchmark` class fields.

In Listing 53, we can see the fields in the `Benchmark` class. These fields are used to keep track of various information needed for saving and running the benchmarks. The `MaxExecutionTime` and `TargetLoopIterationTime` fields are used to ensure that the benchmarks are not run for too long and that they run for long enough, respectively. Furthermore, the `_benchmark` field keeps track of which benchmark we are currently running and the `_rapl` field keeps track of where the RAPL measurements are stored. Lastly the `_normalizedReturnValue` field keeps track of a normalized return value for the benchmark we are running, so we can validate that it gives the same return value as comparable benchmarks.

```
1   public class Benchmark<T> : IBenchmark {
2       ...
3       public void Run() {
4           Console.SetOut(_benchmarkOutputStream);
5           _rapl = new RAPL();
6           ElapsedTime = 0;
7           ...
8           if (CsharpRAPLCLI.Options.UseIterationCalculation) {
9               Iterations = IterationCalculationAll();
10          }
```

```
11          SetLoopIterations(10);
12          _normalizedReturnValue = _benchmark()?.ToString() ??
            ↪ string.Empty;
13          for (var i = 0; i <= Iterations; i++) {
14              ...
15              Start();
16              T benchmarkOutput = _benchmark();
17              End(benchmarkOutput);
18              ...
19          }
20          ...
21      }
22      ...
23  }
```

Listing 54: Running individual benchmarks using the `Run` method 1/3.

In Listing 54 we can see the first part of the `Run` method, in the `Benchmark` class. This starts by setting where the `Console` writes to null, this eliminates a part of the overhead for writing to the `Console`.

After this we create a new `RAPL` object and sets `ElapsedTime` to 0.

After the initial variables have been set up, we calculate the amount of `Iterations` needed by calling the `IterationCalculationAll` method. This has a minimum of 10 iterations to get enough data for us to calculate an iteration count properly.

After calculating the number of iterations, we run the benchmark with a normalized amount of `LoopIterations` so we can get a return value that we can use to validate the results at a later stage.

After we have done this, we run the benchmark as many times as we calculated is necessary to get a significant result. We do this by first calling the `Start` method to set up the `RAPL` object, then calling the benchmark and then calling the `End` method to end the measurements.

```
1   public class Benchmark<T> : IBenchmark {
2       ...
3       public void Run() {
4           ...
5           for (var i = 0; i <= Iterations; i++) {
```

```
6               ...
7               if (_loopIterationsFieldInfo != null &&
                ↪  CsharpRAPLCLI.Options.UseLoopIterationScaling &&
                ↪  _rawResults[^1].ElapsedTime <
                ↪  TargetLoopIterationTime) {
8                   int currentValue = GetLoopIterations();
9                   switch (currentValue) {
10                      ...
11                      default:
12                          SetLoopIterations(currentValue +
                            ↪  currentValue);
13                          _rawResults.Clear();
14                          _normalizedResults.Clear();
15                          i = 0;
16                          continue;
17                  }
18              }
19              ...
20          }
21          ...
22      }
23  }
```

Listing 55: Running individual benchmarks using the `Run` method 2/3.

In Listing 55 the second part of the `Run` method can be seen. Here we double the amount of `LoopIterations` if the elapsed time is less than the target time, which by default is 0,25 seconds. This can be ignored if the user explicitly sets an option for it to be ignored. We use the `SetLoopIterations` method to double the amount of `LoopIterations` in the benchmark currently being run. After this we clear the results we have gathered from running the benchmarks with the old amount of `LoopIterations`, as these would have been run with half the amount of `LoopIterations` and therefore would be misleading.

```
1   public class Benchmark<T> : IBenchmark {
2       ...
3       public void Run() {
```

```
4           ...
5           for (var i = 0; i <= Iterations; i++) {
6               ...
7               if (ElapsedTime < MaxExecutionTime) {
8                   if (CsharpRAPLCLI.Options.
                    ↪ UseIterationCalculation) {
9                       Iterations = IterationCalculationAll();
10                  }
11                  continue;
12              }
13              break;
14          }
15          SaveResults();
16          HasRun = true;
17          ...
18      }
19  }
```

Listing 56: Running individual benchmarks using the `Run` method 3/3.

In Listing 56 the last part of the `Run` method can be seen. Here we check if the benchmarks have run for too long. If not, we recalculate the number of `Iterations` by calling the `IterationCalculationAll` method. We recalculate the number of `Iterations` every cycle as we can get a more precise number of `Iterations` with the data we gather. After recalculating the number of `Iterations`, we `continue` with the `for` loop. If `ElapsedTime` has gone above `MaxExecutionTime` we break the `for` loop, save the results and set `HasRun` to `true`.

```
1   public sealed class RAPL {
2       private readonly DRAMApi _dramApi;
3       private readonly TimerApi _timerApi;
4       private readonly PackageApi _packageApi;
5       private readonly TempApi _tempApi;
6
7       public RAPL() {
8           _dramApi = new DRAMApi();
9           _timerApi = new TimerApi();
```

```
10          _packageApi = new PackageApi();
11          _tempApi = new TempApi();
12      }
13      ...
14  }
```

Listing 57: The RAPL class.

In Listing 57 we can see the fields and constructor of the RAPL class. This class serves to provide access to the measurements provided by RAPL via the Application Programming Interface (API)'s created in [9]. There is a field for each of the RAPL measurements, which are initialized in the constructor.

```
1   public sealed class DRAMApi : DeviceApi {
2       public DRAMApi() : base(CollectionApproach.Difference) { }
3       protected override string OpenRaplFile() {
4           return GetDRAMFile(GetSocketDirectoryName());
5       }
6       private static string GetDRAMFile(string directoryName) {
7           var raplDeviceId = 0;
8           while (Directory.Exists(
            ↪  $"{directoryName}/intel-rapl:0:{raplDeviceId}")) {
9               var dirName =
                ↪  $"{directoryName}/intel-rapl:0:{raplDeviceId}";
10              string content =
                ↪  File.ReadAllText($"{dirName}/name").Trim();
11              if (content.Equals("dram")) {
12                  return $"{dirName}/energy_uj";
13              }
14
15              raplDeviceId += 1;
16          }
17
18          throw new Exception("PyRAPLCantInitDeviceAPI");
19      }
20  }
```

Listing 58: The DRAMApi class.

In Listing 58 we can see the DRAMApi. The class implements the DeviceApi abstract class, and has two methods.

Specifically, the OpenRaplFile is the method that is called when accessing the file that contains the measurement.

For DRAM, the OpenRaplFile calls the GetDRAMFile with the directory name that contains the RAPL files.

The GetDRAMFile goes through all the directories within the socket directory on the form *intel-rapl:0:ID* until it finds a directory that has a file with the name *"name"* which has *"dram"* as the content. When it finds the correct directory, it returns the path to the *"energy_uj"* file within this directory.

```
1  public sealed class TimerApi : DeviceApi {
2      private readonly Stopwatch _sw = new();
3
4      public TimerApi() : base(CollectionApproach.Difference) {
5          _sw.Start();
6      }
7
8      protected override string OpenRaplFile() {
9          return "";
10     }
11
12     protected override double Collect() {
13         return _sw.Elapsed.TotalMilliseconds;
14     }
15 }
```

Listing 59: The TimerApi class.

In Listing 59 we can see the TimerApi. This is less complex than the DRAMApi class, as there is no need to utilize an external file. We utilize the Stopwatch class to measure time. We start the Stopwatch in the constructor and we get the current elapsed time in the Collect method. The OpenRaplFile returns an empty string because there is no file for the timer. The method is still implemented because the TimerApi class inherits the DeviceApi abstract class.

```
public sealed class PackageApi : DeviceApi {
    public PackageApi() : base(CollectionApproach.Difference) {
    ↪ }

    protected override string OpenRaplFile() {
        return $"{GetSocketDirectoryName()}/energy_uj";
    }
}
```

Listing 60: The `PackageApi` class.

In Listing 60 we can see the `PackageApi`. Here we can see that the energy for `Package` is stored in the main socket directory, and therefore we can just return that path in the `OpenRaplFile` method.

```
public sealed class TempApi : DeviceApi {
    public TempApi() : base(CollectionApproach.Average) { }

    protected override string OpenRaplFile() {
        const string path = "/sys/class/thermal/";
        var thermalId = 0;
        while
        ↪ (Directory.Exists($"{path}/thermal_zone{thermalId}"))
        ↪ {
            var dirname = $"{path}/thermal_zone{thermalId}";
            string type =
            ↪ File.ReadAllText($"{dirname}/type").Trim();
            if (type.Contains("pkg_temp")) {
                return $"{dirname}/temp";
            }

            thermalId++;
        }

        throw new Exception("No thermal zone found for the
        ↪ package");
    }
}
```

Listing 61: The `TempApi` class.

In Listing 61 we can see the `TempApi`. This is similar to `DRAMApi` as we need to look through multiple files to find the one we want. The main difference is that we are not looking in the socket directory, but rather in the `thermal` directory, where the temperature measurement is stored.

```
public abstract class DeviceApi {
    public double Delta { get; private set; }
    private double _endValue;
    private double _startValue;
    private readonly CollectionApproach _approach;
    private readonly string _sysFile;
    protected DeviceApi(CollectionApproach collectionApproach) {
        _approach = collectionApproach;
        _sysFile = OpenRaplFile();
    }
    protected abstract string OpenRaplFile();
    ...
}
```

Listing 62: The `DeviceApi` class 1/3.

In Listing 62 we can see the first part of the `DeviceApi` class. This class has five fields, `Delta`, `_endValue`, `_startValue`, `_approach` and `_sysFile`.

The `Delta` field serves to provide a measurement depending on the `_approach` field, for all API's except for `TempApi` the difference between the fields `_endValue` and `_startValue` is used. For the `TempApi` the average between the two values are used instead. The `_sysFile` field has the path to the relevant file path that is gathered from the `OpenRaplFile` method that is implemented in all the API's.

```
public abstract class DeviceApi {
    ...
    protected static string GetSocketDirectoryName() {
        if (Directory.Exists(
        ↪  "/sys/class/powercap/intel-rapl/intel-rapl:0")) {
```

```
5               return
                ↪  "/sys/class/powercap/intel-rapl/intel-rapl:0";
6           }
7           throw new Exception("PyRAPLCantInitDeviceAPI");
8       }
9       protected virtual double Collect() {
10          double res = -1.0;
11          if (double.TryParse(File.ReadAllText(_sysFile), out
            ↪  double energyValue)) {
12              res = energyValue;
13          }
14          return res;
15      }
16      ...
17  }
```

Listing 63: The `DeviceApi` class 2/3.

In Listing 63 we can see the second part of the `DeviceApi` class.

Here we can see the two methods, `GetSocketDirectoryName` and `Collect`.

The `GetSocketDirectoryName` method checks if the directory that contains the RAPL measurements exists, in which case it returns the path to it, otherwise throwing an `Exception`.

The `Collect` method method collects the latest measurement from the `_sysFile`, meaning it collects the latest measurement by reading the file associated with the measurement.

```
1   public abstract class DeviceApi {
2       ...
3       public void Start() {
4           _startValue = Collect();
5       }
6       public void End() {
7           _endValue = Collect();
8           UpdateDelta();
9       }
10      public bool IsValid() {
```

```
11          return Math.Abs(_startValue - -1.0) > double.Epsilon &&
            ↪  Math.Abs(_endValue - -1.0) > double.Epsilon && Delta
            ↪  >= 0;
12      }
13      private void UpdateDelta() {
14          Delta = _approach switch {
15              CollectionApproach.Difference => _endValue -
                ↪  _startValue,
16              CollectionApproach.Average => (_endValue +
                ↪  _startValue) / 2,
17              _ => throw new Exception("Collection approach is not
                ↪  available")
18          };
19      }
20  }
```

Listing 64: The `DeviceApi` class 3/3.

In Listing 64 we can see the last part of the `DeviceApi` class. Here there are four methods, `Start`, `End`, `IsValid` and `UpdateDelta`. The `Start` and `End` method are meant to be called at the start of a measurement and at the end of a measurement. These set the `_startValue` and `_endValue` respectively by calling the `Collect` method. After the `_endValue` has been set, the `UpdateDelta` method is called, which updates the `Delta` field depending on the value the `_approach` field has. The `IsValid` method checks if the values gathered are valid by checking if the `_startValue` and `_endValue` are different from the default value of -1 and checking if `Delta` is higher than or equal to 0.

```
1   public class Benchmark<T> : IBenchmark {
2       ...
3       private int IterationCalculationAll(double confidence =
        ↪  0.95) {
4           int dramIteration = IterationCalculation(confidence,
            ↪  BenchmarkResultType.DRAMEnergy);
5           int timeIteration = IterationCalculation(confidence,
            ↪  BenchmarkResultType.ElapsedTime);
6           int packageIteration = IterationCalculation(confidence);
```

```
7           return Math.Max(dramIteration, Math.Max(timeIteration,
            ↪ packageIteration));
8       }
9       ...
10  }
```

Listing 65: Calculating the Iteration for all measurement types.

In Listing 65 the code for the `IterationCalculationAll` method can be seen. The purpose of this method is to call `IterationCalculation` for all the different measurement types to ensure that we get enough samples for a significant result, which by default is a confidence of 95% (p-value of 0,05). After calculating the `Iteration` amount for each of the measurement types, we return the highest amount.

```
1   public class Benchmark<T> : IBenchmark {
2       ...
3       private int IterationCalculation(double confidence = 0.95,
        ↪ BenchmarkResultType resultType =
        ↪ BenchmarkResultType.PackageEnergy) {
4           if (_rawResults.Count < 10) {
5               return 10;
6           }
7           double alpha = 1 - confidence;
8           var values = new double[_rawResults.Count];
9           for (var i = 0; i < _rawResults.Count; i++) {
10              values[i] = resultType switch {
11                  BenchmarkResultType.Temperature =>
                    ↪ _rawResults[i].Temperature,
12                  ...
13              };
14          }
15          double mean = values.Average();
16          double stdDeviation = values.StandardDeviation(mean);
17          NormalDistribution nd = new(mean, stdDeviation);
18          return (int) Math.Ceiling(Math.Pow(nd.ZScore(
            ↪ nd.GetRange(alpha / 2).Max) * stdDeviation / (0.005
            ↪ * mean), 2));
19      }
```

```
20      ...
21  }
```

Listing 66: Calculating the Iteration for a specific measurement types.

In Listing 66, we can see the code that implements the formula seen in Equation 4.2. We utilize the library Accord to help with the calculation, specifically we use it for the standard deviation and normal distribution [106].

We have 10 as the minimum number of measurements, as recommended by [95]. Once 10 measurements have been completed, we have a sample from which we can calculate the number of iterations needed for a significant result.

We start by calculating the α value, which is the p-value, this is calculated by subtracting the confidence wanted from 1. After this, we create an array of doubles that has the same size as the number of results we have gathered so far. We add the relevant measures gathered to the values array, after which we calculate the mean and standard deviation of these values. With the mean and standard deviation, we can create a normal distribution that can be used to calculate the z-score for the equation.

At the end, we put in the numbers as they are seen in Equation 4.2 and return this value. The z-score is here calculated by giving the `ZScore` method the highest value in the first $alpha/2$ part of the normal-distribution, as the `ZScore` method takes a value instead of a percentage.

```
1  public class Benchmark<T> : IBenchmark {
2      ...
3      private void Start() {
4          if (_rapl is null) {
5              throw new NotSupportedException("Rapl has not been
                   ↪  initialized");
6          }
7          _rapl.Start();
8      }
9      ...
10 }
```

Listing 67: The `Start` method in the `Benchmark` class.

In Listing 67 we can see the `Start` method in the `Benchmark` class, that is called during the `Run` method.

Here we first ensure that `_rapl` is initialized, after which we call the `Start` method on it.

```
public sealed class RAPL {
    ...
    public void Start() {
        _timerApi.Start();
        _tempApi.Start();
        _dramApi.Start();
        _packageApi.Start();
    }
    ...
}
```

Listing 68: The `Start` method in the `RAPL` class.

In Listing 68 we can see the `Start` method in the `RAPL` class, this serves to call the `Start` methods on all the API's.

```
public class Benchmark<T> : IBenchmark {
    ...
    private void End(T benchmarkOutput) {
        if (_rapl is null) {
            throw new NotSupportedException("Rapl has not been
            ↪  initialized");
        }
        _rapl.End();
        if (!_rapl.IsValid()) {
            return;
        }
        BenchmarkResult result = _rapl.GetResults() with {
            BenchmarkReturnValue = benchmarkOutput?.ToString()
            ↪  ?? string.Empty
        };

```

```
15          BenchmarkResult normalizedResult =
            ↪ _rapl.GetNormalizedResults(GetLoopIterations()) with
            ↪ {
16              BenchmarkReturnValue = _normalizedReturnValue ??
                ↪ string.Empty
17          };
18          _rawResults.Add(result);
19          _normalizedResults.Add(normalizedResult);
20          ElapsedTime += _rawResults[^1].ElapsedTime / 1_000;
21      }
22      ...
23  }
```

Listing 69: The `End` method in the `Benchmark` class.

In Listing 67 we can see the `End` method in the `Benchmark` class, that is called during the `Run` method.

Here we first ensure that `_rapl` is initialized, after this we call the `End` method on the `_rapl` object.

Once we have ended the measurements, we check if the results are valid by calling the `IsValid` method on the `_rapl` object.

After checking if the results are valid, we get the raw results and the normalized results. In the raw results we cast the `benchmarkOutput` in the result to a string and assign it to `BenchmarkReturnValue` in the `BenchmarkResult` object if `benchmarkOutput` is set. If `benchmarkOutput` is not set we set the `BenchmarkReturnValue` to an empty string.

In the normalized results we use the `_normalizedReturnValue` as the `BenchmarkReturnValue` as this is what is used at a later stage when verifying the results of a benchmark.

After getting the results, we add them to the relevant lists and increase the `ElapsedTime` by the time the benchmark has run for, divided by 1.000 to get it in seconds.

```
1   public sealed class RAPL {
2       ...
3       public void End() {
4           _packageApi.End();
5           _dramApi.End();
```

```
6           _tempApi.End();
7           _timerApi.End();
8       }
9       ...
10  }
```

Listing 70: The `End` method in the `RAPL` class.

In Listing 70 we can see the `End` method in the `RAPL` class, this serves to call the `End` methods on all the API's.

```
1   public sealed class RAPL {
2       ...
3       public bool IsValid() {
4           return _dramApi.IsValid() && _tempApi.IsValid() &&
            ↪  _timerApi.IsValid() && _packageApi.IsValid();
5       }
6       ...
7   }
```

Listing 71: The `IsValid` method in the `RAPL` class.

In Listing 71 we can see the `IsValid` method in the `RAPL` class.

This method serves to call the `IsValid` method on all the API's, making sure that all of the values gathered can be used.

```
1   public sealed class RAPL {
2       ...
3       public BenchmarkResult GetResults() {
4           BenchmarkResult result = new() {
5               DRAMEnergy = _dramApi.Delta,
6               Temperature = _tempApi.Delta,
7               ElapsedTime = _timerApi.Delta,
8               PackageEnergy = _packageApi.Delta
9           };
10          return result;
11      }
12      ...
13  }
```

Listing 72: The `GetResults` method in the `RAPL` class.

In Listing 72 we can see the `GetResults` method in the `RAPL` class.

This method gets the results that are saved in the `Delta` fields in each of the API's and saves them in a `BenchmarkResult` object which is returned.

```
public record BenchmarkResult {
    [Index(0)]
    public double ElapsedTime { get; init; }
    [Index(1)]
    public double PackageEnergy { get; init; }
    [Index(2)]
    public double DRAMEnergy { get; init; }
    [Index(3)]
    public double Temperature { get; init; }
    [Index(4)]
    public string BenchmarkReturnValue { get; init; } =
    ↪ string.Empty;
}
```

Listing 73: The `BenchmarkResult` record class.

In Listing 73, the `BenchmarkResult` record class can be seen. This contains the relevant information with regards to the results of a benchmark, including `ElapsedTime`, `PackageEnergy`, `DRAMEnergy`, `Temperature` and lastly `BenchmarkReturnValue`.

The `ElapsedTime` is the amount of time a benchmark has run for. The `PackageEnergy` is the amount of package energy a benchmark has used when run. The `DRAMEnergy` is the amount of DRAM energy a benchmark has used when run. The `Temperature` is the temperate the CPU was at when the benchmark was run. The `BenchmarkReturnValue` is the return value of the benchmark.

The `Index` attribute serves to tell the `CsvWriter` in `SaveResults` what column the result should go in.

```
public sealed class RAPL {
    ...
```

```
3       public BenchmarkResult GetNormalizedResults(int
        ↪  loopIterations, int normalizedIterations = 1000000) {
4           BenchmarkResult result = new() {
5               DRAMEnergy = _dramApi.Delta /
                ↪  ((double)loopIterations / normalizedIterations),
6               Temperature = _tempApi.Delta / 1000,
7               ElapsedTime = _timerApi.Delta /
                ↪  ((double)loopIterations / normalizedIterations),
8               PackageEnergy = _packageApi.Delta /
                ↪  ((double)loopIterations / normalizedIterations)
9           };
10          return result;
11      }
12      ...
13  }
```

Listing 74: The `GetNormalizedResults` method in the `RAPL` class.

In Listing 74 we can see the `GetNormalizedResults` method in the `RAPL` class. This method gets the results that are saved in the `Delta` fields in each of the API's and divides them with `loopIterations` divided by `normalizedIterations` and saves them in a `BenchmarkResult` object which is returned. By default this normalizes all of the results to 1.000.000 `loopIterations`, making them comparable. 1.000.000 is chosen as that ensures the resulting numbers are large enough to be practical to work with.

Note that `Temperature` is not normalized as the result is an average instead of a difference and therefore it is not necessary.

```
1   public class Benchmark<T> : IBenchmark {
2       ...
3       private int GetLoopIterations() {
4           if (_loopIterationsFieldInfo != null) {
5               return (int)(_loopIterationsFieldInfo.GetValue(null)
                ↪  ?? throw new InvalidOperationException());
6           }
7           return 0;
8       }
9       ...
10  }
```

Listing 75: The `GetLoopIterations` method in the `Benchmark` class.

In Listing 75 the `GetLoopIterations` method in the `Benchmark` class can be seen. This method returns the value of the `_loopIterationsFieldInfo` field, which is the value of `LoopIterations` in the benchmark that is currently being run.

```
public class Benchmark<T> : IBenchmark {
    ...
    private void SetLoopIterations(int value) {
        if (_loopIterationsFieldInfo != null) {
            _loopIterationsFieldInfo.SetValue(null, value);
        }
    }
    ...
}
```

Listing 76: The `SetLoopIterations` method in the `Benchmark` class.

In Listing 76 the `SetLoopIterations` method in the `Benchmark` class can be seen. This method sets the value of the `_loopIterationsFieldInfo` field, which is the value of `LoopIterations` in the benchamrk that is currently being run.

```
public class Benchmark<T> : IBenchmark {
    ...
    private void SaveResults() {
        DateTime dateTime = DateTime.Now;
        var time = $"{dateTime.ToString("s").Replace(":",
        ↪  "-")}-{dateTime.Millisecond}";
        string outputPath = Group != null
            ?
            ↪  $"{CsharpRAPLCLI.Options.OutputPath}/{Group}/{Name}"
            : $"{CsharpRAPLCLI.Options.OutputPath}/{Name}";
        Directory.CreateDirectory(outputPath);
        using var writer = new
        ↪  StreamWriter($"{outputPath}/{Name}-{time}.csv");
```

```
11          using var csv = new CsvWriter(writer, new
            ↪ CsvConfiguration(CultureInfo.InvariantCulture)
12              { Delimiter = ";" });
13          csv.WriteRecords(GetResults());
14      }
15      ...
16  }
```

Listing 77: The `SaveResults` method in the `Benchmark` class.

In Listing 77 the `SaveResults` method in the `Benchmark` class can be seen.

This method serves to create a csv file with the results gathered from the benchmark. Here we start by getting the current timestamp so we can use this in our filename. We then create an output path with the group and name of the benchmark, so we have some structure in our results. We then create the directories needed for this output path, important to note here is that if the directories already exists, the `CreateDirectory` method does not do anything. Next, a `StreamWriter` is created with the `outputPath` and the desired name of the file, which is the name of the benchmark concatenated with the timestamp and *.csv* at the end to create a csv file. After this, we create a `CsvWriter` where we use the `StreamWriter` as the first parameter. The only configuration we change here is that we change the delimiter to ";" instead of "," as using a comma can give issues with some numbers. At last we use the `WriteRecords` method on the `CsvWriter` utilizing the `GetResults` method inside the `Benchmark` class to get the normalized results.

```
1   public class Benchmark<T> : IBenchmark {
2       ...
3       public List<BenchmarkResult> GetResults(bool ignoreFirst =
        ↪ true) {
4           return ignoreFirst ? new
            ↪ List<BenchmarkResult>(_normalizedResults.Skip(1)) :
            ↪ new List<BenchmarkResult>(_normalizedResults);
5       }
6       ...
7   }
```

Listing 78: The `GetResults` method in the `Benchmark` class.

In Listing 78 we can see the `GetResults` method in the `Benchmark` class. This method returns the `_normalizedResults` field, by default it skips the first item as preliminary results (seen in Section A.1) have shown that the first item can be an outlier.

A.2.3 Analysis Implementation

Now that we have made it possible to create and run benchmarks, we create a way to analyse and view results of these benchmarks.

BoxPlot

The first thing we do, is create a way to plot the results, so it is possible to visually see the results. To do this, we use the ScottPlot library [131], this library does not have box plots built in, so we create an implementation for that ourselves.

```
1  public class BoxPlot : IPlottable {
2      public double Position { get; }
3      public double[] PlotData { get; }
4      public double MaxValue { get; }
5      public double MinValue { get; }
6      public double UpperPValueQuantile { get; }
7      public double LowerPValueQuantile { get; }
8      public double Average { get; set; }
9      public double Median { get; set; }
10     public PlotOptions PlotOptions { get; }
11     public bool IsVisible { get; set; } = true;
12     public int XAxisIndex { get; set; }
13     public int YAxisIndex { get; set; }
14     private readonly double _errorBelow;
15     private readonly double _errorAbove;
16     ...
17 }
```

Listing 79: `BoxPlot` class fields.

In Listing 79 we can see the fields we have in our `BoxPlot` class. Most of the fields are self-explanatory, important to note is that the `Position`

field is used to keep track of which dataset we are currently plotting, as in, if it is the first dataset, second dataset etc. `PlotData` is the data we are plotting, `UpperPValueQuantile` and `LowerPValueQuantile` are used to figure out where the boxes in the boxplot should be plotted, `PlotOptions` are the options we have available when we want to plot some data, and `_errorBelow` and `_errorAbove` are used to plot the furthest outliers.

```
1  public class BoxPlot : IPlottable {
2      ...
3      public BoxPlot(double position, double[] plotData, double
       ↪  errorBelow, double errorAbove, PlotOptions plotOptions,
       ↪  double pValue = 0.05) {
4          PlotData = plotData.Length != 0 ? plotData : throw new
           ↪  ArgumentException("plotData must be an array that
           ↪  contains elements");
5          Position = position;
6          UpperPValueQuantile = plotData.Quantile(1-(pValue/2));
7          LowerPValueQuantile = plotData.Quantile(pValue/2);
8          MaxValue = plotData.Max();
9          MinValue = plotData.Min();
10         Average = plotData.Average();
11         Median = plotData.Median();
12         _errorBelow = errorBelow;
13         _errorAbove = errorAbove;
14         PlotOptions = plotOptions;
15     }
16     ...
17 }
```

Listing 80: The `BoxPlot` constructor.

In Listing 80 we can see the constructor for a `BoxPlot`. Here we can see how the values, which are not sent to the constructor, are calculated. Importantly, the `UpperPValueQuantile` and `LowerPValueQuantile` are calculated with the `Quantile` method to find the correct values to plot the box, with the default p-value being 0,05.

```
1  public class BoxPlot : IPlottable {
2      ...
```

```
    public AxisLimits GetAxisLimits() {
        double minSize = Math.Min(_errorBelow,
        ↪  LowerPValueQuantile);
        double maxSize = Math.Max(_errorAbove,
        ↪  UpperPValueQuantile);
        if (PlotOptions.StartFromZero) {
            minSize = 0.0;
        }
        double startPosition = Position - PlotOptions.BarWidth /
        ↪  2.0;
        double endPosition = Position + PlotOptions.BarWidth /
        ↪  2.0;
        return new AxisLimits(startPosition, endPosition,
        ↪  minSize, maxSize);
    }
    ...
}
```

Listing 81: The `GetAxisLimits` method in the `BoxPlot` class.

In Listing 81 we calculate the axis limits of the datasets, this means the plot does not extend far above, below or to the side of the datasets. The y-value is `minSize` and `maxSize` while the x-values is `startPosition` and `endPosition`. This ensures there is some space between the top and bottom of the plot to the graphs as well as some space between the datasets on the plot.

```
public class BoxPlot : IPlottable {
    ...
    public void ValidateData(bool deep = false) {
        Validate.AssertHasElements("PlotData", PlotData);
        if (deep) {
            Validate.AssertAllReal("PlotData", PlotData);
        }
    }
    ...
}
```

Listing 82: The `ValidateData` method in the `BoxPlot` class.

In Listing 82 we can see how the data is validated in the plot. Here we check if the `PlotData` array has any elements and if `deep` is set to true, if all elements have real numbers.

```
public class BoxPlot : IPlottable {
    ...
    public void Render(PlotDimensions dims, Bitmap bmp, bool
    ↪ lowQuality = false) {
        using Graphics gfx = Graphics.FromImage(bmp);
        gfx.SmoothingMode = lowQuality ? SmoothingMode.HighSpeed
        ↪ : SmoothingMode.AntiAlias;
        gfx.TextRenderingHint = lowQuality ?
        ↪ TextRenderingHint.SingleBitPerPixelGridFit :
        ↪ TextRenderingHint.AntiAliasGridFit;
        RenderBarVertical(dims, gfx, Position);
    }
    ...
}
```

Listing 83: The `Render` method in the `BoxPlot` class.

In Listing 83 we can see how the data is rendered, here we instantiate a `Graphics` object from an empty bitmap image. After this, depending on how if we want a low quality or high quality image, we set the `SmoothingMode` and `TextRenderingHint` to specific values and then call the `RenderBarVertical` method.

```
public class BoxPlot : IPlottable {
    ...
    private void RenderBarVertical(PlotDimensions dims, Graphics
    ↪ gfx) {
        float edge = dims.GetPixelX(Position -
        ↪ PlotOptions.BarWidth / 2);
        double valueSpan = UpperPValueQuantile -
        ↪ LowerPValueQuantile;
```

```
6           var rect = new RectangleF(edge,
            ↪ dims.GetPixelY(UpperPValueQuantile),
            ↪ (float)(PlotOptions.BarWidth * dims.PxPerUnitX),
            ↪ (float)(valueSpan * dims.PxPerUnitY));
7           RenderBarFromRect(rect, gfx);
8           if (!(PlotOptions.ErrorLineWidth > 0) || !(_errorAbove >
            ↪ double.Epsilon) || !(_errorBelow > double.Epsilon))
            ↪ {
9               return;
10          }
11          RenderExtra(dims, gfx);
12      }
13      ...
14  }
```

Listing 84: The `RenderBarVertical` method in the `BoxPlot` class.

In Listing 84 we define the rectangle that is the box in our boxplot, this means the values from quantile 0,025 to 0,975. The `edge` variable is the leftmost x-value of the box, the `valueSpan` variable is the height of the box and the `rect` variable is the actual rectangle that is the box. After creating the box, we call the `RenderBarFromRect` method, after which we check if we should render the minimum and maximum lines on the plot, in which case we call `RenderErrorBar`.

```
1   public class BoxPlot : IPlottable {
2       ...
3       private void RenderBarFromRect(RectangleF rect, Graphics
        ↪ gfx) {
4           using var outlinePen = new Pen(PlotOptions.BorderColor,
            ↪ PlotOptions.BorderLineWidth);
5           using Brush fillBrush = GDI.Brush(PlotOptions.FillColor,
            ↪ PlotOptions.HatchColor, PlotOptions.HatchStyle);
6           gfx.FillRectangle(fillBrush, rect.X, rect.Y, rect.Width,
            ↪ rect.Height);
7           if (PlotOptions.BorderLineWidth > 0) {
8               gfx.DrawRectangle(outlinePen, rect.X, rect.Y,
                ↪ rect.Width, rect.Height);
9           }
```

```
    }
    ...
}
```

Listing 85: The `RenderBarFromRect` method in the `BoxPlot` class.

In Listing 85 we can see how the box is rendered. Here we create a `Pen` and a `Brush` with the correct parameters, fill out the rectangle and draw the rectangle around the box if the border width is above 0.

```
public class BoxPlot : IPlottable {
    ...
    private void RenderExtra(PlotDimensions dims, Graphics gfx)
    ↪ {
        float errorCapStartX = dims.GetPixelX(Position -
        ↪ PlotOptions.ErrorCapSize * PlotOptions.BarWidth /
        ↪ 2);
        float errorCapEndX = dims.GetPixelX(Position +
        ↪ PlotOptions.ErrorCapSize * PlotOptions.BarWidth /
        ↪ 2);
        float errorCapAboveY = dims.GetPixelY(_errorAbove);
        float errorCapBelowY = dims.GetPixelY(_errorBelow);
        float startX = dims.GetPixelX(Position -
        ↪ PlotOptions.BarWidth / 2);
        float endX = dims.GetPixelX(Position +
        ↪ PlotOptions.BarWidth / 2);
        float averageStartY = dims.GetPixelY(Average);
        float averageEndY = dims.GetPixelY(Average);
        float medianStartY = dims.GetPixelY(Median);
        float medianEndY = dims.GetPixelY(Median);
        ...
    }
    ...
}
```

Listing 86: The `RenderExtra` method in the `BoxPlot` class 1/2.

In Listing 86 we can see the first part of how we render the extra values in the box plot. Here we first get the center of the boxplot, get where

the error lines (minimum and maximum values) should start and end and where they should be placed on the y-axis. Furthermore, we get the same values for the average and median lines.

```
1  public class BoxPlot : IPlottable {
2      ...
3      private void RenderErrorBar(PlotDimensions dims, Graphics
       ↪  gfx) {
4          ...
5          using var pen = new Pen(PlotOptions.ErrorColor,
           ↪  PlotOptions.ErrorLineWidth);
6          gfx.DrawLine(pen, centerBottom,
           ↪  dims.GetPixelY(UpperPValueQuantile), centerBottom,
           ↪  errorCapAboveY);
7          gfx.DrawLine(pen, centerBottom,
           ↪  dims.GetPixelY(UpperPValueQuantile), centerBottom,
           ↪  errorCapBelowY);
8          gfx.DrawLine(pen, errorCapStartX, errorCapAboveY,
           ↪  errorCapEndX, errorCapAboveY);
9          gfx.DrawLine(pen, errorCapStartX, errorCapBelowY,
           ↪  errorCapEndX, errorCapBelowY);
10         pen.Width = 2;
11         gfx.DrawLine(pen, startX - 5, medianStartY, endX + 5,
           ↪  medianEndY);
12         pen.Width = 1;
13         pen.DashStyle = DashStyle.Dash;
14         gfx.DrawLine(pen, startX, averageStartY, endX,
           ↪  averageEndY);
15     }
16     ...
17 }
```

Listing 87: The `RenderErrorBar` method in the `BoxPlot` class 2/2.

In Listing 87 we can see the second part of how we render the extra values in the box plot. Here we create a `Pen` to draw the the error, average and median lines. The error lines are drawn with default settings, meaning a solid 1 pixel wide line. The median line has a solid 2 pixel wide line, and the average line is a dashed 1 pixel wide line.

Plotting Groups of Benchmark Measurements

Now that we have created a way to render boxplots, we can begin plotting the groups of benchmarks.

```
1  public class BenchmarkSuite {
2      ...
3      public void PlotGroups() {
4          foreach ((string? group, List<IBenchmark>? benchmarks)
           ↪  in GetBenchmarksByGroup()) {
5              BenchmarkPlot.PlotAllResults(benchmarks.ToArray(),
               ↪  new PlotOptions { Name = group });
6          }
7      }
8  }
```

Listing 88: The `PlotGroups` method in the `BenchmarkSuite` class.

In Listing 88 we can see the `PlotGroups` method in the `BenchmarkSuite` class. This method is called after all of the benchmarks have been run, as seen in Listing 45. This method serves to call the static method `PlotAllResults`, which plots all the results of a benchmark on seperate boxplots to make it possible to visualize the results.

```
1  public class PlotOptions {
2      ...
3      public string Name { get; set; } = "";
4      public int Width { get; set; } = 600;
5      public int Height { get; set; } = 450;
6      ...
7  }
```

Listing 89: The `PlotOptions` class fields.

In Listing 89 we can see the `PlotOptions` class. This class serves to provide options for the plots that are created, for example the name that is displayed at the top of the plot, the width and the height of the plot etc.

```
public static class BenchmarkPlot {
    ...
    public static void PlotAllResults(IBenchmark[] dataSet,
    ↪ PlotOptions? plotOptions = null) {
        PlotResults(BenchmarkResultType.ElapsedTime, dataSet,
        ↪ plotOptions);
        PlotResults(BenchmarkResultType.PackageEnergy, dataSet,
        ↪ plotOptions);
        PlotResults(BenchmarkResultType.DRAMEnergy, dataSet,
        ↪ plotOptions);
        PlotResults(BenchmarkResultType.Temperature, dataSet,
        ↪ plotOptions);
    }
    ...
}
```

Listing 90: The `PlotAllResults` method in the `BenchmarkPlot` class.

In Listing 90 we can see the `PlotAllResults` method. This method serves to plot the results of the different measurements of a benchmark by calling the `PlotResults` with the relevant `BenchmarkResultType`.

```
public static class BenchmarkPlot {
    ...
    public static void PlotResults(BenchmarkResultType
    ↪ resultType, IBenchmark[] dataSets, PlotOptions?
    ↪ plotOptions = null) {
        DataSet[] data = dataSets.Select(benchmark => new
        ↪ DataSet(benchmark.Name,
        ↪ benchmark.GetResults())).ToArray();
        PlotResults(resultType, data, plotOptions);
    }
    ...
}
```

Listing 91: The `PlotResults` method that sets up data to call an overloaded version of `PlotResults` in the `BenchmarkPlot` class.

In Listing 91 we can see the `PlotResults` method. This method serves to set up the data from each benchmark to properly call the overloaded version of `PlotResults`.

```
public static class BenchmarkPlot {
    ...
    public static void PlotResults(BenchmarkResultType
    ↪ resultType, DataSet[] dataSets, PlotOptions? plotOptions
    ↪ = null) {
        if (!ValidateData(ref dataSets)) {
            return;
        }
        plotOptions ??= new PlotOptions();
        var plt = new Plot(plotOptions.Width,
        ↪ plotOptions.Height);
        dataSets = dataSets.OrderBy(set => set.Name).ToArray();
        string[] names = dataSets.Select(set =>
        ↪ set.Name.Humanize(LetterCasing.Title)).ToArray();
        var hatchIndex = 3;
        foreach ((int index, DataSet dataSet) in
        ↪ dataSets.WithIndex()) {
            double[] plotData = GetPlotData(dataSet,
            ↪ resultType);
            if (plotData.Length == 0) {
                Console.Error.WriteLine($"No data for
                ↪ {resultType} skipping.");
                continue;
            }
            ...
        }
        ...
    }
    ...
}
```

Listing 92: The overloaded version of `PlotResults` which plots and saves the results of a benchmark in the `BenchmarkPlot` class 1/3.

In Listing 92 the first part of the overloaded `PlotResults` method can be seen. We start by validating the data that is sent to the method, by calling the `ValidateData` method with the datasets. We then set `plotOptions` to a new instance of `PlotOptions` if `plotOptions` is `null`. After this, we create a new instance of `Plot` with the correct width and height. The `Plot` class is from the Scottplot library [131]. We order the datasets by their name and humanize the names via the casing of the letters, for example "ForEach" turns into "For Each". We then create a variable for hatch index so we can create plots where each dataset has a different hatch. A hatch is the pattern of a graph, for example a box in a boxplot or a bar in a bargraph [139]. After this, we begin the `foreach` loop which serves to create the actual plot so we can save it to a file afterwards. In this `foreach` loop, we go through all of the datasets, and start by getting the plot data for the dataset relevant. We check if there is any data, if there is no data we skip this plot.

```
1  public static class BenchmarkPlot {
2      ...
3      public static void PlotResults(BenchmarkResultType
       ↪  resultType, DataSet[] dataSets, PlotOptions? plotOptions
       ↪  = null) {
4          ...
5          foreach ((int index, DataSet dataSet) in
           ↪  dataSets.WithIndex()) {
6              ...
7              double min = plotData.Min();
8              double max = plotData.Max();
9              BoxPlot boxPlot = plt.AddBoxPlot(index, plotData,
               ↪  min, max, plotOptions);
10             if (hatchIndex > 9) {
11                 hatchIndex = 0;
12             }
13             if (boxPlot.PlotOptions.UseColorRange) {
14                 boxPlot.PlotOptions.FillColor =
                   ↪  plt.GetSettings().GetNextColor();
15             }
16             boxPlot.PlotOptions.HatchStyle =
               ↪  (HatchStyle)hatchIndex;
17             boxPlot.PlotOptions.HatchColor = Color.Gray;
18             hatchIndex++;
```

```
19          }
20          ...
21      }
22      ...
23  }
```

Listing 93: The overloaded version of `PlotResults` which plots and saves the results of a benchmark in the `BenchmarkPlot` class 2/3.

In Listing 93 we see the next part of the `foreach` loop. Here we find the minimum and maximum values in the data, and call the `AddBoxPlot` method with the index of the dataset, the `plotData`, the minimum value, the maximum value and the `plotOptions` we have created. After this, we check if `hatchIndex` has gone above the max value allowed, in which case we reset it to 0. Next, we check if we want to use color or grayscale, if we want to use color, we set the `FillColor` option to the next color available. At last, we set the hatch style to the index we are currently at, set the color of the hatches to gray and increment the `hatchIndex` by one.

```
1   public static class BenchmarkPlot {
2       ...
3       public static void PlotResults(BenchmarkResultType
        ↪   resultType, DataSet[] dataSets, PlotOptions? plotOptions
        ↪   = null) {
4           ...
5           if (plotOptions.RotateText && names.Max(s => s.Length) >
            ↪   10 && dataSets.Length > 3) {
6               plt.XAxis.TickLabelStyle(rotation: 45);
7           }
8           plt.XTicks(Enumerable.Range(0,
            ↪   dataSets.Length).Select(i1 => (double)i1).ToArray(),
            ↪   names);
9           plt.XLabel("Benchmark");
10          plt.YLabel(GetYLabel(resultType));
11          plt.Title(string.IsNullOrEmpty(plotOptions.Name) ?
            ↪   $"{resultType}" :
            ↪   $"{plotOptions.Name.Humanize(LetterCasing.Title)}");
12          DateTime dateTime = DateTime.Now;
```

```
13          var time = $"{dateTime.ToString("s").Replace(":",
            ↪  "-")}-{dateTime.Millisecond}";
14          Directory.CreateDirectory($"{
            ↪  CsharpRAPLCLI.Options.PlotOutputPath}/{resultType}");
15          plt.SaveFig(string.IsNullOrEmpty(plotOptions.Name) ?
            ↪  $"{CsharpRAPLCLI.Options.PlotOutputPath}/{
            ↪  resultType}/{time}.png" :
            ↪  $"{CsharpRAPLCLI.Options.PlotOutputPath}/{
            ↪  resultType}/{plotOptions.Name}-{time}.png");
16      }
17      ...
18  }
```

Listing 94: The overloaded version of `PlotResults` which plots and saves the results of a benchmark in the `BenchmarkPlot` class 3/3.

Last in the `PlotResults` method, we set up the last parts of the plot. Here we start by checking if the names of the datasets are long and there are more than 3 of them, in which case we rotate the text by 45 degrees so the names can be read on the plot. We then label the datasets with the `XTicks` method from the ScottPlot library [131]. This method puts a small tick at each dataset and puts a label at the dataset with the name of the dataset. After this we label the X and Y-axes with "Benchmark" and the result type of the measurement, for example Package Energy (μJ) by calling the `GetYLabel` method. Then we set up the title of the plot, if the name of the plot is set to null, the title is just the result type, otherwise we humanize the name by the casing of the letters, for example "StringConcat" turns into "String Concat". Then we get a timestamp of the current time, create a directory for where we want to save the plots and save the plot to this directory.

```
1   public static class BenchmarkPlot {
2       ...
3       private static bool ValidateData(ref DataSet[] dataSets) {
4           if (dataSets.Length == 0) {
5               throw new NotSupportedException("Plotting without
                ↪  data is not supported.");
6           }
```

```
7           List<DataSet> dataWhereZero = dataSets.Where(set =>
            ↪ set.Data.Count == 0).ToList();
8           if (dataWhereZero.Count == dataSets.Length) {
9               throw new NotSupportedException("Plotting without
                ↪ data is not supported.");
10          }
11          if (dataWhereZero.Count == 0) {
12              return true;
13          }
14          List<DataSet> newDatasets = dataSets.ToList();
15          foreach (DataSet dataSet in dataWhereZero) {
16              Console.Error.WriteLine($"Skipping plotting
                ↪ {dataSet.Name} since it contains no data.");
17              newDatasets.Remove(dataSet);
18          }
19          dataSets = newDatasets.ToArray();
20          return true;
21      }
22      ...
23  }
```

Listing 95: The `ValidateData` method in the `BenchmarkPlot` class.

In Listing 95 we can see how we validate if we have a dataset that can be plotted. Here we check if the length of the datasets is 0, check if there are as many datasets without data as there are datasets and check if there are any datasets without data. In case there are some datasets without data, we remove these from the array of datasets as plotting no data is unnecessary.

```
1   public static class Helpers {
2       [Pure]
3       public static IEnumerable<(int index, TSource value)>
        ↪ WithIndex<TSource>(this IEnumerable<TSource> enumerable)
        ↪ {
4           return enumerable.Select((value, index) => (index,
            ↪ value));
5       }
6       ...
7   }
```

Listing 96: The extension method `WithIndex`.

In Listing 96 we can see the extension method `WithIndex`. This method extends `IEnumerable` with a method called `WithIndex`, which returns the value with the index in the collection it is called on.

```
1  public static class BenchmarkPlot {
2      ...
3      private static double[] GetPlotData(DataSet dataSet,
       ↪  BenchmarkResultType resultType) {
4          List<double> data = resultType switch {
5              BenchmarkResultType.ElapsedTime =>
               ↪  dataSet.Data.Where(result => result.ElapsedTime
               ↪  > double.Epsilon)
6                  .Select(result => result.ElapsedTime).ToList(),
7              BenchmarkResultType.PackageEnergy =>
               ↪  dataSet.Data.Where(result =>
               ↪  result.PackageEnergy > double.Epsilon)
8                  .Select(result =>
                   ↪  result.PackageEnergy).ToList(),
9              BenchmarkResultType.DRAMEnergy =>
               ↪  dataSet.Data.Where(result => result.DRAMEnergy >
               ↪  double.Epsilon)
10                 .Select(result => result.DRAMEnergy).ToList(),
11             BenchmarkResultType.Temperature =>
               ↪  dataSet.Data.Where(result => result.Temperature
               ↪  > double.Epsilon)
12                 .Select(result => result.Temperature).ToList(),
13             _ => throw new
               ↪  ArgumentOutOfRangeException(nameof(resultType),
               ↪  resultType, null)
14         };
15         return data.ToArray();
16     }
17     ...
18 }
```

Listing 97: The `GetPlotData` method in the `BenchmarkPlot` class.

In Listing 97 we can see the `GetPlotData` method, which returns an array of doubles that contains the data that is asked for. Here we can see that all the measurements that are 0 are filtered away, as RAPL does not have instantaneous sample rate.

```
1  public static class PlotExtensionMethods {
2      public static BoxPlot AddBoxPlot(this Plot plot, double
       ↪  position, double[] data, double errorBelow, double
       ↪  errorAbove, PlotOptions? plotOptions = null, bool
       ↪  autoAxis = true) {
3          PlotOptions plotOpts = plotOptions != null ? new
           ↪  PlotOptions(plotOptions) : new PlotOptions();
4          var boxPlot = new BoxPlot(position, data, errorBelow,
           ↪  errorAbove, plotOpts);
5          plot.Add(boxPlot);
6          if (!autoAxis) {
7              return boxPlot;
8          }
9          plot.AxisAuto();
10         return boxPlot;
11     }
12 }
```

Listing 98: The extension method `AddBoxPlot`.

In Listing 98 we can see the extension method `AddBoxPlot`. Here we start by creating a new set of `PlotOptions`, which we use to create a new `BoxPlot` with the correct parameters. After we have created a new `BoxPlot`, we add it to the plot. If we have `autoAxis` turned off, we return the `boxPlot` now, otherwise we set the axis automatically and then return the `boxPlot`.

```
1  public static class BenchmarkPlot {
2      ...
3      private static string GetYLabel(BenchmarkResultType
       ↪  resultType) {
```

```
4           string yLabel = resultType switch {
5               BenchmarkResultType.ElapsedTime => "Elapsed Time
                ↪  (ms)",
6               BenchmarkResultType.PackageEnergy => "Package Energy
                ↪  (µJ)",
7               BenchmarkResultType.DRAMEnergy => "DRAM Energy
                ↪  (µJ)",
8               BenchmarkResultType.Temperature => "Temperature
                ↪  (C°)",
9               _ => throw new
                ↪  ArgumentOutOfRangeException(nameof(resultType),
                ↪  resultType, null)
10          };
11          return yLabel;
12      }
13      ...
14  }
```

Listing 99: The `GetYLabel` method in the `BenchmarkPlot` class.

In Listing 99 we can see the method `GetYLabel` in the `BenchmarkPlot` class, which serves to return a human readable string that shows what the Y axis represents. For example, for elapsed time this is "Elapsed Time (ms)", both denoting what is being shown and in what unit it is in.

Analysing Results

Now that we have managed to plot the results, we also want to analyse the results so we can make some conclusions. The analysis methods are contained within the `Analysis` class.

```
1   public class Analysis {
2       private readonly DataSet _firstDataset;
3       private readonly DataSet _secondDataset;
4       ...
5   }
```

Listing 100: The `Analysis` class fields.

In Listing 100 we can see the fields in the `Analysis` class. Here there are two fields, `_firstDataset` and `_secondDataset`. These serve to hold the two datasets that are currently being compared so we can check the p-value for them being different.

```
public class DataSet {
    public string Name { get; }
    public List<BenchmarkResult> Data { get; }
    ...
}
```

Listing 101: The `DataSet` class fields.

In Listing 101 we can see the fields in the `DataSet` class. These fields contain the name of the dataset as well as the results of the benchmark relevant to this dataset. This class serves to provide simple helper functions when working with datasets, for example getting the minimum or maximum values that are measured.

```
public class DataSet {
    ...
    public (bool isValid, string message) EnsureResults() {
        List<string> first = Data.Select(result =>
        ↪ result.BenchmarkReturnValue).Distinct().ToList();
        return first.Count switch {
            0 => (false, $"{Name} has no results"),
            > 1 => (false,
                $"Not all results in {Name} was equal. Namely:
                ↪ {first[0]}, {first[1]}{(first.Count > 2 ? "
                ↪ and more." : ".")}"),
            _ => (true, "")
        };
    }
    ...
}
```

Listing 102: The `EnsureResults` method in the `DataSet` class.

In Listing 102 we can see the `EnsureResults` method in the `DataSet` class. This method serves to check if the results in a dataset are consistent, i.e. that the benchmark has returned the same value each time it was run. Here we first check how many different results there are. If there are 0 results we return that the dataset is invalid and that there are no results. If there are more than 1 result, we return that the dataset is invalid and that the results are inconsistent between at least 2 values. Otherwise, the results are valid and we return that they are without a message.

```
public class Analysis {
    ...
    public (bool isValid, string message) EnsureResults() {
        (bool isValid, string message) first =
        ↪ _firstDataset.EnsureResults();
        if (!first.isValid) {
            return first;
        }
        (bool isValid, string message) second =
        ↪ _secondDataset.EnsureResults();
        if (!second.isValid) {
            return second;
        }
        return (true, "");
    }
    ...
}
```

Listing 103: The `EnsureResults` method in the `Analysis` class.

In Listing 103, we can see the `EnsureResults` method in the `Analysis` class. This method checks if both of the individual datasets are valid by calling the `EnsureResults` method on the datasets, as seen in Listing 102. If either of them are invalid, the first invalid result is returned so it can be dealt with.

```
public class Analysis {
    ...
    public (bool isValid, string message)
    ↪ EnsureResultsMutually() {
```

```
4           List<string> first = _firstDataset.Data.Select(result =>
            ↪  result.BenchmarkReturnValue).Distinct().ToList();
5           List<string> second = _secondDataset.Data.Select(result
            ↪  => result.BenchmarkReturnValue).Distinct().ToList();
6           (bool isValid, string message) firstEnsure =
            ↪  _firstDataset.EnsureResults();
7           if (!firstEnsure.isValid) {
8               return firstEnsure;
9           }
10          (bool isValid, string message) secondEnsure =
            ↪  _secondDataset.EnsureResults();
11          if (!secondEnsure.isValid) {
12              return secondEnsure;
13          }
14          ...
15      }
16      ...
17  }
```

Listing 104: The `EnsureResultsMutually` method in the `Analysis` class 1/2.

In Listing 104, we can see the first part of the `EnsureResultsMutually` method in the `Analysis` class. Here we check if each of the datasets are valid individually, if they are not we return the first invalid result is returned like in Listing 103.

```
1   public class Analysis {
2       ...
3       public (bool isValid, string message)
        ↪  EnsureResultsMutually() {
4           ...
5           if (first.Count != second.Count) {
6               return (false,
7                   $"The two data sets have an unequal number of
                    ↪  results. {_firstDataset.Name}:
                    ↪  [{string.Join(", ", first)}],
                    ↪  {_secondDataset.Name}: [{string.Join(", ",
                    ↪  second)}]");
8           }
```

```
9           for (var i = 0; i < first.Count; i++) {
10              if (first[i] != second[i]) {
11                  return (false, $"The to datasets differ in
                        ↪ {first[i]} and {second[i]}");
12              }
13          }
14          return (true, "");
15      }
16      ...
17  }
```

Listing 105: The `EnsureResultsMutually` method in the `Analysis` class 2/2.

In Listing 105, we can see the second part of the `EnsureResultsMutually` method in the `Analysis` class. Here we check if there is an equal number of results in each of the datasets and then afterwards check if each of the results are equal, in which case we return `true` without a message, but otherwise return `false` with a message that shows where they are different.

```
1   public class Analysis {
2       ...
3       public static Dictionary<string, double>
        ↪ CalculatePValueForGroup(List<IBenchmark> dataSets) {
4           var groupToPValue = new Dictionary<string, double>();
5           for (int i = 0; i < dataSets.Count; i++) {
6               for (int j = i + 1; j < dataSets.Count; j++) {
7                   var analysis = new Analysis(dataSets[i],
                        ↪ dataSets[j]);
8                   foreach ((string message, double value) in
                        ↪ analysis.CalculatePValue()) {
9                       groupToPValue.Add(message, value);
10                  }
11              }
12          }
13          return groupToPValue;
14      }
15  }
```

Listing 106: Calculating the P-value for a group of results in the `Analysis` class.

In Listing 106 we can see how a list of benchmarks has their p-value calculated between them all with the `CalculatePValueForGroup` method. We do this by going through all of the datasets and creating a new `Analysis` object with the datasets we are at. After this we call the `CalculatePValue` method on them, which we save the return value of.

```
1  public class Analysis {
2      ...
3      public List<(string Message, double Value)>
       ↪  CalculatePValue() {
4          PValueData firstDataSet = new(_firstDataset);
5          PValueData secondDataSet = new(_secondDataset);
6          var timeTTest = new
           ↪  TwoSampleTTest(firstDataSet.TimesValues,
           ↪  secondDataSet.TimesValues);
7          var pkgTTest = new
           ↪  TwoSampleTTest(firstDataSet.PackageValues,
           ↪  secondDataSet.PackageValues);
8          var dramTTest = new
           ↪  TwoSampleTTest(firstDataSet.DRAMValues,
           ↪  secondDataSet.DRAMValues);
9          return new List<(string Message, double Value)> {
10             ($"{firstDataSet.Name} significantly different from
               ↪  {secondDataSet.Name} - Time", timeTTest.PValue),
11             ($"{firstDataSet.Name} significantly different from
               ↪  {secondDataSet.Name} - Package",
               ↪  pkgTTest.PValue),
12             ($"{firstDataSet.Name} significantly different from
               ↪  {secondDataSet.Name} - DRAM", dramTTest.PValue)
13         };
14     }
15     ...
16 }
```

Listing 107: Calculating the P-value between two datasets in the `Analysis` class.

In Listing 107 we can see the `CalculatePValue` method. This method serves to calculate the p-value for two different datasets. We start this by

setting up two `PValueData` objects with the relevant datasets, after this we create three `TwoSampleTTest` objects with the running time measured, the package energy measured and the DRAM energy measured. At the end we create a list of tuples with strings and numbers, the string serves to make it possible to read what the number means, the number is the actual p-value between the two sets of data.

```
1  public class PValueData {
2      public readonly string Name;
3      public readonly double[] TimesValues;
4      public readonly double[] PackageValues;
5      public readonly double[] DRAMValues;
6      public PValueData(DataSet dataSet) {
7          Name = dataSet.Name;
8          TimesValues = dataSet.Data.Select(data =>
           ↪ data.ElapsedTime).ToArray();
9          PackageValues = dataSet.Data.Select(data =>
           ↪ data.PackageEnergy).ToArray();
10         DRAMValues = dataSet.Data.Select(data =>
           ↪ data.DRAMEnergy).ToArray();
11     }
12 }
```

Listing 108: The `PValueData` class.

In Listing 108 we can see the `PValueData` class. This class serves to keep track of the name of the dataset, as well as the different measurements for this dataset, specifically for the running time measured during the benchmark, the package energy measured during the benchmark and the DRAM energy measured during the benchmark. Important to note is that the fields `TimesValues`, `PackageValues` and `DRAMValues` are arrays, as arrays are needed to instantiate the `TwoSampleTTest` objects.

A.3 All Results

A.3.1 Primitive Types

Integer Types

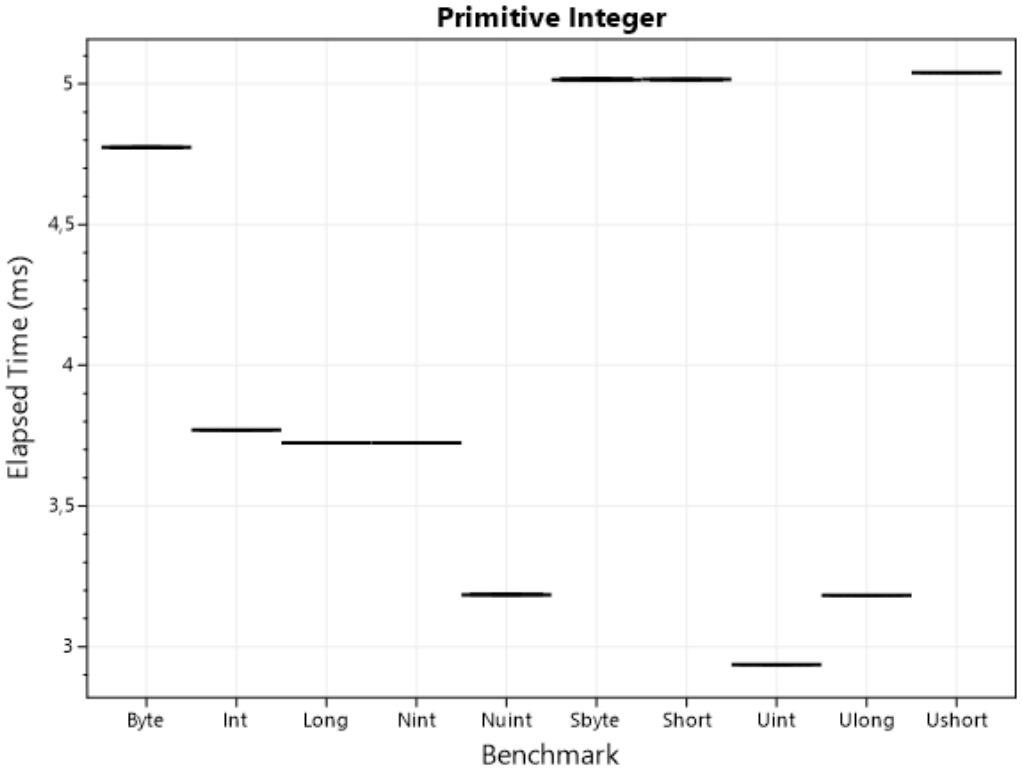

Figure A.5: Boxplot showing how efficient different Primitive Integer types are with regards to Elapsed time.

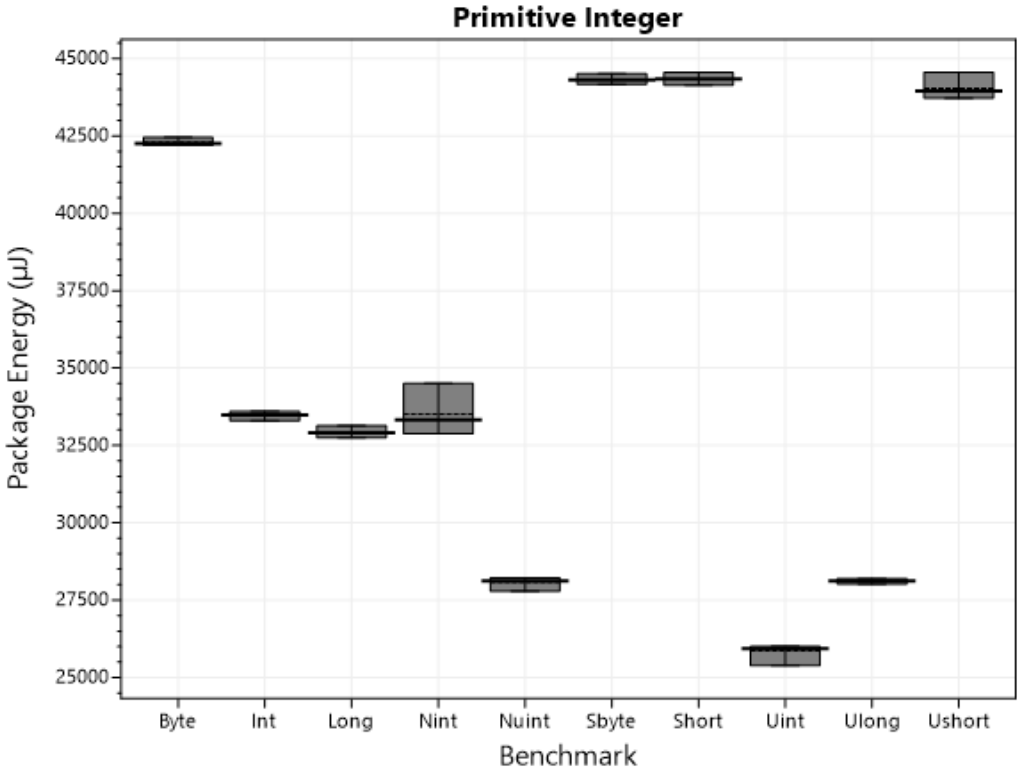

Figure A.6: Boxplot showing how efficient different Primitive Integer types are with regards to Package Energy.

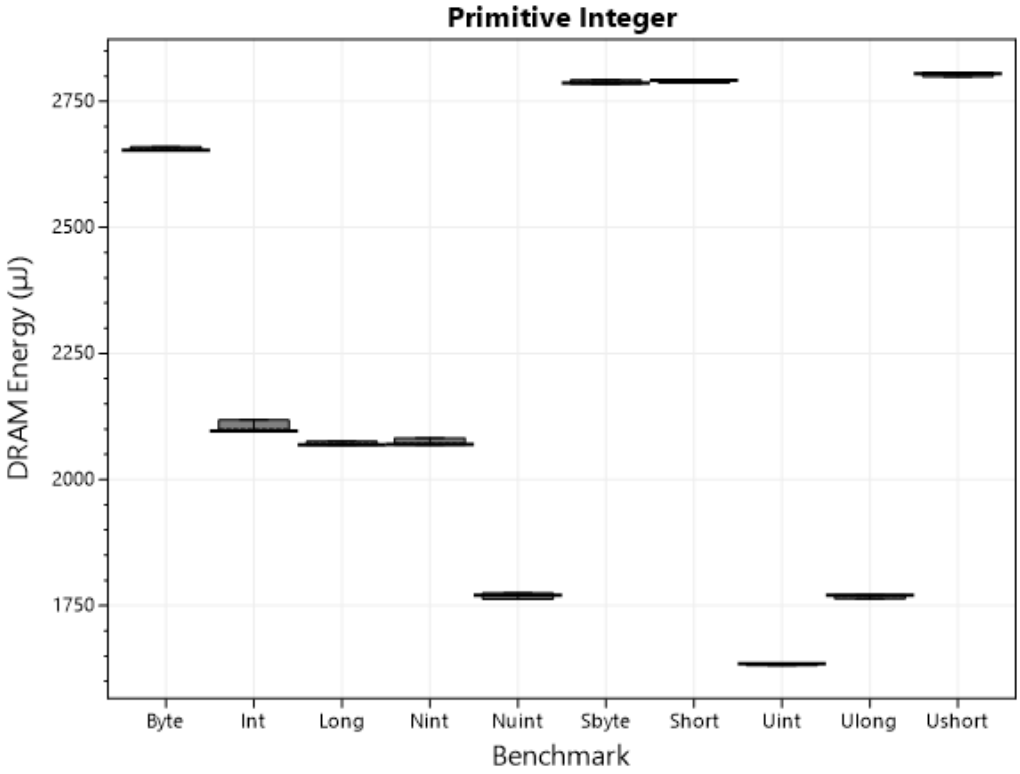

Figure A.7: Boxplot showing how efficient different Primitive Integer types are with regards to Dram energy.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Byte	4,774144	42.291,558	2.654,990
Int	3,770063	33.451,857	2.100,078
Long	3,723723	32.924,190	2.071,137
Nint	3,723477	33.503,079	2.071,678
Nuint	3,184053	28.078,657	1.769,831
Sbyte	5,014215	44.328,415	2.788,301
Short	5,015832	44.327,446	2.790,484
Uint	2,935853	25.862,204	1.633,408
Ulong	3,183120	28.112,855	1.768,374
Ushort	5,040331	44.033,497	2.804,712

Table A.1: Table showing the elapsed time and energy measurement for each Primitive Integer.

Elapsed Time *p*-Values	Int	Uint	Nint	Nuint	Long	Ulong	Short	Ushort	Byte	Sbyte
Int	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Uint	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Nint	<0,05	<0,05	-	<0,05	0,123	<0,05	<0,05	<0,05	<0,05	<0,05
Nuint	<0,05	<0,05	<0,05	-	<0,05	0,171	<0,05	<0,05	<0,05	<0,05
Long	<0,05	<0,05	0,123	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Ulong	<0,05	<0,05	<0,05	0,171	<0,05	-	<0,05	<0,05	<0,05	<0,05
Short	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	0,249
Ushort	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Byte	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05
Sbyte	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	0,249	<0,05	<0,05	-

Table A.2: Table showing the *p*-values for the group Primitive Integer with regards to Elapsed Time.

Package Energy *p*-Values	Int	Uint	Nint	Nuint	Long	Ulong	Short	Ushort	Byte	Sbyte
Int	-	<0,05	0,751	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Uint	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Nint	0,751	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Nuint	<0,05	<0,05	<0,05	-	<0,05	0,498	<0,05	<0,05	<0,05	<0,05
Long	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Ulong	<0,05	<0,05	<0,05	0,498	<0,05	-	<0,05	<0,05	<0,05	<0,05
Short	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	0,985
Ushort	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Byte	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05
Sbyte	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	0,985	<0,05	<0,05	-

Table A.3: Table showing the *p*-values for the group Primitive Integer with regards to Package Energy.

DRAM Energy *p*-Values	Int	Uint	Nint	Nuint	Long	Ulong	Short	Ushort	Byte	Sbyte
Int	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Uint	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Nint	<0,05	<0,05	-	<0,05	0,687	<0,05	<0,05	<0,05	<0,05	<0,05
Nuint	<0,05	<0,05	<0,05	-	<0,05	0,380	<0,05	<0,05	<0,05	<0,05
Long	<0,05	<0,05	0,687	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Ulong	<0,05	<0,05	<0,05	0,380	<0,05	-	<0,05	<0,05	<0,05	<0,05
Short	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	0,164
Ushort	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Byte	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05
Sbyte	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	0,164	<0,05	<0,05	-

Table A.4: Table showing the *p*-values for the group Primitive Integer with regards to DRAM Energy.

Floating Point Types

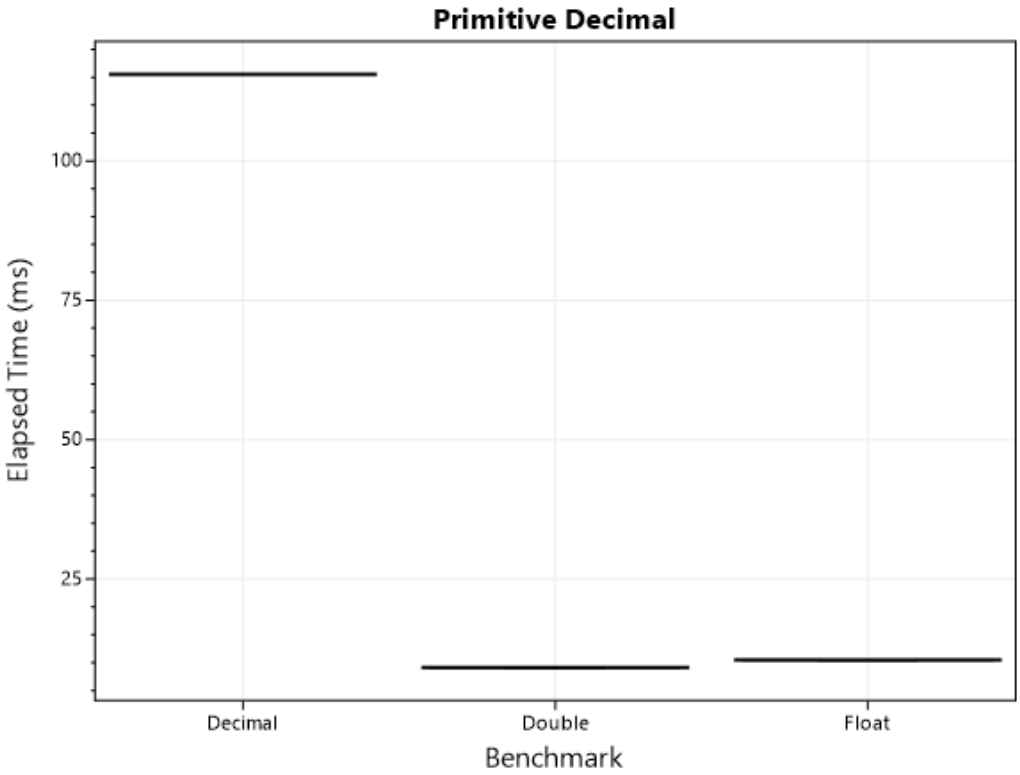

Figure A.8: Boxplot showing how efficient different Primitive Decimal types are with regards to Elapsed Time.

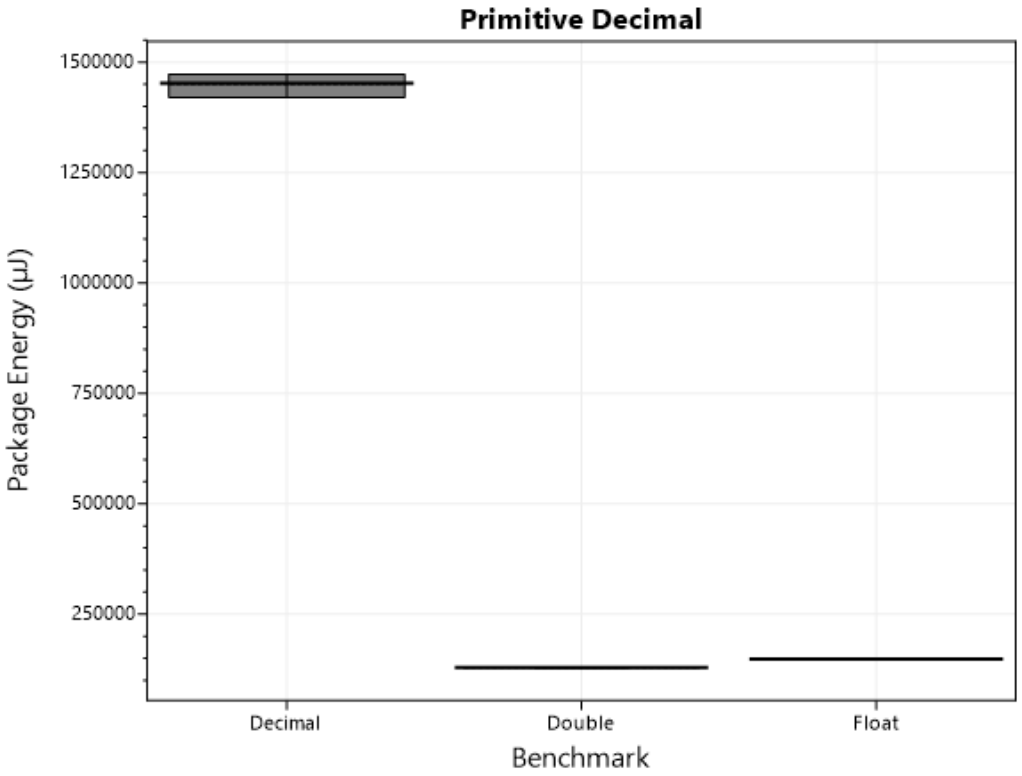

Figure A.9: Boxplot showing how efficient different Primitive Decimal types are with regards to Package Energy.

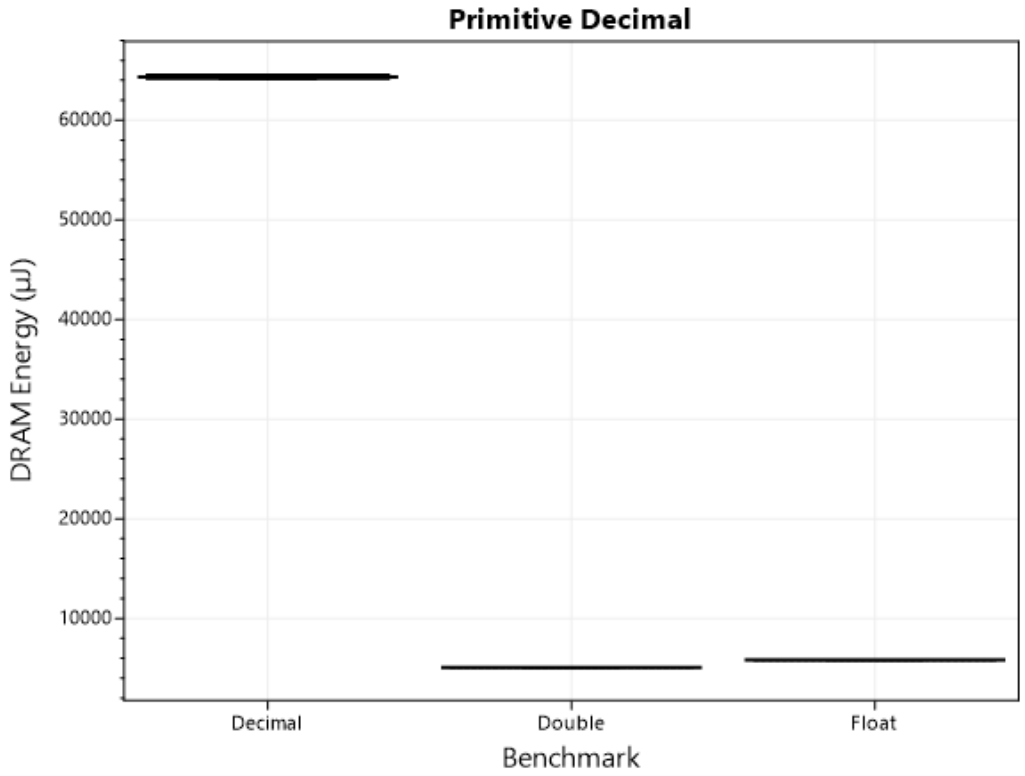

Figure A.10: Boxplot showing how efficient different Primitive Decimal types are with regards to Dram Energy.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Decimal	115,533295	1.450.443,888	64.285,851
Double	9,105950	129.109,597	5.067,126
Float	10,435074	148.072,794	5.803,612

Table A.5: Table showing the elapsed time and energy measurement for each Primitive Decimal.

Elapsed Time *p*-Values	Float	Double	Decimal
Float	-	<0,05	<0,05
Double	<0,05	-	<0,05
Decimal	<0,05	<0,05	-

Table A.6: Table showing the *p*-values for the group Primitive Decimal with regards to Elapsed Time.

Package Energy *p*-Values	Float	Double	Decimal
Float	-	<0,05	<0,05
Double	<0,05	-	<0,05
Decimal	<0,05	<0,05	-

Table A.7: Table showing the *p*-values for the group Primitive Decimal with regards to Package Energy.

DRAM Energy *p*-Values	Float	Double	Decimal
Float	-	<0,05	<0,05
Double	<0,05	-	<0,05
Decimal	<0,05	<0,05	-

Table A.8: Table showing the *p*-values for the group Primitive Decimal with regards to DRAM Energy.

Booleans

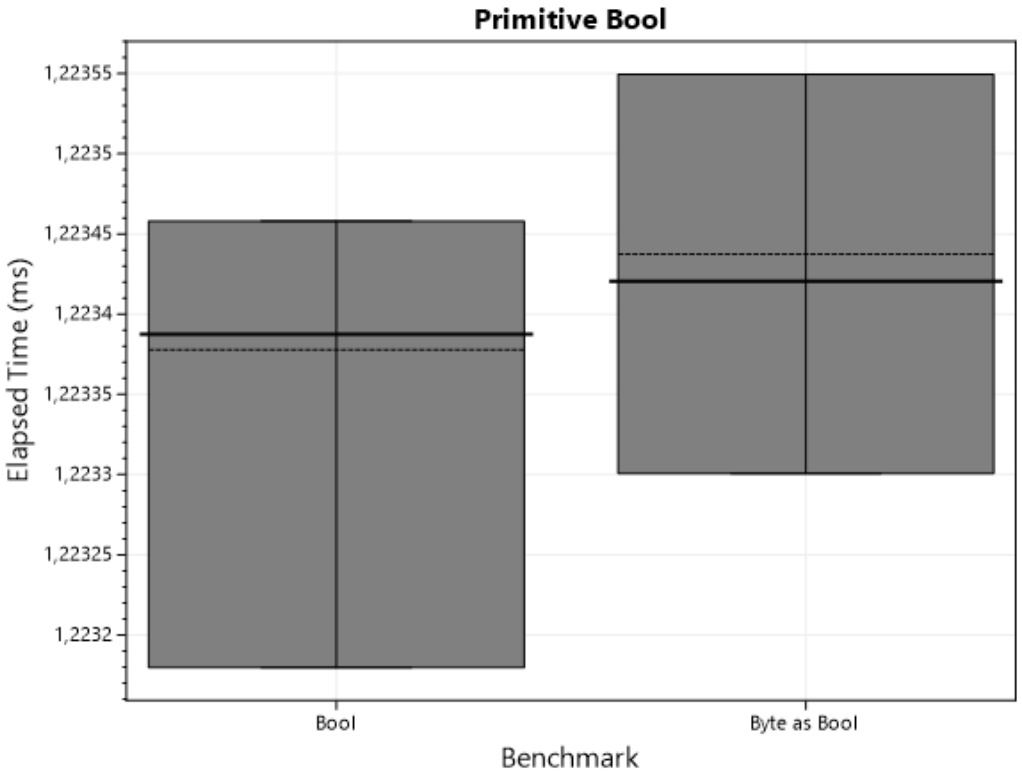

Figure A.11: Boxplot showing how efficient different Primitive Bool types are with regards to Elapsed Time.

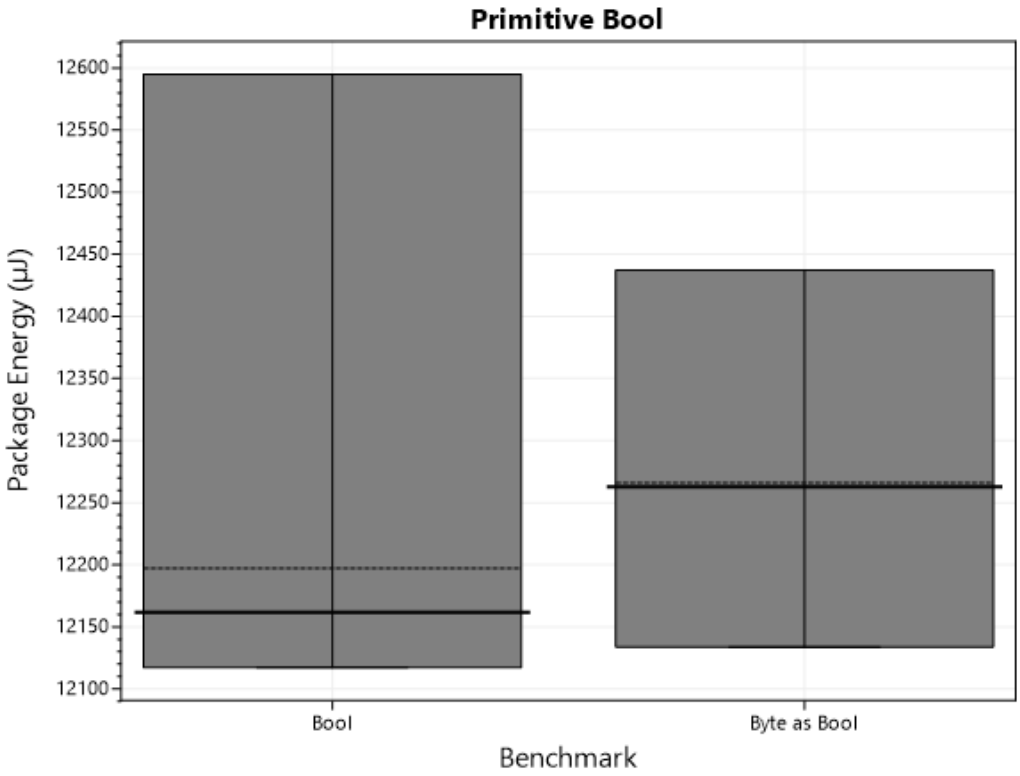

Figure A.12: Boxplot showing how efficient different Primitive Bool types are with regards to Package Energy.

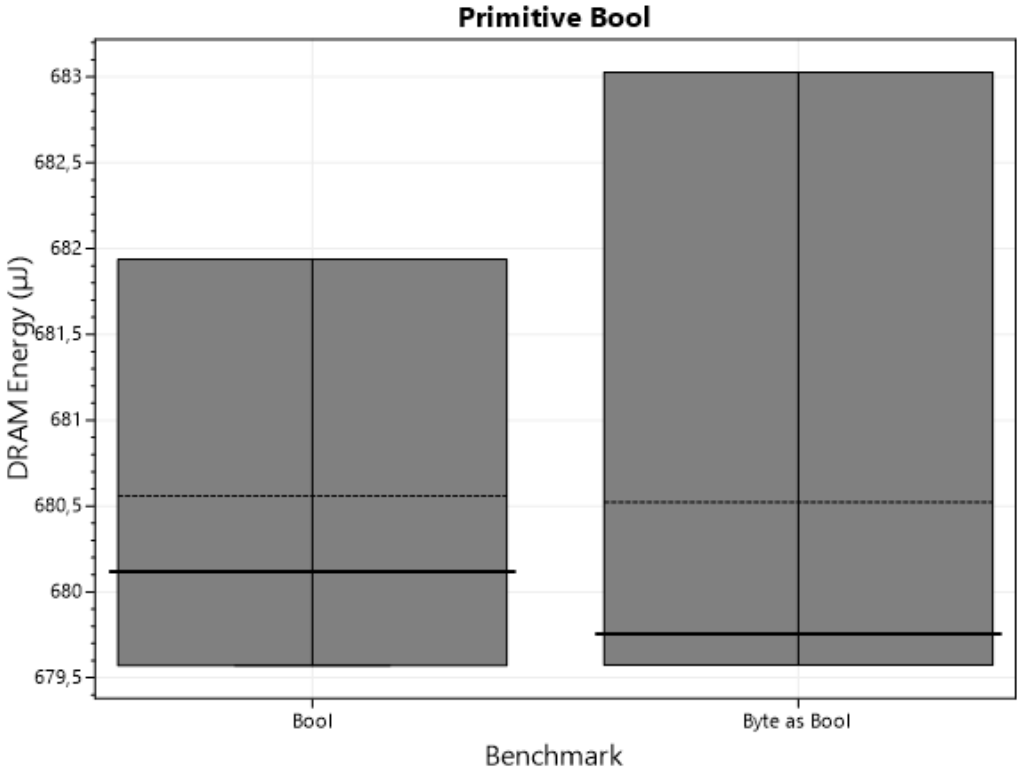

Figure A.13: Boxplot showing how efficient different Primitive Bool types are with regards to Dram Energy.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Bool	1,223378	12.197,154	680,559
Byte as Bool	1,223437	12.265,960	680,523

Table A.9: Table showing the elapsed time and energy measurement for each Primitive Bool.

Elapsed Time *p*-Values	Bool	Byte as Bool
Bool	-	<0,05
Byte as Bool	<0,05	-

Table A.10: Table showing the *p*-values for the group Primitive Bool with regards to Elapsed Time.

Package Energy *p*-Values	Bool	Byte as Bool
Bool	-	0,105
Byte as Bool	0,105	-

Table A.11: Table showing the *p*-values for the group Primitive Bool with regards to Package Energy.

DRAM Energy *p*-Values	Bool	Byte as Bool
Bool	-	0,932
Byte as Bool	0,932	-

Table A.12: Table showing the *p*-values for the group Primitive Bool with regards to DRAM Energy.

A.3.2 Operations

Addition

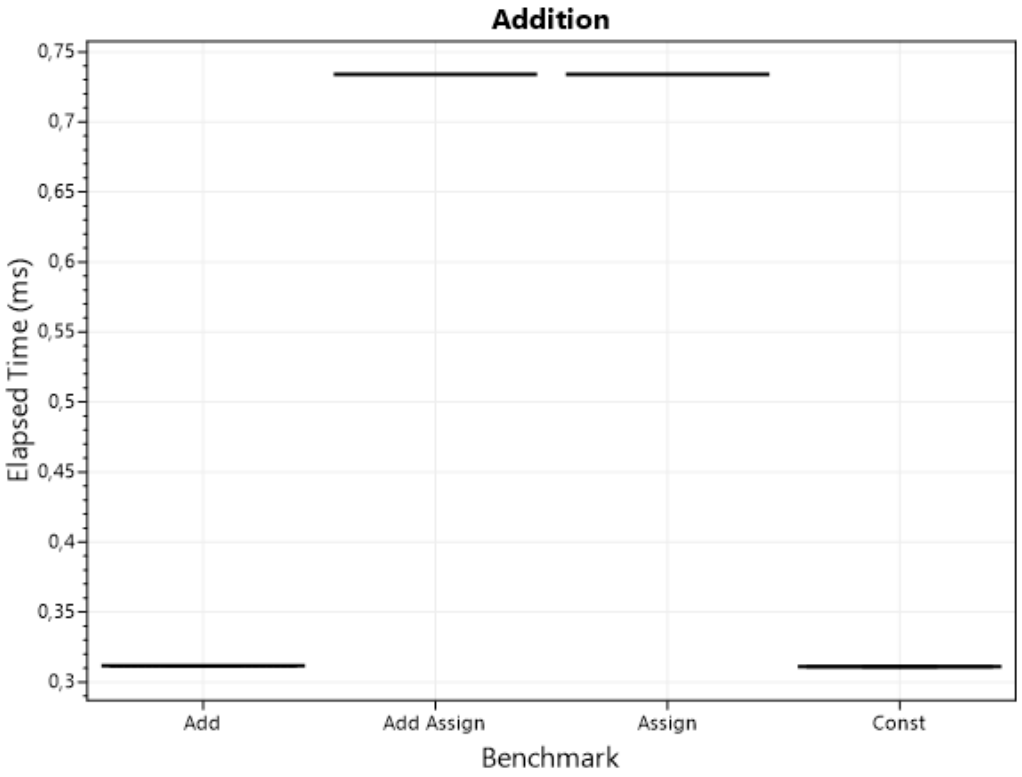

Figure A.14: Boxplot showing how efficient different Addition operations are with regards to Elapsed Time.

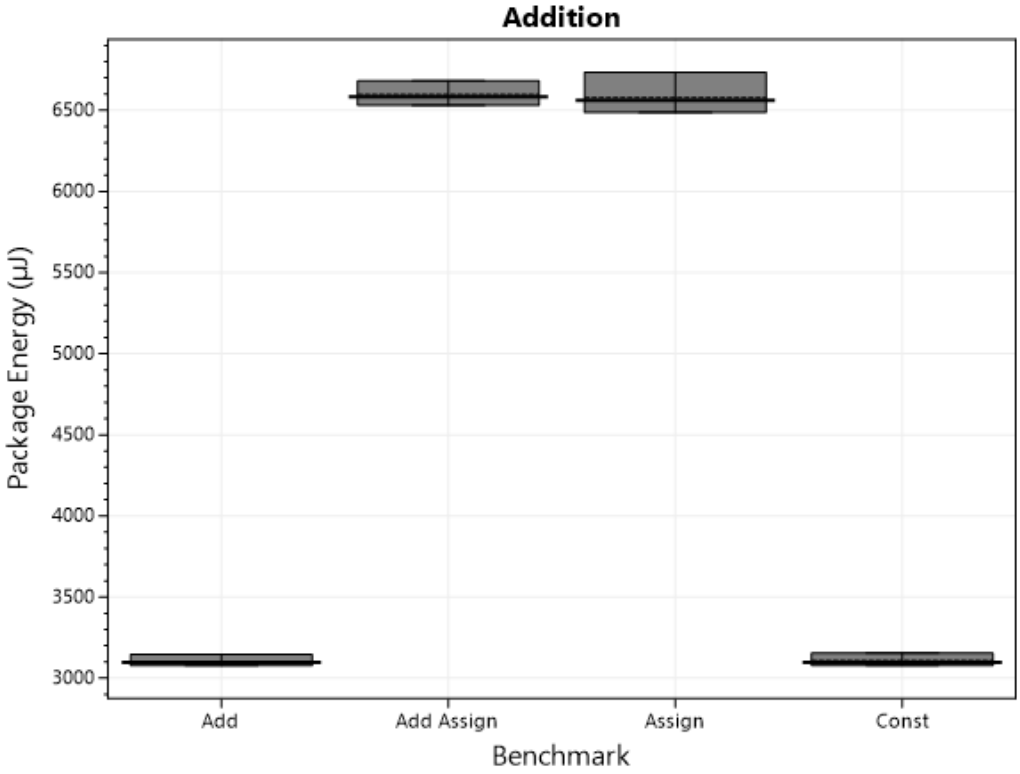

Figure A.15: Boxplot showing how efficient different Addition operations are with regards to Package Energy.

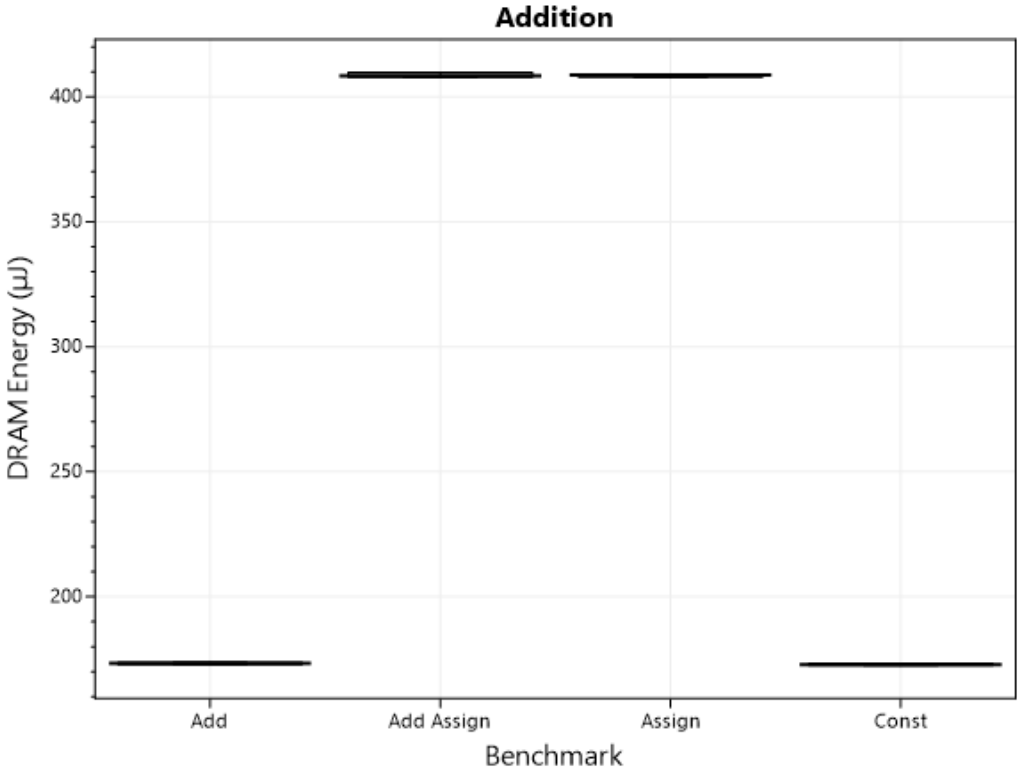

Figure A.16: Boxplot showing how efficient different Addition operations are with regards to Dram Energy.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Add	0,311497	3.103,862	173,250
Add Assign	0,733963	6.596,394	408,362
Assign	0,734009	6.578,976	408,300
Const	0,310839	3.108,339	172,874

Table A.13: Table showing the elapsed time and energy measurement for each Addition.

Elapsed Time *p*-Values	Add	Const	Add Assign	Assign
Add	-	<0,05	<0,05	<0,05
Const	<0,05	-	<0,05	<0,05
Add Assign	<0,05	<0,05	-	<0,05
Assign	<0,05	<0,05	<0,05	-

Table A.14: Table showing the *p*-values for the group Addition with regards to Elapsed Time.

Package Energy *p*-Values	Add	Const	Add Assign	Assign
Add	-	0,615	<0,05	<0,05
Const	0,615	-	<0,05	<0,05
Add Assign	<0,05	<0,05	-	0,510
Assign	<0,05	<0,05	0,510	-

Table A.15: Table showing the *p*-values for the group Addition with regards to Package Energy.

DRAM Energy *p*-Values	Add	Const	Add Assign	Assign
Add	-	<0,05	<0,05	<0,05
Const	<0,05	-	<0,05	<0,05
Add Assign	<0,05	<0,05	-	0,764
Assign	<0,05	<0,05	0,764	-

Table A.16: Table showing the *p*-values for the group Addition with regards to DRAM Energy.

Subtraction

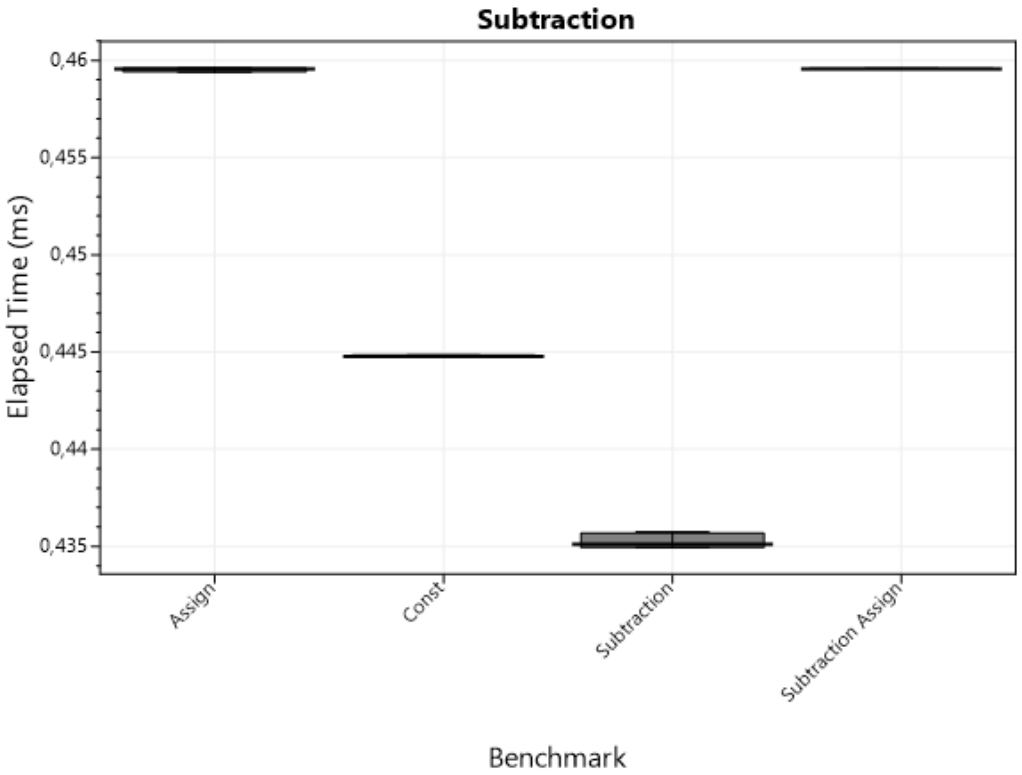

Figure A.17: Boxplot showing how efficient different Subtraction operations are with regards to Elapsed Time.

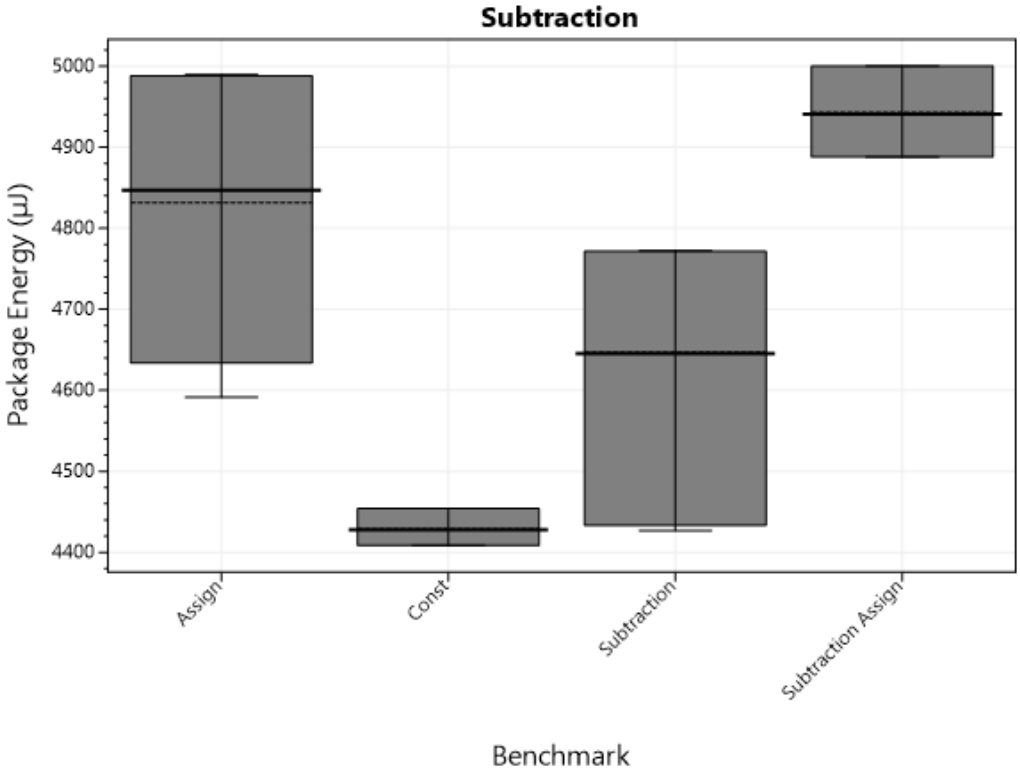

Figure A.18: Boxplot showing how efficient different Subtraction operations are with regards to Package Energy.

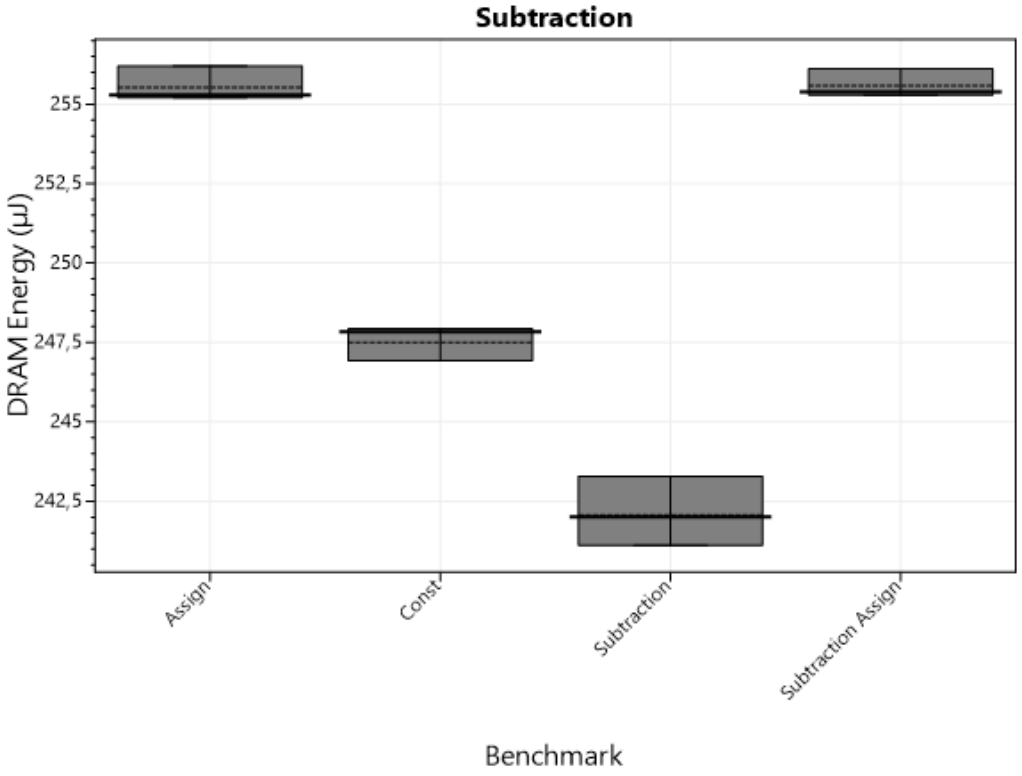

Figure A.19: Boxplot showing how efficient different Subtraction operations are with regards to Dram Energy.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Assign	0,459557	4.831,602	255,527
Const	0,444774	4.429,046	247,493
Subtraction	0,435115	4.646,768	242,061
Subtraction Assign	0,459570	4.942,778	255,587

Table A.17: Table showing the elapsed time and energy measurement for each Subtraction.

Elapsed Time *p*-Values	Subtraction	Const	Subtraction Assign	Assign
Subtraction	-	<0,05	<0,05	<0,05
Const	<0,05	-	<0,05	<0,05
Subtraction Assign	<0,05	<0,05	-	0,206
Assign	<0,05	<0,05	0,206	-

Table A.18: Table showing the *p*-values for the group Subtraction with regards to Elapsed Time.

Package Energy *p*-Values	Subtraction	Const	Subtraction Assign	Assign
Subtraction	-	<0,05	<0,05	<0,05
Const	<0,05	-	<0,05	<0,05
Subtraction Assign	<0,05	<0,05	-	<0,05
Assign	<0,05	<0,05	<0,05	-

Table A.19: Table showing the *p*-values for the group Subtraction with regards to Package Energy.

DRAM Energy *p*-Values	Subtraction	Const	Subtraction Assign	Assign
Subtraction	-	<0,05	<0,05	<0,05
Const	<0,05	-	<0,05	<0,05
Subtraction Assign	<0,05	<0,05	-	0,671
Assign	<0,05	<0,05	0,671	-

Table A.20: Table showing the *p*-values for the group Subtraction with regards to DRAM Energy.

Multiplication

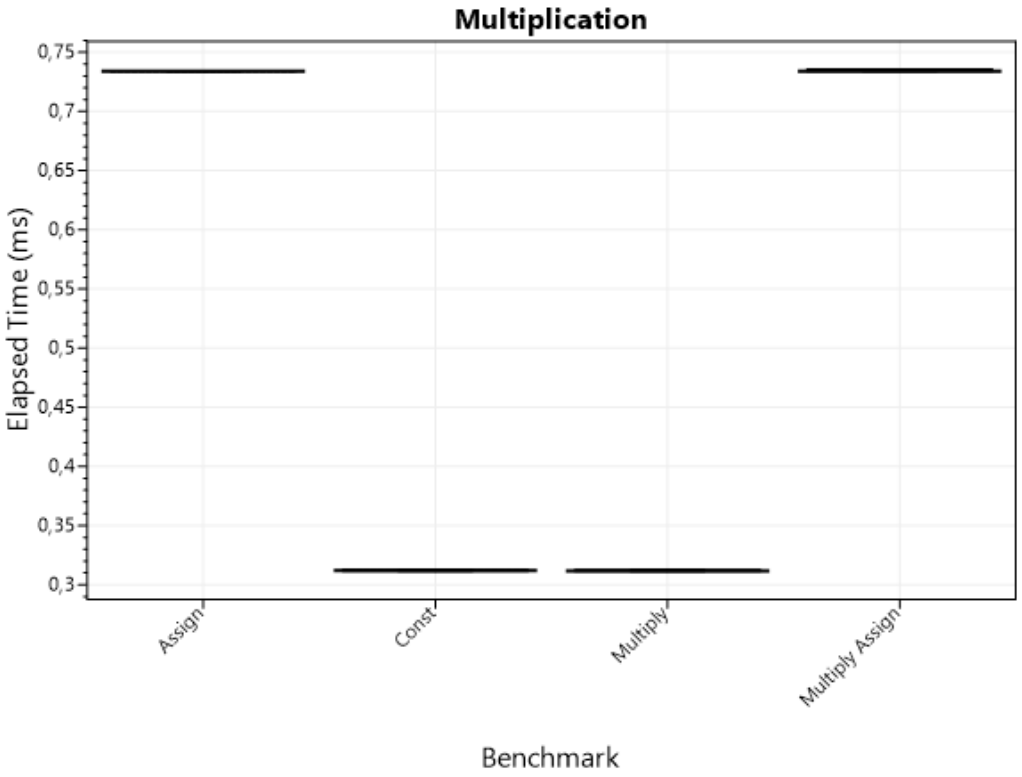

Figure A.20: Boxplot showing how efficient different Multiplication operations are with regards to Elapsed Time.

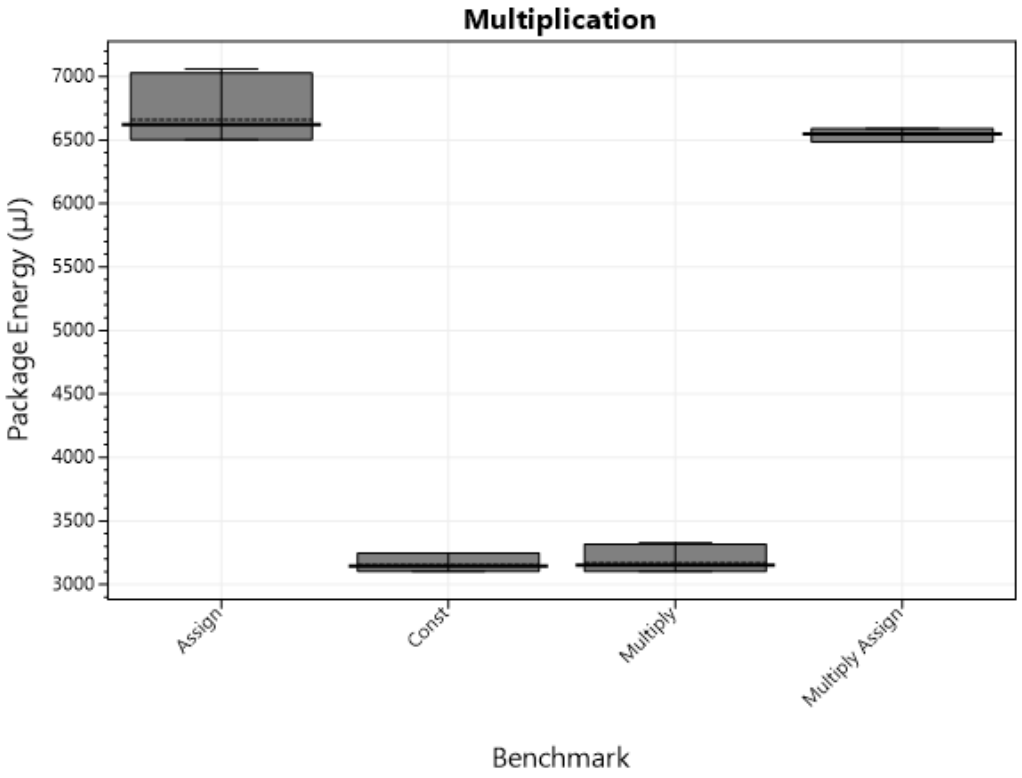

Figure A.21: Boxplot showing how efficient different Multiplication operations are with regards to Package Energy.

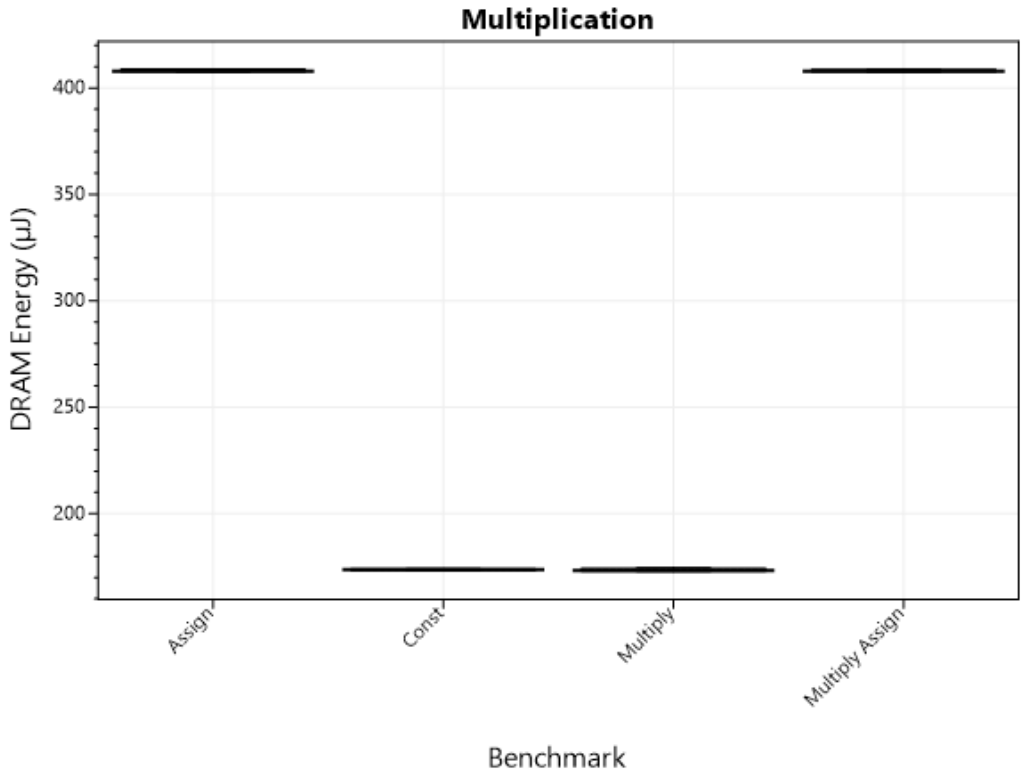

Figure A.22: Boxplot showing how efficient different Multiplication operations are with regards to Dram Energy.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Assign	0,734010	6.658,213	408,203
Const	0,312009	3.156,558	173,616
Multiply	0,311845	3.168,088	173,460
Multiply Assign	0,734220	6.541,948	408,089

Table A.21: Table showing the elapsed time and energy measurement for each Multiplication.

Elapsed Time *p*-Values	Multiply	Const	Multiply Assign	Assign
Multiply	-	0,094	<0,05	<0,05
Const	0,094	-	<0,05	<0,05
Multiply Assign	<0,05	<0,05	-	<0,05
Assign	<0,05	<0,05	<0,05	-

Table A.22: Table showing the *p*-values for the group Multiplication with regards to Elapsed Time.

Package Energy *p*-Values	Multiply	Const	Multiply Assign	Assign
Multiply	-	0,375	<0,05	<0,05
Const	0,375	-	<0,05	<0,05
Multiply Assign	<0,05	<0,05	-	<0,05
Assign	<0,05	<0,05	<0,05	-

Table A.23: Table showing the *p*-values for the group Multiplication with regards to Package Energy.

DRAM Energy *p*-Values	Multiply	Const	Multiply Assign	Assign
Multiply	-	0,056	<0,05	<0,05
Const	0,056	-	<0,05	<0,05
Multiply Assign	<0,05	<0,05	-	0,463
Assign	<0,05	<0,05	0,463	-

Table A.24: Table showing the *p*-values for the group Multiplication with regards to DRAM Energy.

Division

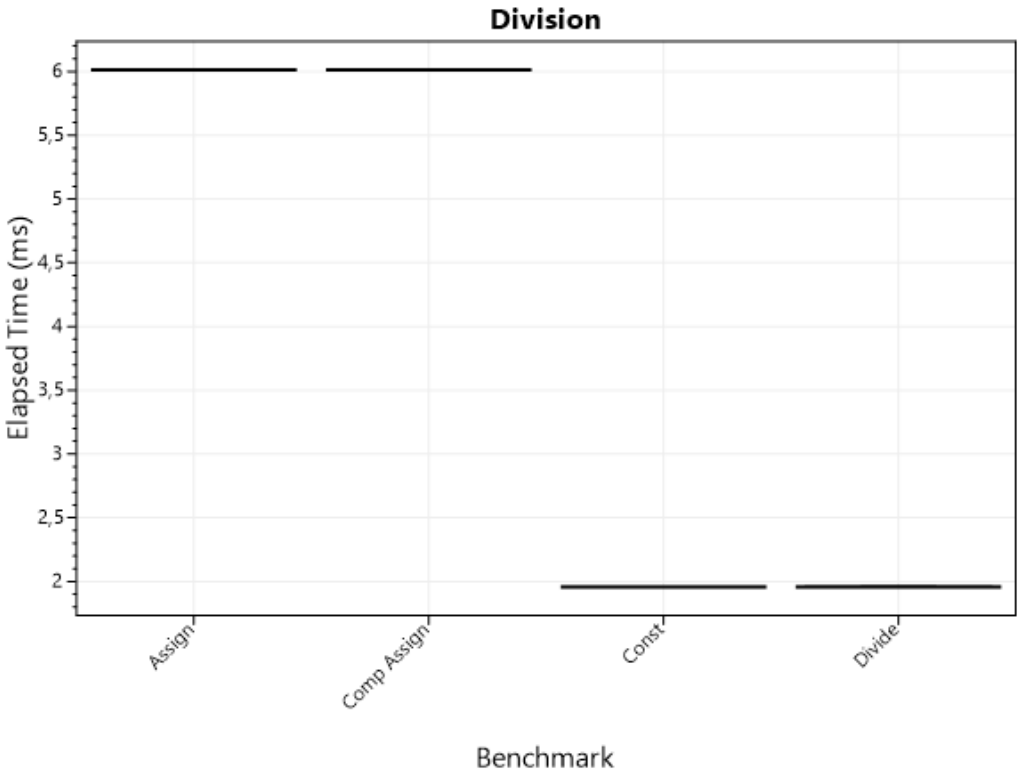

Figure A.23: Boxplot showing how efficient different Division operations are with regards to Elapsed Time.

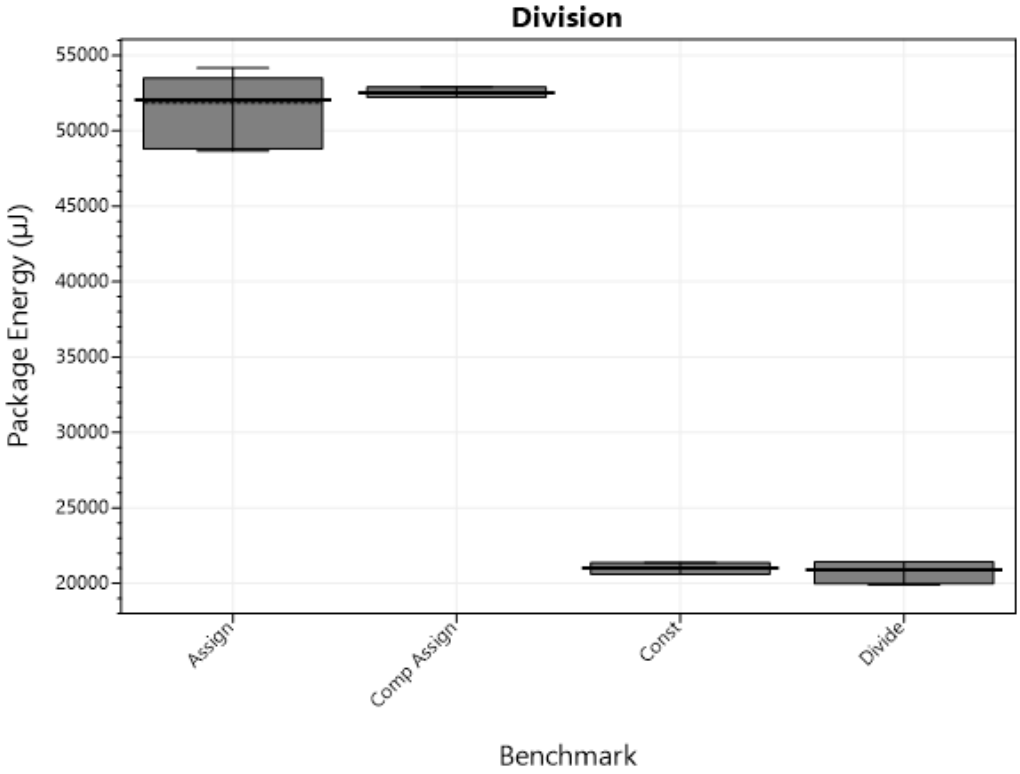

Figure A.24: Boxplot showing how efficient different Division operations are with regards to Package Energy.

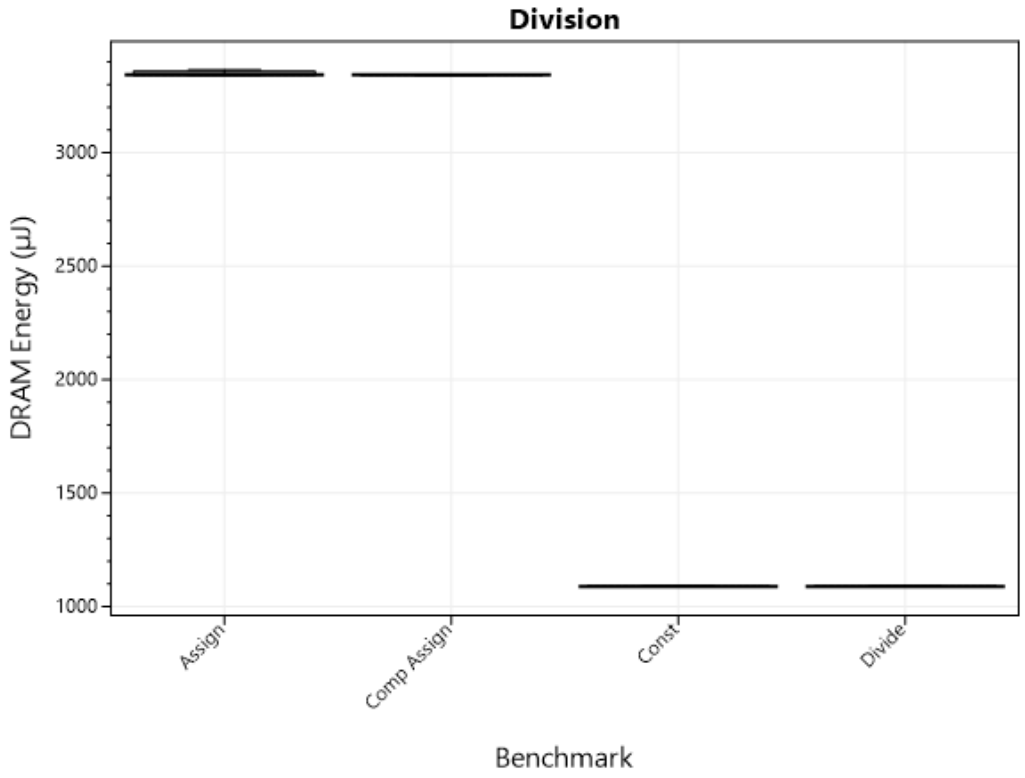

Figure A.25: Boxplot showing how efficient different Division operations are with regards to Dram Energy.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Assign	6,011532	51.841,695	3.343,327
Comp Assign	6,011435	52.574,642	3.342,242
Const	1,957693	21.028,555	1.088,716
Divide	1,957883	20.849,585	1.088,613

Table A.25: Table showing the elapsed time and energy measurement for each Division.

Elapsed Time *p*-Values	Divide	Const	Comp Assign	Assign
Divide	-	<0,05	<0,05	<0,05
Const	<0,05	-	<0,05	<0,05
Comp Assign	<0,05	<0,05	-	0,622
Assign	<0,05	<0,05	0,622	-

Table A.26: Table showing the *p*-values for the group Division with regards to Elapsed Time.

Package Energy *p*-Values	Divide	Const	Comp Assign	Assign
Divide	-	0,099	<0,05	<0,05
Const	0,099	-	<0,05	<0,05
Comp Assign	<0,05	<0,05	-	<0,05
Assign	<0,05	<0,05	<0,05	-

Table A.27: Table showing the *p*-values for the group Division with regards to Package Energy.

DRAM Energy *p*-Values	Divide	Const	Comp Assign	Assign
Divide	-	0,815	<0,05	<0,05
Const	0,815	-	<0,05	<0,05
Comp Assign	<0,05	<0,05	-	0,458
Assign	<0,05	<0,05	0,458	-

Table A.28: Table showing the *p*-values for the group Division with regards to DRAM Energy.

Modulo

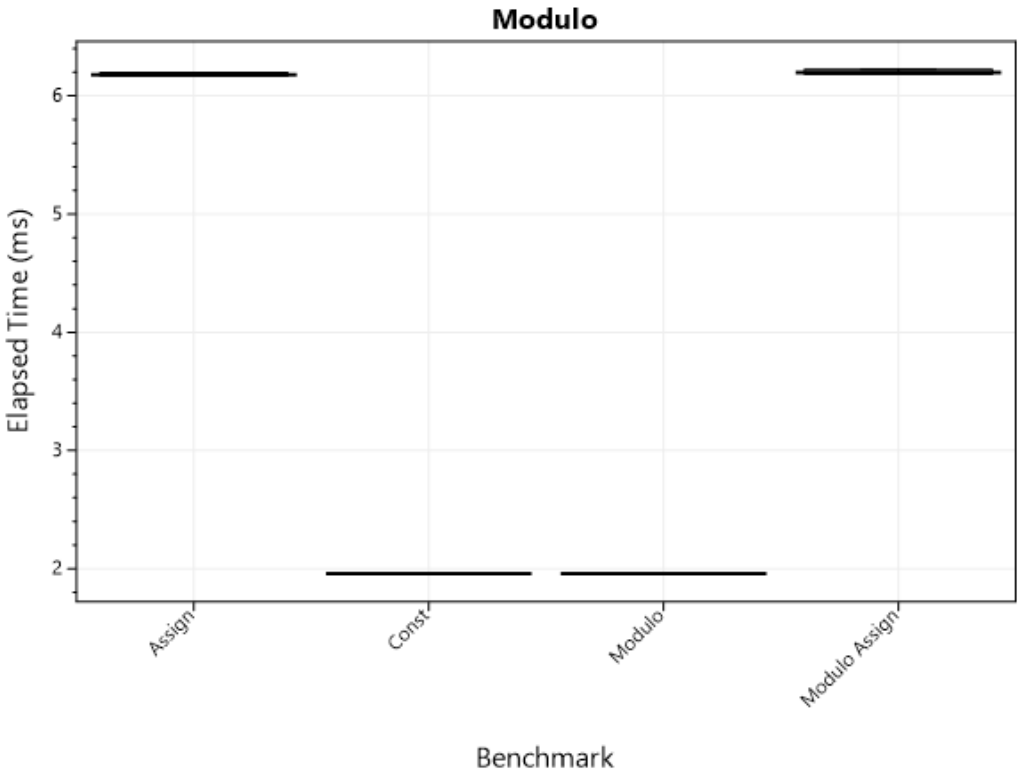

Figure A.26: Boxplot showing how efficient different Modulo types are with regards to Elapsed Time.

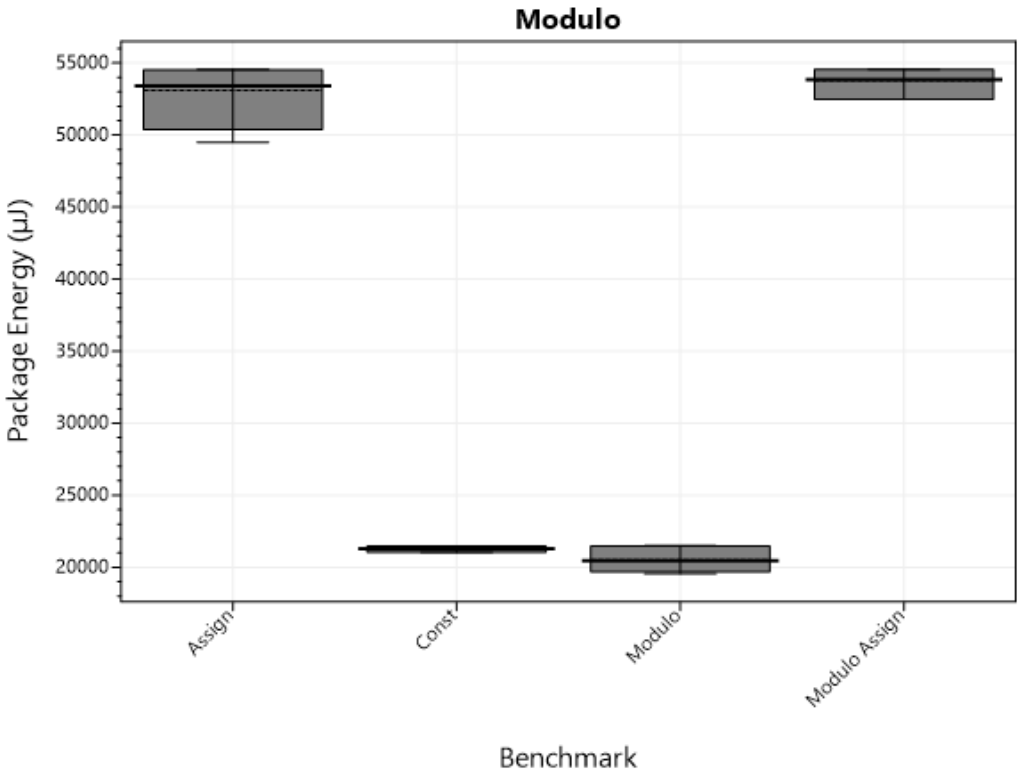

Figure A.27: Boxplot showing how efficient different Modulo types are with regards to Package Energy.

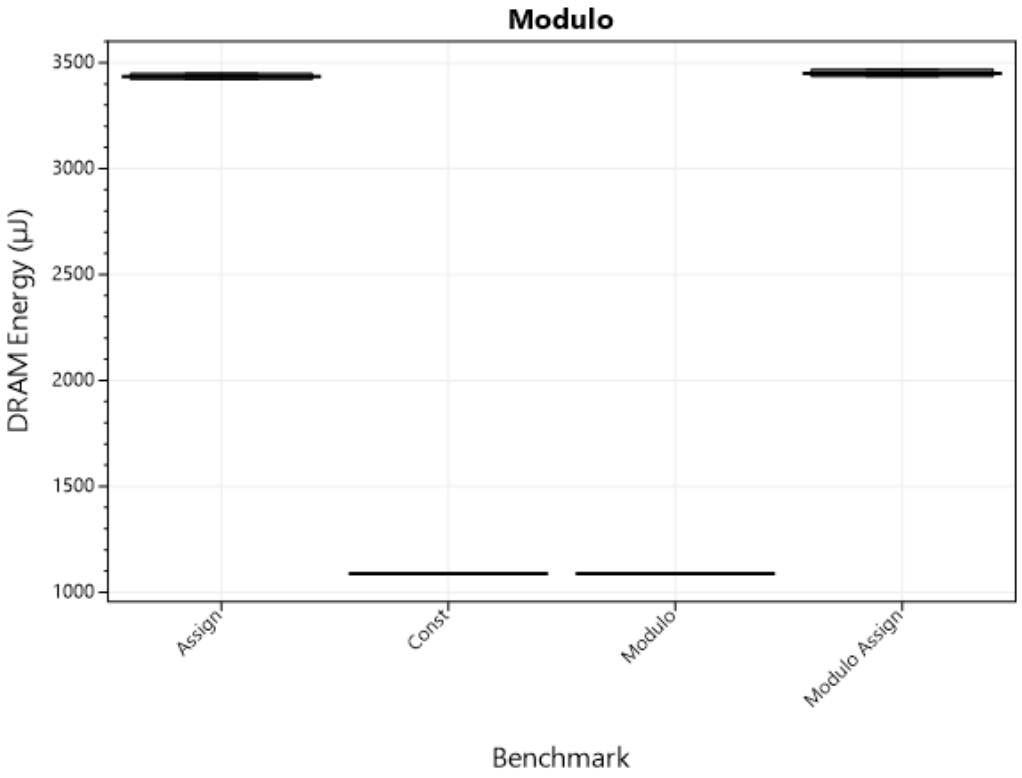

Figure A.28: Boxplot showing how efficient different Modulo types are with regards to Dram Energy.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Assign	6,179679	53.088,922	3.435,937
Const	1,957728	21.259,890	1.088,683
Modulo	1,957831	20.521,440	1.088,674
Modulo Assign	6,198073	53.787,849	3.448,196

Table A.29: Table showing the elapsed time and energy measurement for each Modulo.

Elapsed Time *p*-Values	Modulo	Const	Modulo Assign	Assign
Modulo	-	0,324	<0,05	<0,05
Const	0,324	-	<0,05	<0,05
Modulo Assign	<0,05	<0,05	-	<0,05
Assign	<0,05	<0,05	<0,05	-

Table A.30: Table showing the *p*-values for the group Modulo with regards to Elapsed Time.

Package Energy *p*-Values	Modulo	Const	Modulo Assign	Assign
Modulo	-	<0,05	<0,05	<0,05
Const	<0,05	-	<0,05	<0,05
Modulo Assign	<0,05	<0,05	-	<0,05
Assign	<0,05	<0,05	<0,05	-

Table A.31: Table showing the *p*-values for the group Modulo with regards to Package Energy.

DRAM Energy *p*-Values	Modulo	Const	Modulo Assign	Assign
Modulo	-	0,987	<0,05	<0,05
Const	0,987	-	<0,05	<0,05
Modulo Assign	<0,05	<0,05	-	<0,05
Assign	<0,05	<0,05	<0,05	-

Table A.32: Table showing the *p*-values for the group Modulo with regards to DRAM Energy.

Unary Operators

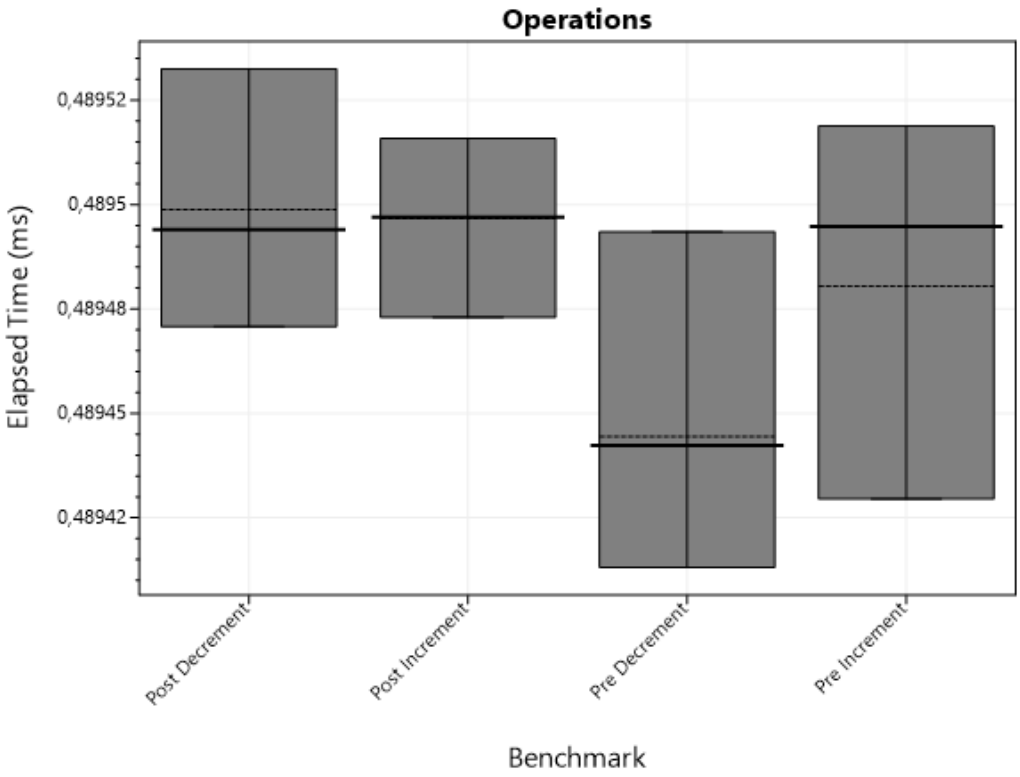

Figure A.29: Boxplot showing how efficient different Unary Operations types are with regards to Elapsed Time.

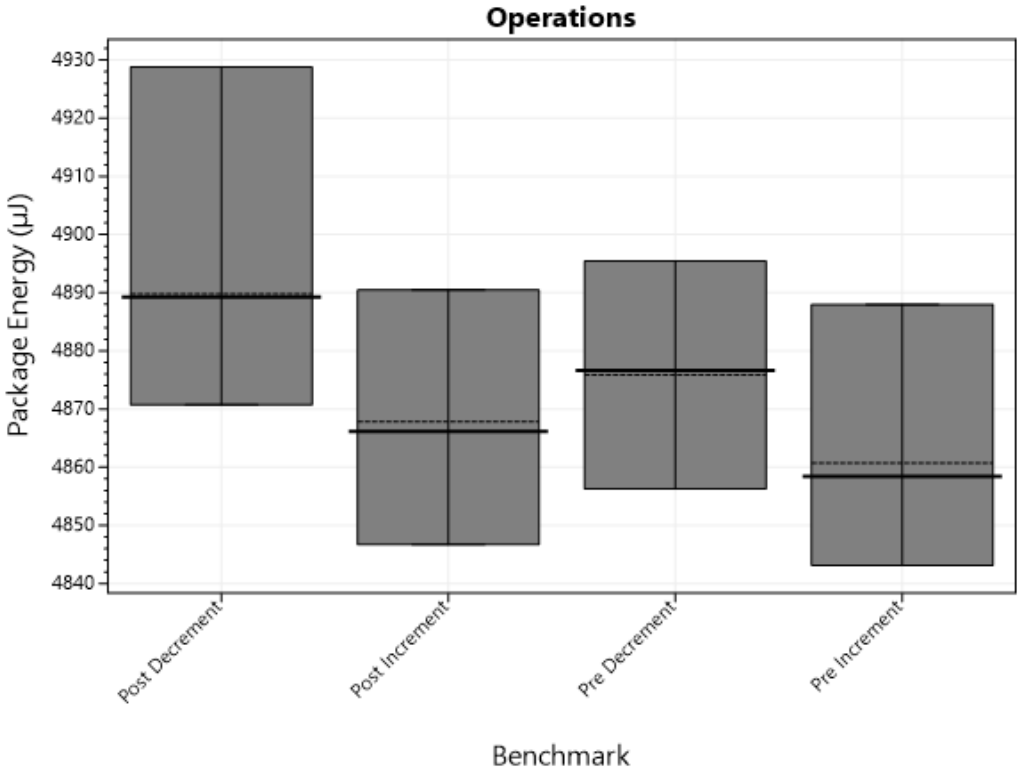

Figure A.30: Boxplot showing how efficient different Unary Operations types are with regards to Package Energy.

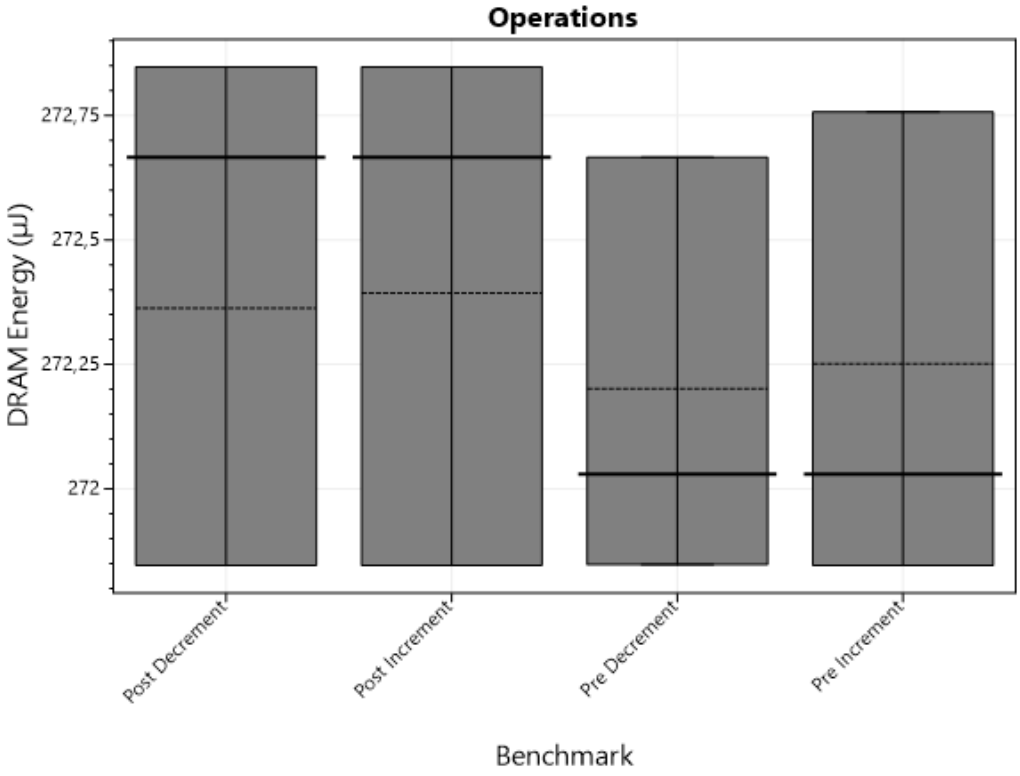

Figure A.31: Boxplot showing how efficient different Unary Operations types are with regards to Dram Energy.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Post Decrement	0,489499	4.889,805	272,363
Post Increment	0,489497	4.867,795	272,393
Pre Decrement	0,489444	4.875,879	272,201
Pre Increment	0,489480	4.860,691	272,252

Table A.33: Table showing the elapsed time and energy measurement for each Operations.

Elapsed Time *p*-Values	Post Increment	Post Decrement	Pre Increment	Pre Decrement
Post Increment	-	0,760	0,181	<0,05
Post Decrement	0,760	-	0,152	<0,05
Pre Increment	0,181	0,152	-	<0,05
Pre Decrement	<0,05	<0,05	<0,05	-

Table A.34: Table showing the *p*-values for the group Operations with regards to Elapsed Time.

Package Energy *p*-Values	Post Increment	Post Decrement	Pre Increment	Pre Decrement
Post Increment	-	<0,05	0,268	0,189
Post Decrement	<0,05	-	<0,05	0,070
Pre Increment	0,268	<0,05	-	<0,05
Pre Decrement	0,189	0,070	<0,05	-

Table A.35: Table showing the *p*-values for the group Operations with regards to Package Energy.

DRAM Energy *p*-Values	Post Increment	Post Decrement	Pre Increment	Pre Decrement
Post Increment	-	0,892	0,515	0,337
Post Decrement	0,892	-	0,613	0,425
Pre Increment	0,515	0,613	-	0,793
Pre Decrement	0,337	0,425	0,793	-

Table A.36: Table showing the *p*-values for the group Operations with regards to DRAM Energy.

Bitwise operators

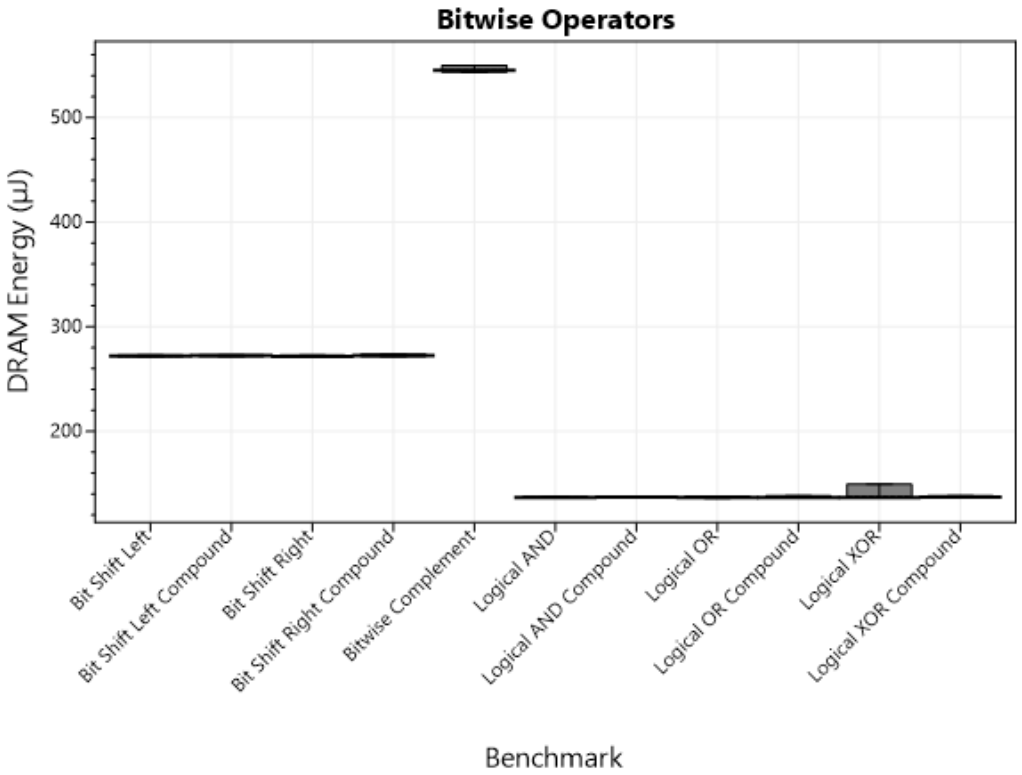

Figure A.32: Boxplot showing how efficient different Bitwise Operators types are with regards to DRAM Energy.

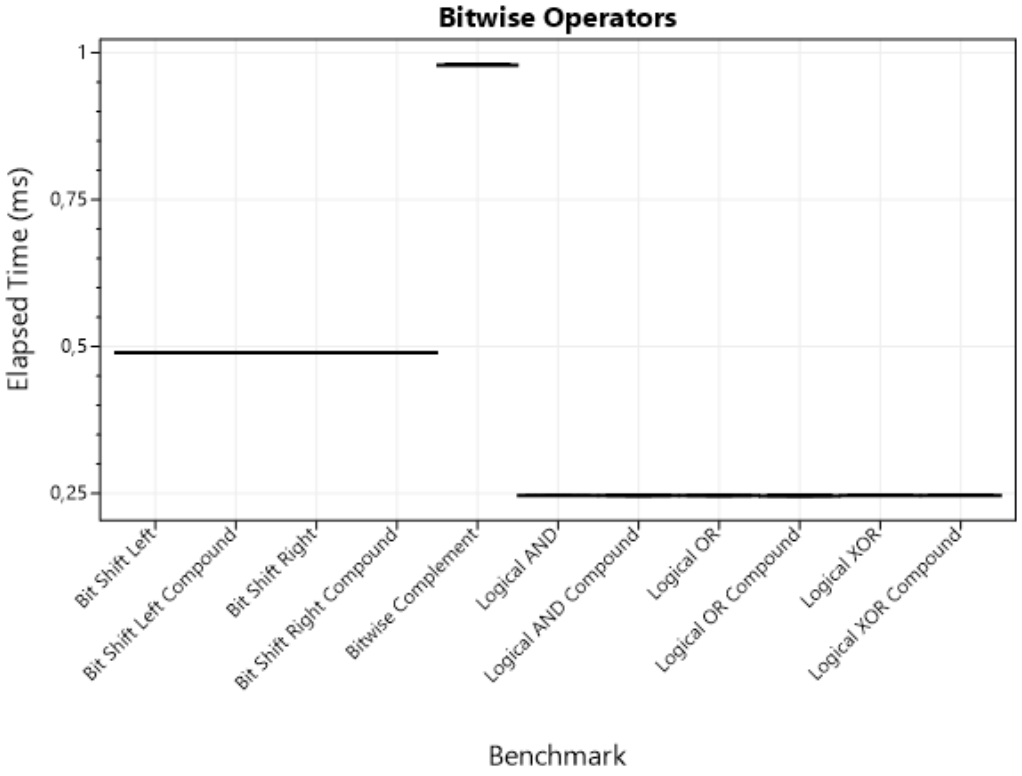

Figure A.33: Boxplot showing how efficient different Bitwise Operators types are with regards to Elapsed Time.

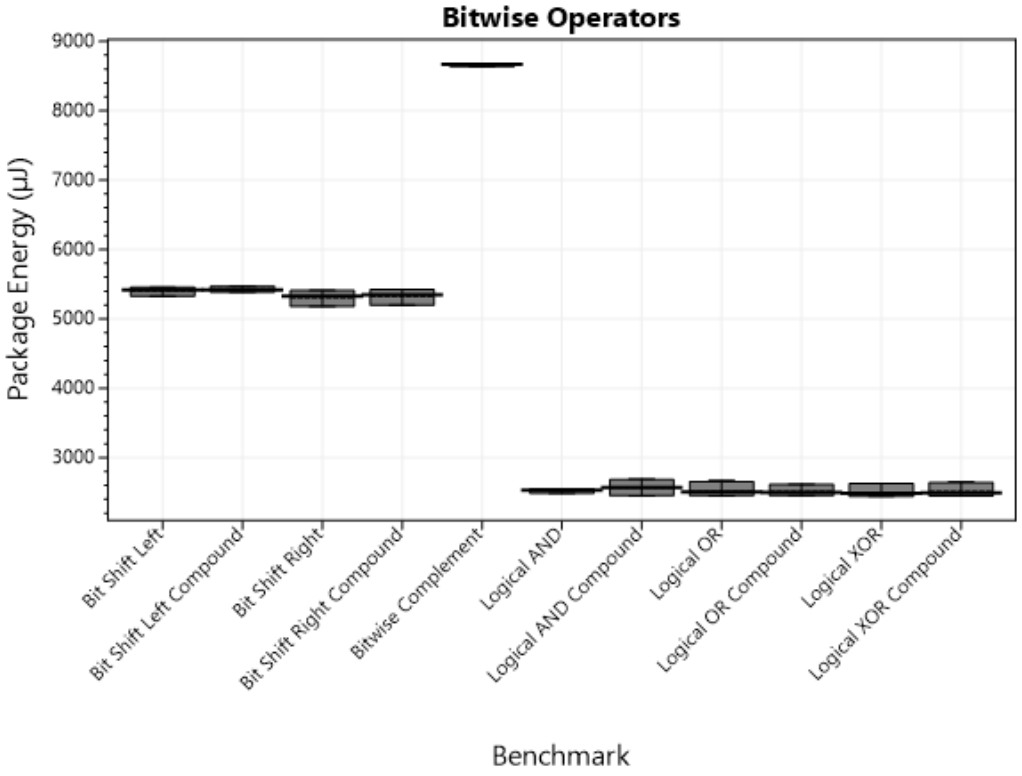

Figure A.34: Boxplot showing how efficient different Bitwise Operators types are with regards to Package Energy.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Bit Shift Left	0,489503	5.404,416	272,252
Bit Shift Left Compound	0,489495	5.426,810	272,353
Bit Shift Right	0,489454	5.305,456	272,084
Bit Shift Right Compound	0,489485	5.325,906	272,443
Bitwise Complement	0,979219	8.663,097	545,332
Logical AND	0,246354	2.525,528	136,933
Logical AND Compound	0,246497	2.571,386	137,070
Logical OR	0,246280	2.512,265	136,968
Logical OR Compound	0,246271	2.514,505	136,955
Logical XOR	0,246448	2.492,575	137,410
Logical XOR Compound	0,246563	2.514,283	137,146

Table A.37: Table showing the elapsed time and energy measurement for each Bitwise Operators.

Elapsed Time *p*-Values	Bit Shift Left	Bit Shift Left Compound	Bit Shift Right	Bit Shift Right Compound	Logical AND	Logical AND Compound	Logical OR	Logical OR Compound	Logical XOR	Logical XOR Compound	Bitwise Complement
Bit Shift Left	-	0,279	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Bit Shift Left Compound	0,279	-	<0,05	0,086	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Bit Shift Right	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Bit Shift Right Compound	<0,05	0,086	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Logical AND	<0,05	<0,05	<0,05	<0,05	-	0,571	0,755	0,795	0,752	0,411	<0,05
Logical AND Compound	<0,05	<0,05	<0,05	<0,05	0,571	-	0,074	0,140	0,760	0,601	<0,05
Logical OR	<0,05	<0,05	<0,05	<0,05	0,755	0,074	-	0,955	0,295	<0,05	<0,05
Logical OR Compound	<0,05	<0,05	<0,05	<0,05	0,795	0,140	0,955	-	0,392	0,081	<0,05
Logical XOR	<0,05	<0,05	<0,05	<0,05	0,752	0,760	0,295	0,392	-	0,499	<0,05
Logical XOR Compound	<0,05	<0,05	<0,05	<0,05	0,411	0,601	<0,05	0,081	0,499	-	<0,05
Bitwise Complement	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-

Table A.38: Table showing the *p*-values for the group Bitwise Operators with regards to Elapsed Time.

Package Energy *p*-Values	Bit Shift Left	Bit Shift Left Compound	Bit Shift Right	Bit Shift Right Compound	Logical AND	Logical AND Compound	Logical OR	Logical OR Compound	Logical XOR	Logical XOR Compound	Bitwise Complement
Bit Shift Left	-	0,205	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Bit Shift Left Compound	0,205	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Bit Shift Right	<0,05	<0,05	-	0,297	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Bit Shift Right Compound	<0,05	<0,05	0,297	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Logical AND	<0,05	<0,05	<0,05	<0,05	-	<0,05	0,397	0,442	<0,05	0,494	<0,05
Logical AND Compound	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Logical OR	<0,05	<0,05	<0,05	<0,05	0,397	<0,05	-	0,802	<0,05	0,818	<0,05
Logical OR Compound	<0,05	<0,05	<0,05	<0,05	0,442	<0,05	0,802	-	<0,05	0,981	<0,05
Logical XOR	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Logical XOR Compound	<0,05	<0,05	<0,05	<0,05	0,494	<0,05	0,818	0,981	<0,05	-	<0,05
Bitwise Complement	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-

Table A.39: Table showing the *p*-values for the group Bitwise Operators with regards to Package Energy.

DRAM Energy *p*-Values	Bit Shift Left	Bit Shift Left Compound	Bit Shift Right	Bit Shift Right Compound	Logical AND	Logical AND Compound	Logical OR	Logical OR Compound	Logical XOR	Logical XOR Compound	Bitwise Complement
Bit Shift Left	-	0,632	0,319	0,394	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Bit Shift Left Compound	0,632	-	0,104	0,678	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Bit Shift Right	0,319	0,104	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Bit Shift Right Compound	0,394	0,678	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Logical AND	<0,05	<0,05	<0,05	<0,05	-	0,360	0,816	0,911	0,489	0,186	<0,05
Logical AND Compound	<0,05	<0,05	<0,05	<0,05	0,360	-	0,167	0,204	0,155	0,325	<0,05
Logical OR	<0,05	<0,05	<0,05	<0,05	0,816	0,167	-	0,890	0,108	<0,05	<0,05
Logical OR Compound	<0,05	<0,05	<0,05	<0,05	0,911	0,204	0,890	-	0,170	0,063	<0,05
Logical XOR	<0,05	<0,05	<0,05	<0,05	0,489	0,155	0,108	0,170	-	0,352	<0,05
Logical XOR Compound	<0,05	<0,05	<0,05	<0,05	0,186	0,325	<0,05	0,063	0,352	-	<0,05
Bitwise Complement	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-

Table A.40: Table showing the *p*-values for the group Bitwise Operators with regards to DRAM Energy.

Conditional Operators

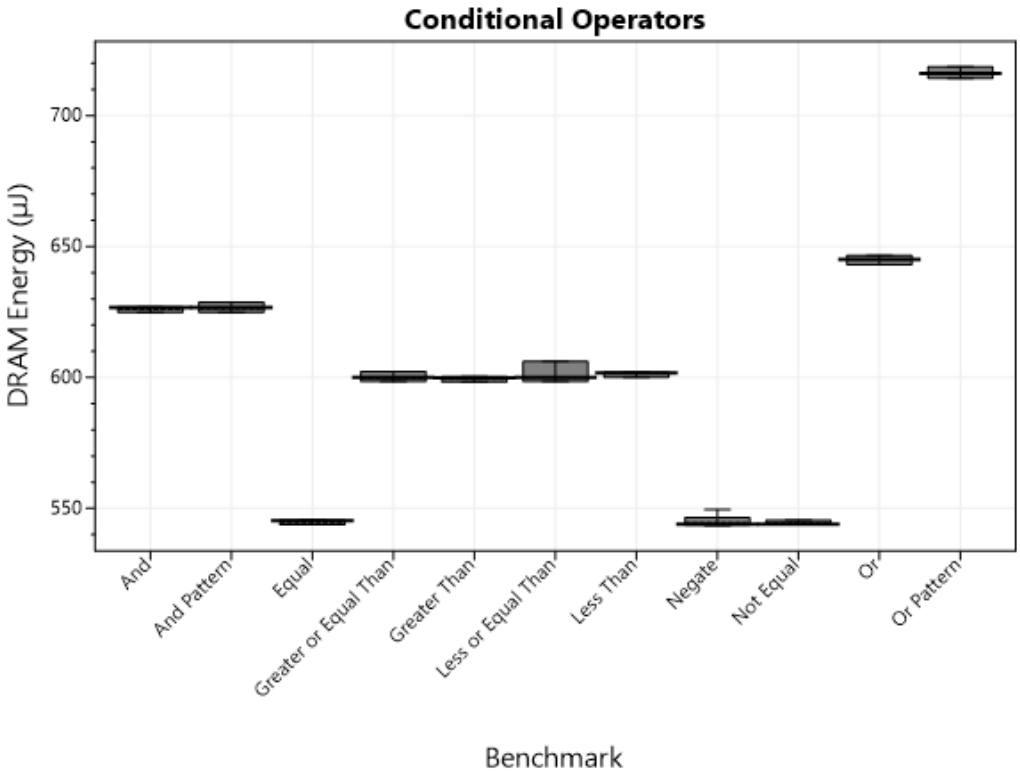

Figure A.35: Boxplot showing how efficient different Conditional Operators types are with regards to DRAM Energy.

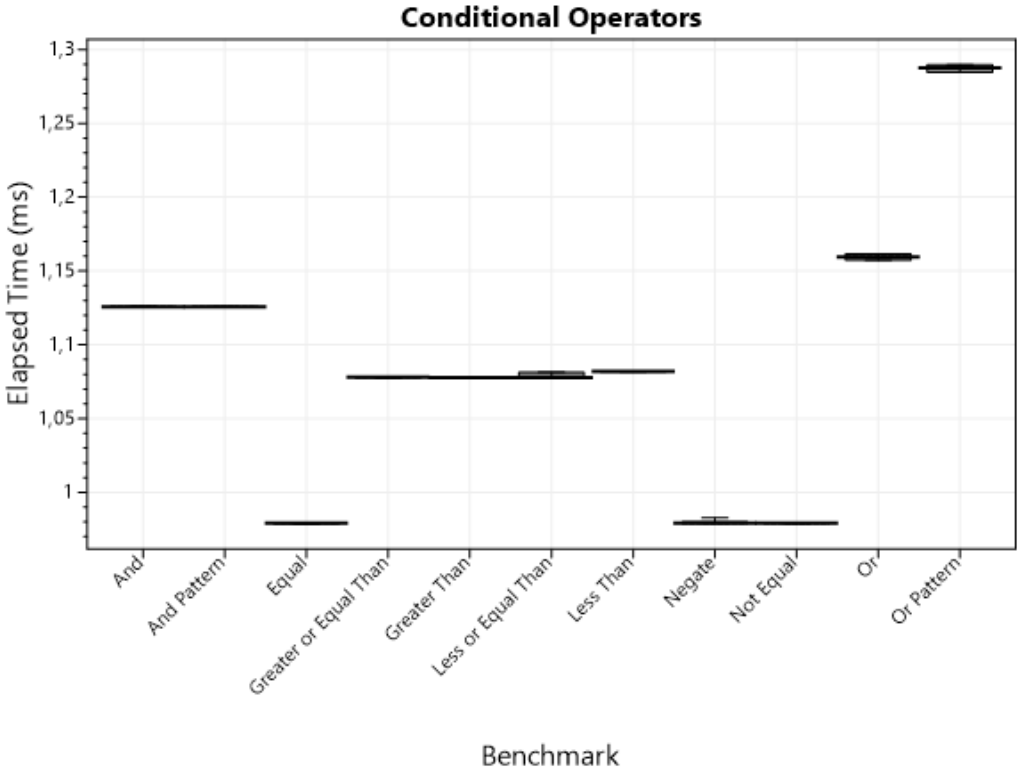

Figure A.36: Boxplot showing how efficient different Conditional Operators types are with regards to Elapsed Time.

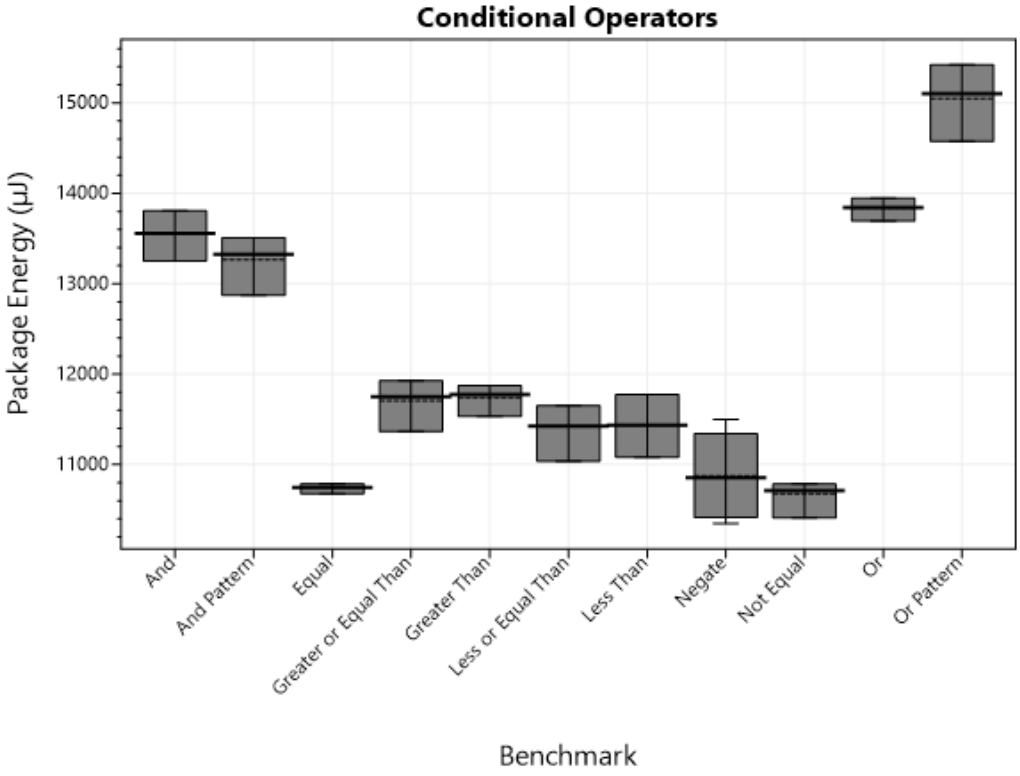

Figure A.37: Boxplot showing how efficient different Conditional Operators types are with regards to Package Energy.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
And	1,125594	13.552,047	626,095
And Pattern	1,125644	13.267,380	626,182
Equal	0,979089	10.749,654	544,705
Greater or Equal Than	1,077929	11.707,635	599,607
Greater Than	1,077843	11.742,556	599,372
Less or Equal Than	1,077949	11.421,332	599,992
Less Than	1,081881	11.441,822	601,726
Negate	0,979230	10.875,233	544,601
Not Equal	0,979070	10.675,230	544,408
Or	1,159638	13.835,520	645,054
Or Pattern	1,287214	15.046,085	715,958

Table A.41: Table showing the elapsed time and energy measurement for each Conditional Operators.

Elapsed Time *p*-Values	Equal	Not Equal	Greater Than	Less Than	Greater or Equal Than	Less or Equal Than	Or	Or Pattern	And	And Pattern	Negate
Equal	-	0,246	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	0,321
Not Equal	0,246	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	0,175
Greater Than	<0,05	<0,05	-	<0,05	<0,05	0,588	<0,05	<0,05	<0,05	<0,05	<0,05
Less Than	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Greater or Equal Than	<0,05	<0,05	<0,05	<0,05	-	0,886	<0,05	<0,05	<0,05	<0,05	<0,05
Less or Equal Than	<0,05	<0,05	0,588	<0,05	0,886	-	<0,05	<0,05	<0,05	<0,05	<0,05
Or	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Or Pattern	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05
And	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
And Pattern	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05
Negate	0,321	0,175	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-

Table A.42: Table showing the *p*-values for the group Conditional Operators with regards to Elapsed Time.

Package Energy *p*-Values	Equal	Not Equal	Greater Than	Less Than	Greater or Equal Than	Less or Equal Than	Or	Or Pattern	And	And Pattern	Negate
Equal	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	0,114
Not Equal	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Greater Than	<0,05	<0,05	-	<0,05	0,445	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Less Than	<0,05	<0,05	<0,05	-	<0,05	0,625	<0,05	<0,05	<0,05	<0,05	<0,05
Greater or Equal Than	<0,05	<0,05	0,445	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Less or Equal Than	<0,05	<0,05	<0,05	0,625	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Or	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Or Pattern	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05
And	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
And Pattern	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05
Negate	0,114	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-

Table A.43: Table showing the *p*-values for the group Conditional Operators with regards to Package Energy.

DRAM Energy *p*-Values	Equal	Not Equal	Greater Than	Less Than	Greater or Equal Than	Less or Equal Than	Or	Or Pattern	And	And Pattern	Negate
Equal	-	0,448	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	0,769
Not Equal	0,448	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	0,520
Greater Than	<0,05	<0,05	-	<0,05	0,519	0,219	<0,05	<0,05	<0,05	<0,05	<0,05
Less Than	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Greater or Equal Than	<0,05	<0,05	0,519	<0,05	-	0,319	<0,05	<0,05	<0,05	<0,05	<0,05
Less or Equal Than	<0,05	<0,05	0,219	<0,05	0,319	-	<0,05	<0,05	<0,05	<0,05	<0,05
Or	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Or Pattern	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05
And	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	0,787	<0,05
And Pattern	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	0,787	-	<0,05
Negate	0,769	0,520	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-

Table A.44: Table showing the *p*-values for the group Conditional Operators with regards to DRAM Energy.

A.3.3 Variables

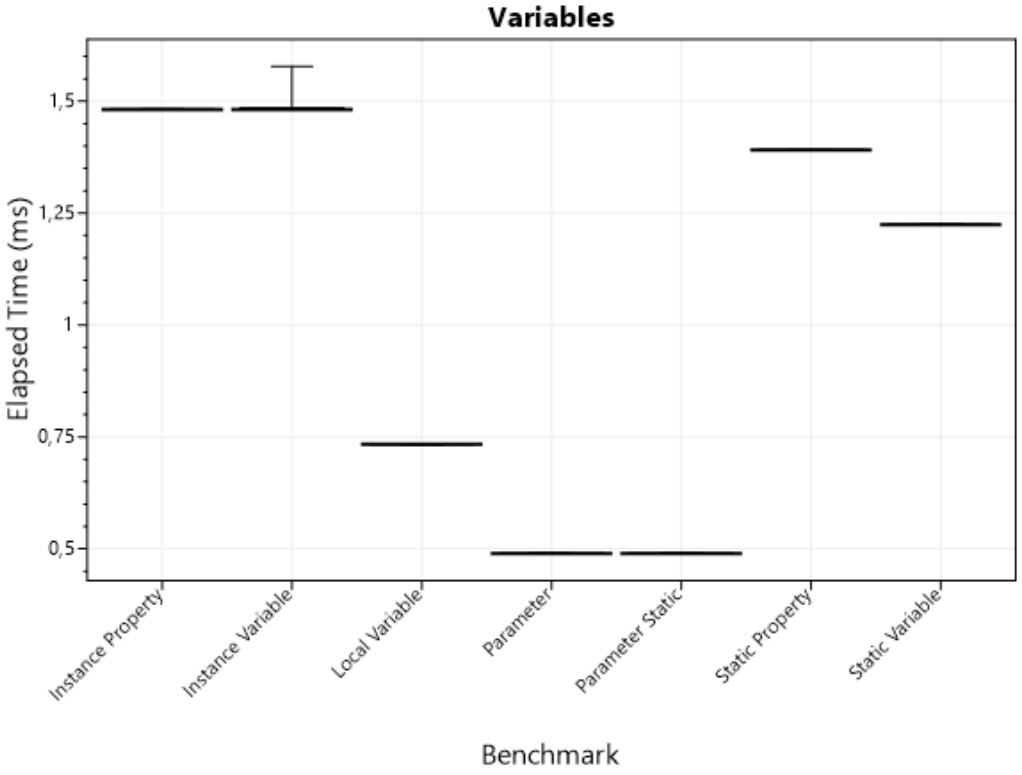

Figure A.38: Boxplot showing how efficient different Variables types are with regards to Elapsed Time.

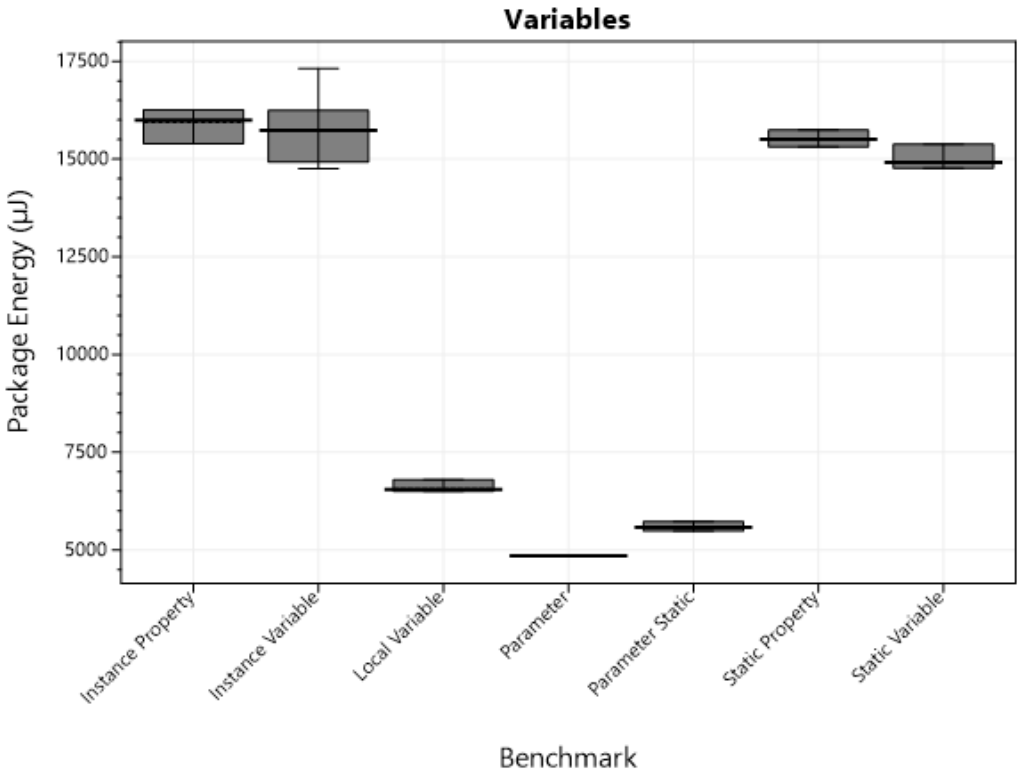

Figure A.39: Boxplot showing how efficient different Variables types are with regards to Package Energy.

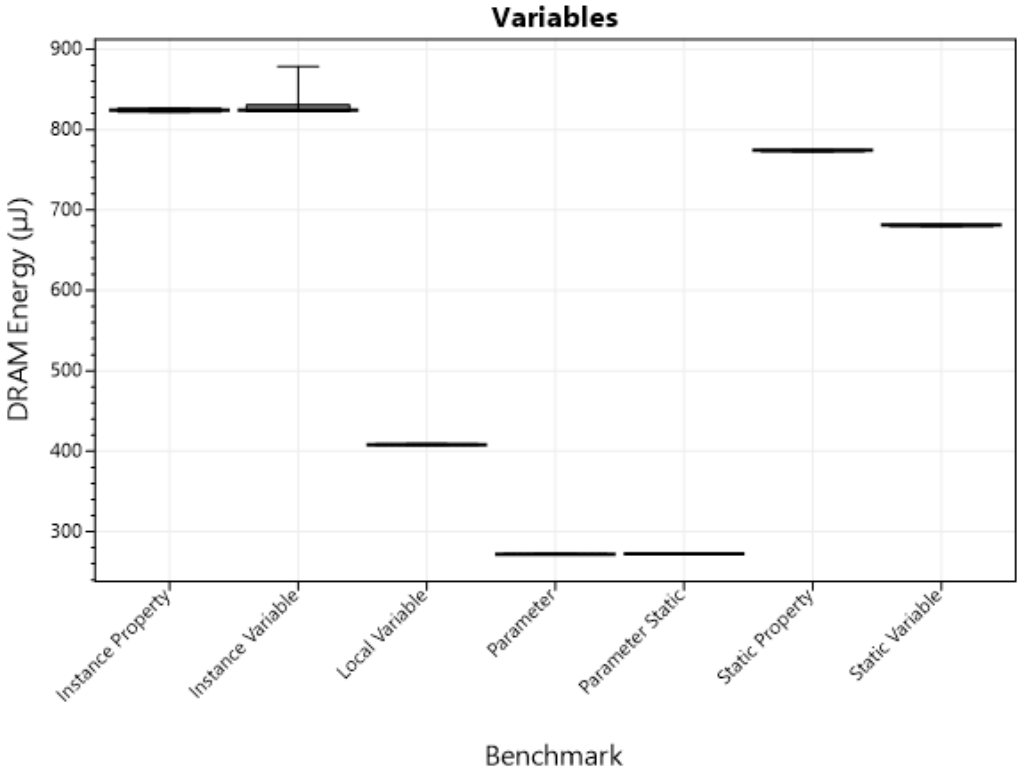

Figure A.40: Boxplot showing how efficient different Variables types are with regards to Dram Energy.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Instance Property	1,481333	15.932,726	823,824
Instance Variable	1,482308	15.719,031	824,549
Local Variable	0,734008	6.569,280	408,226
Parameter	0,489496	4.851,040	272,080
Parameter Static	0,489482	5.590,717	272,338
Static Property	1,390867	15.521,078	773,538
Static Variable	1,224481	14.923,368	680,867

Table A.45: Table showing the elapsed time and energy measurement for each Variables.

Elapsed Time *p*-Values	Local Variable	Static Property	Instance Property	Static Variable	Instance Variable	Parameter	Parameter Static
Local Variable	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Static Property	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Instance Property	<0,05	<0,05	-	<0,05	0,568	<0,05	<0,05
Static Variable	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05
Instance Variable	<0,05	<0,05	0,568	<0,05	-	<0,05	<0,05
Parameter	<0,05	<0,05	<0,05	<0,05	<0,05	-	0,063
Parameter Static	<0,05	<0,05	<0,05	<0,05	<0,05	0,063	-

Table A.46: Table showing the *p*-values for the group Variables with regards to Elapsed Time.

Package Energy *p*-Values	Local Variable	Static Property	Instance Property	Static Variable	Instance Variable	Parameter	Parameter Static
Local Variable	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Static Property	<0,05	-	<0,05	<0,05	0,099	<0,05	<0,05
Instance Property	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Static Variable	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05
Instance Variable	<0,05	0,099	<0,05	<0,05	-	<0,05	<0,05
Parameter	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05
Parameter Static	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-

Table A.47: Table showing the *p*-values for the group Variables with regards to Package Energy.

DRAM Energy *p*-Values	Local Variable	Static Property	Instance Property	Static Variable	Instance Variable	Parameter	Parameter Static
Local Variable	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Static Property	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Instance Property	<0,05	<0,05	-	<0,05	0,466	<0,05	<0,05
Static Variable	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05
Instance Variable	<0,05	<0,05	0,466	<0,05	-	<0,05	<0,05
Parameter	<0,05	<0,05	<0,05	<0,05	<0,05	-	0,118
Parameter Static	<0,05	<0,05	<0,05	<0,05	<0,05	0,118	-

Table A.48: Table showing the *p*-values for the group Variables with regards to DRAM Energy.

A.3.4 Selection

If Statement

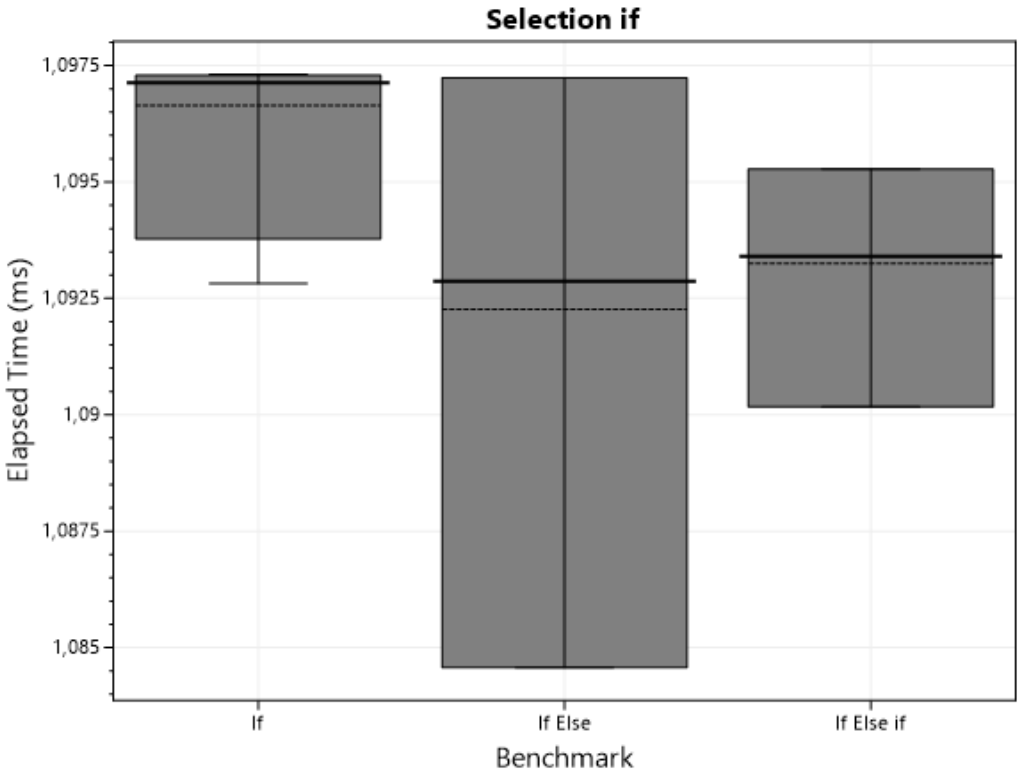

Figure A.41: Boxplot showing how efficient different Selection if types are with regards to Elapsed Time.

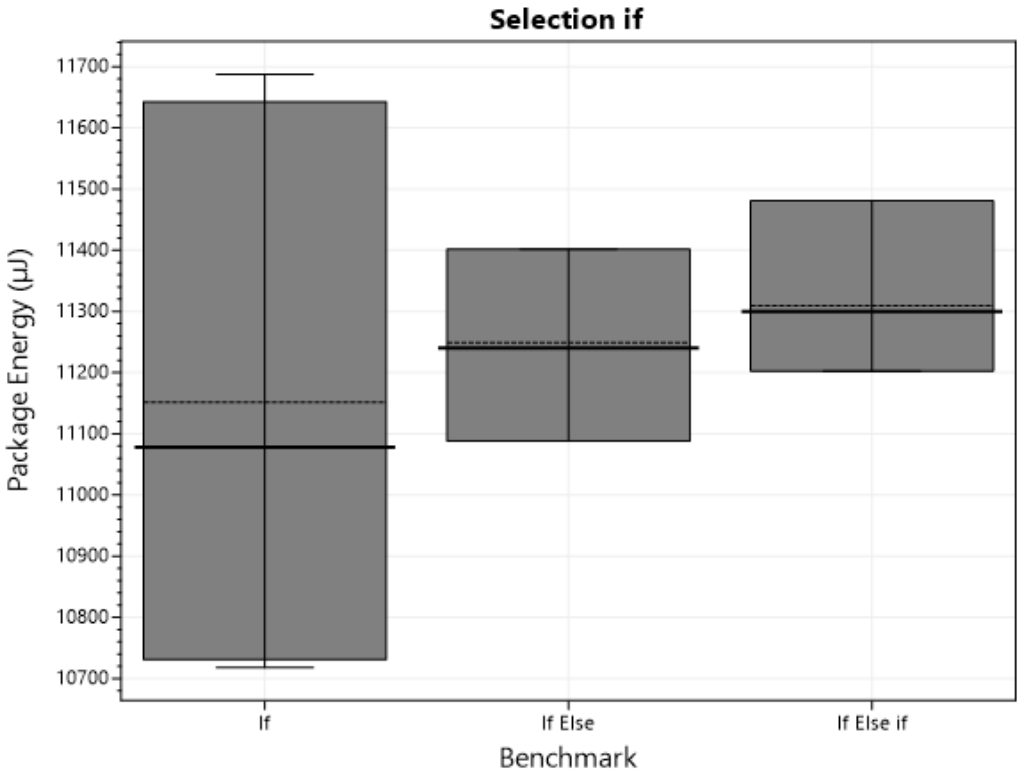

Figure A.42: Boxplot showing how efficient different Selection if types are with regards to Package Energy.

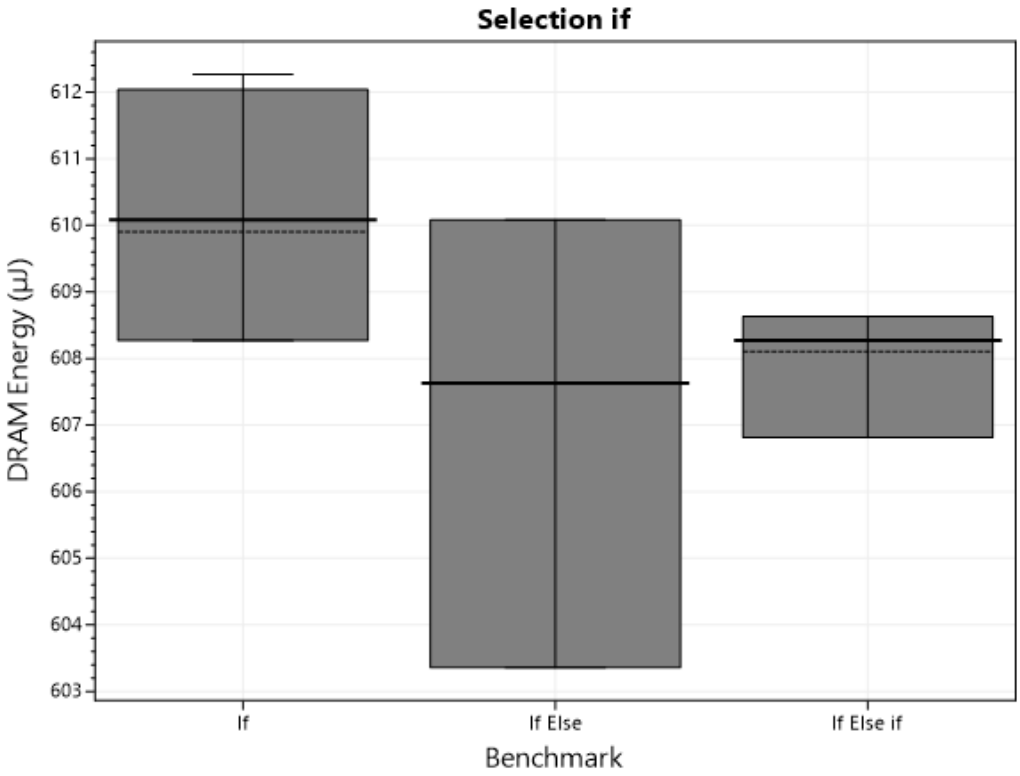

Figure A.43: Boxplot showing how efficient different Selection if types are with regards to DRAM Energy.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
If	1,096640	11.151,124	609,902
If Else	1,092265	11.248,654	607,645
If Else if	1,093252	11.308,931	608,107

Table A.49: Table showing the elapsed time and energy measurement for each Selection if.

Elapsed Time *p*-Values	If	If Else	If Else if
If	-	<0,05	<0,05
If Else	<0,05	-	0,527
If Else if	<0,05	0,527	-

Table A.50: Table showing the *p*-values for the group Selection if with regards to Elapsed Time.

Package Energy *p*-Values	If	If Else	If Else if
If	-	0,098	<0,05
If Else	0,098	-	0,143
If Else if	<0,05	0,143	-

Table A.51: Table showing the *p*-values for the group Selection if with regards to Package Energy.

DRAM Energy *p*-Values	If	If Else	If Else if
If	-	<0,05	<0,05
If Else	<0,05	-	0,559
If Else if	<0,05	0,559	-

Table A.52: Table showing the *p*-values for the group Selection if with regards to DRAM Energy.

Conditional

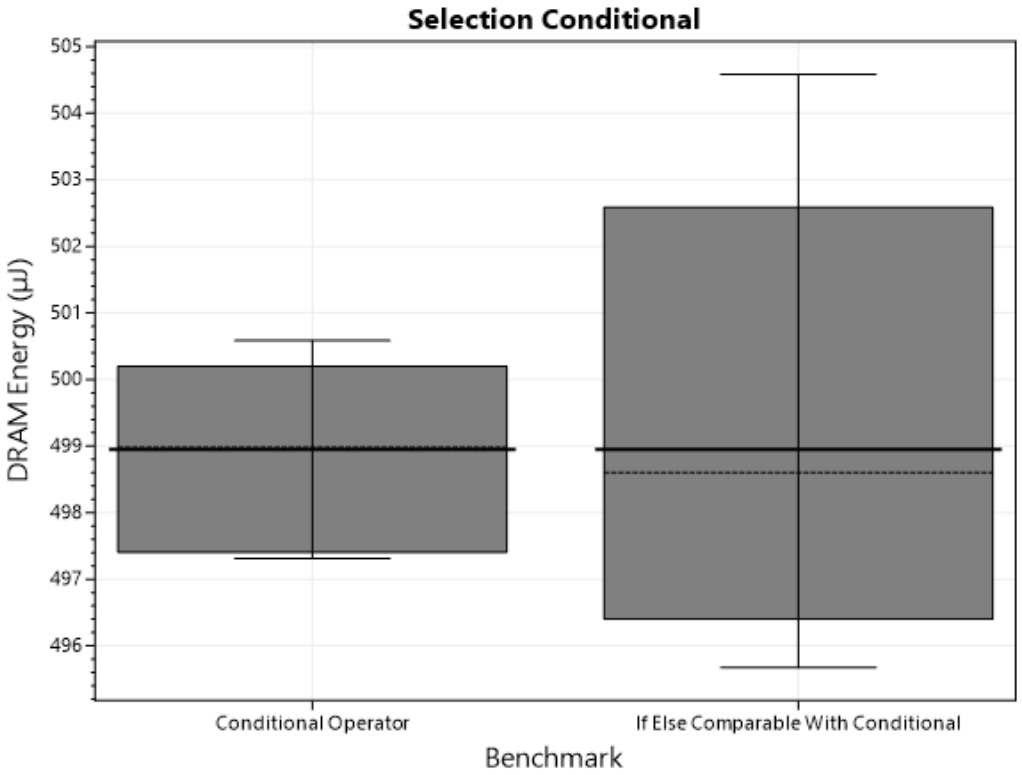

Figure A.44: Boxplot showing how efficient different Selection Conditional types are with regards to DRAM Energy.

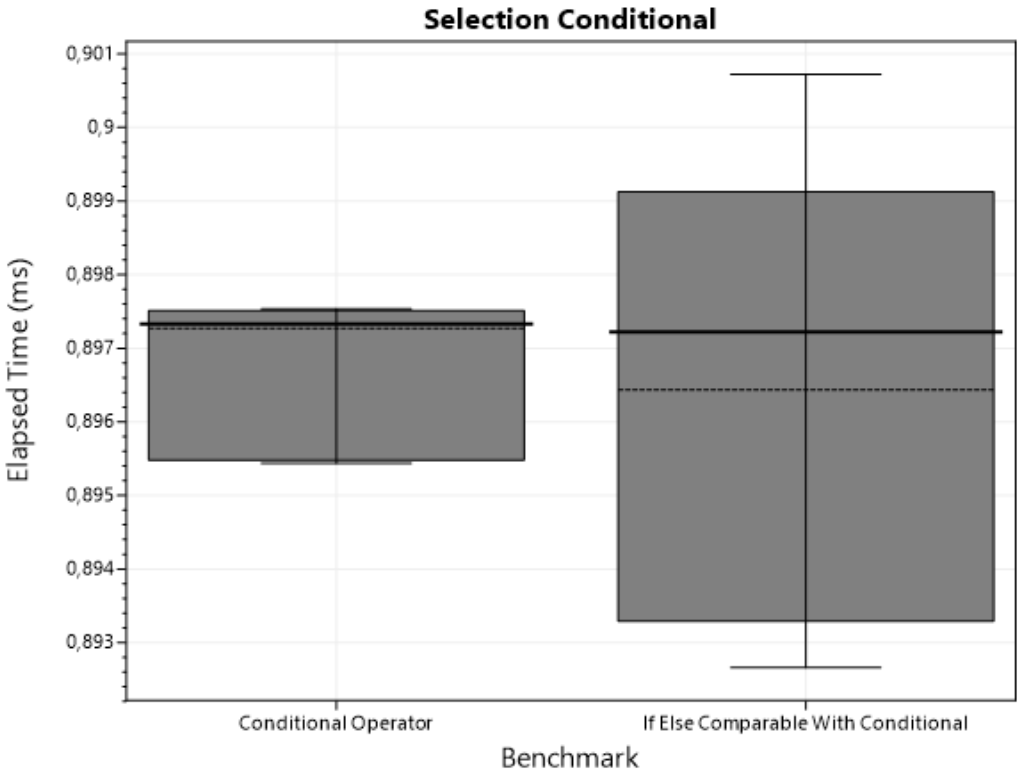

Figure A.45: Boxplot showing how efficient different Selection Conditional types are with regards to Elapsed Time.

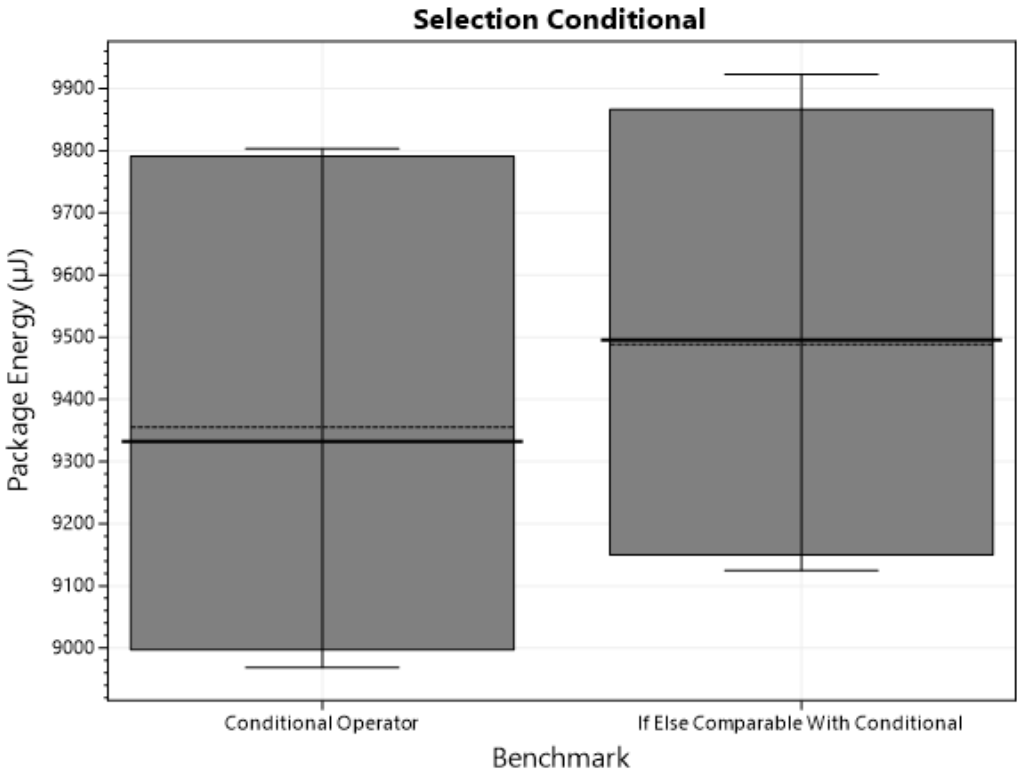

Figure A.46: Boxplot showing how efficient different Selection Conditional types are with regards to Package Energy.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Conditional Operator	0,897267	9.355,048	498,984
If Else Comparable With Conditional	0,896432	9.488,077	498,599

Table A.53: Table showing the elapsed time and energy measurement for each Selection Conditional.

Elapsed Time *p*-Values	If Else Comparable With Conditional	Conditional Operator
If Else Comparable With Conditional	-	<0,05
Conditional Operator	<0,05	-

Table A.54: Table showing the *p*-values for the group Selection Conditional with regards to Elapsed Time.

Package Energy *p*-Values	If Else Comparable With Conditional	Conditional Operator
If Else Comparable With Conditional	-	<0,05
Conditional Operator	<0,05	-

Table A.55: Table showing the *p*-values for the group Selection Conditional with regards to Package Energy.

DRAM Energy *p*-Values	If Else Comparable With Conditional	Conditional Operator
If Else Comparable With Conditional	-	<0,05
Conditional Operator	<0,05	-

Table A.56: Table showing the *p*-values for the group Selection Conditional with regards to DRAM Energy.

Const

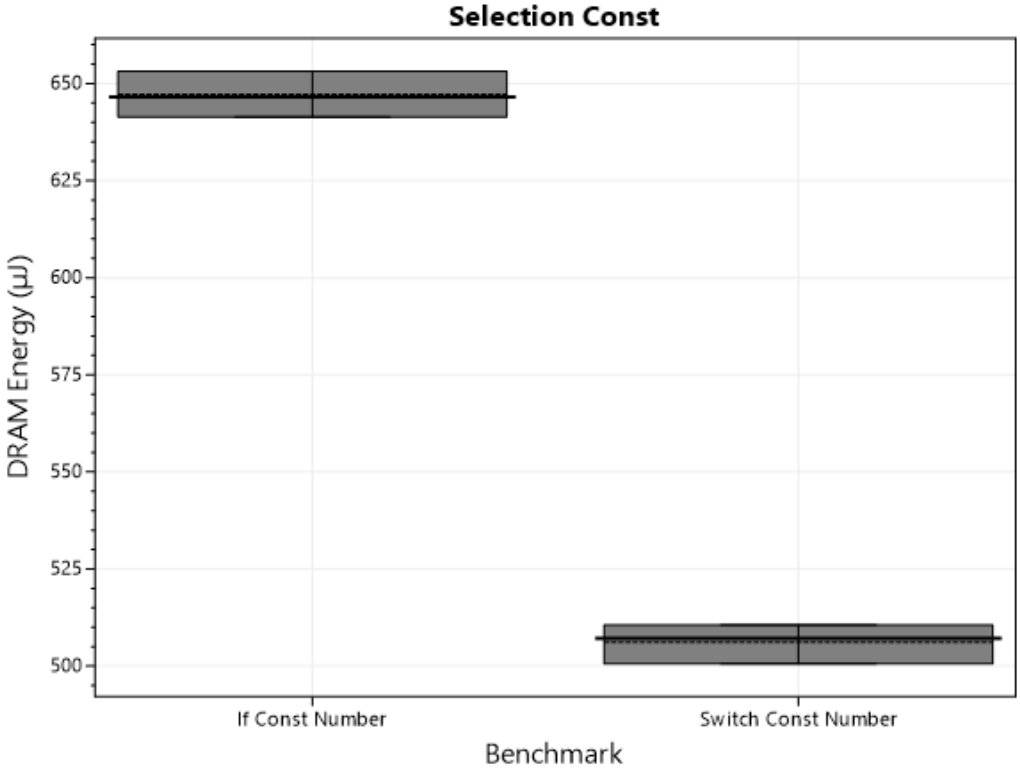

Figure A.47: Boxplot showing how efficient different Selection Const types are with regards to DRAM Energy.

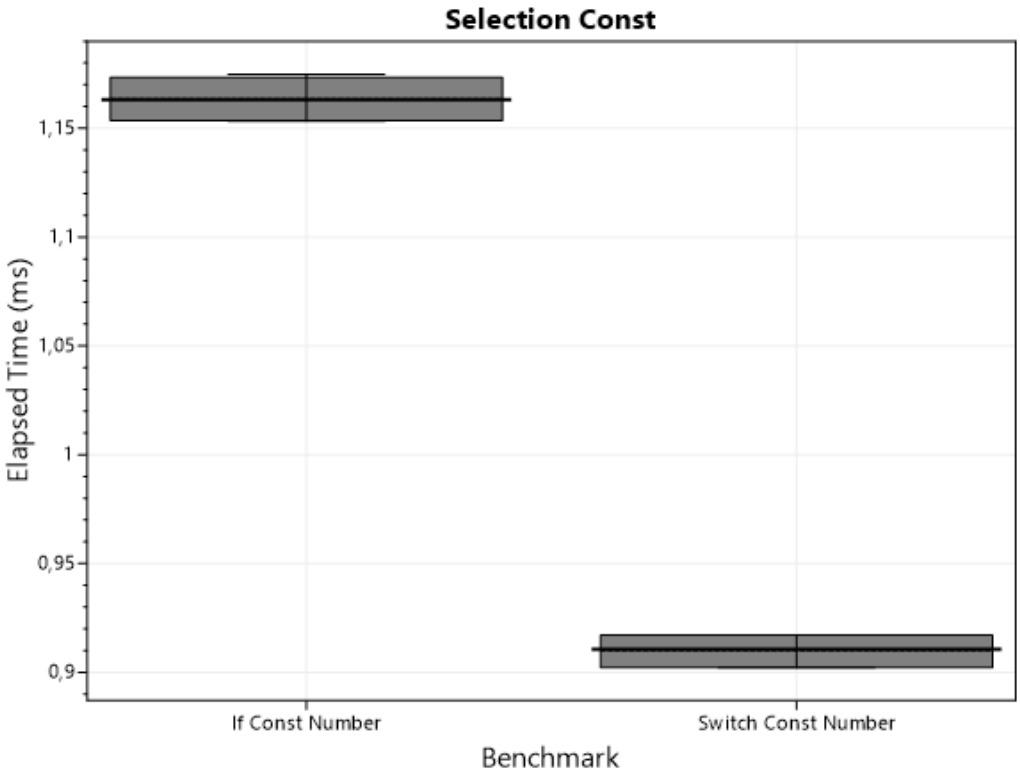

Figure A.48: Boxplot showing how efficient different Selection Const types are with regards to Elapsed Time.

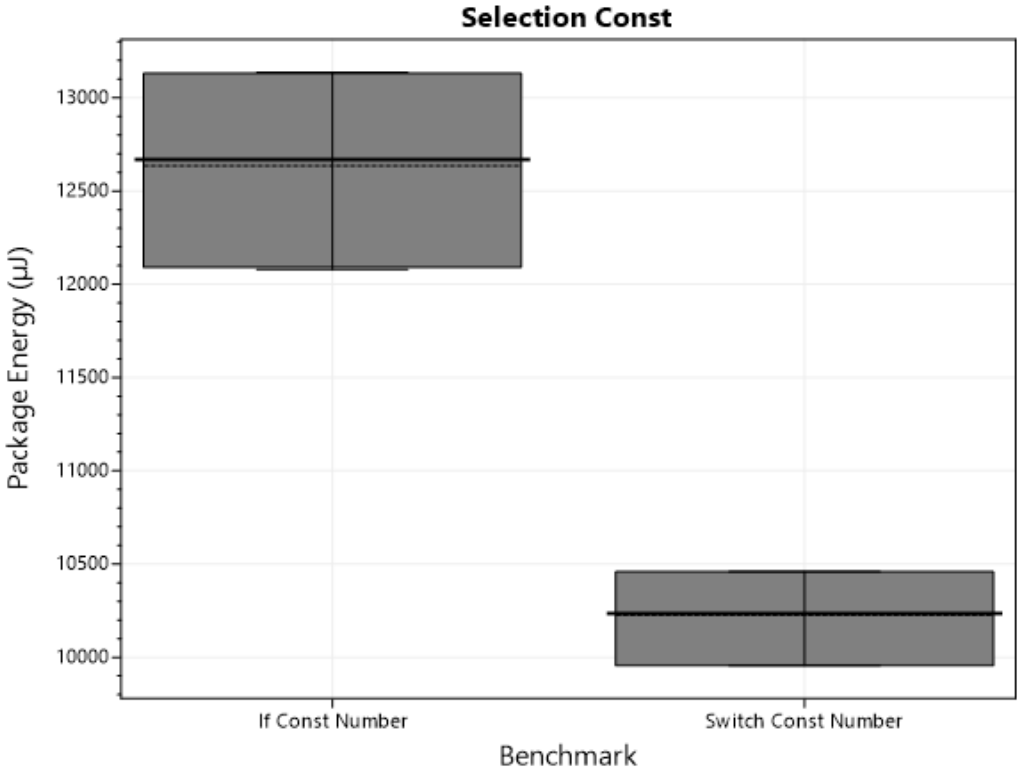

Figure A.49: Boxplot showing how efficient different Selection Const types are with regards to Package Energy.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
If Const Number	1,163601	12.636,177	647,165
Switch Const Number	0,909927	10.226,356	506,081

Table A.57: Table showing the elapsed time and energy measurement for each Selection Const.

Elapsed Time p-Values	Switch Const Number	If Const Number
Switch Const Number	-	<0,05
If Const Number	<0,05	-

Table A.58: Table showing the p-values for the group Selection Const with regards to Elapsed Time.

Package Energy *p*-Values	Switch Const Number	If Const Number
Switch Const Number	-	<0,05
If Const Number	<0,05	-

Table A.59: Table showing the *p*-values for the group Selection Const with regards to Package Energy.

DRAM Energy *p*-Values	Switch Const Number	If Const Number
Switch Const Number	-	<0,05
If Const Number	<0,05	-

Table A.60: Table showing the *p*-values for the group Selection Const with regards to DRAM Energy.

Switch

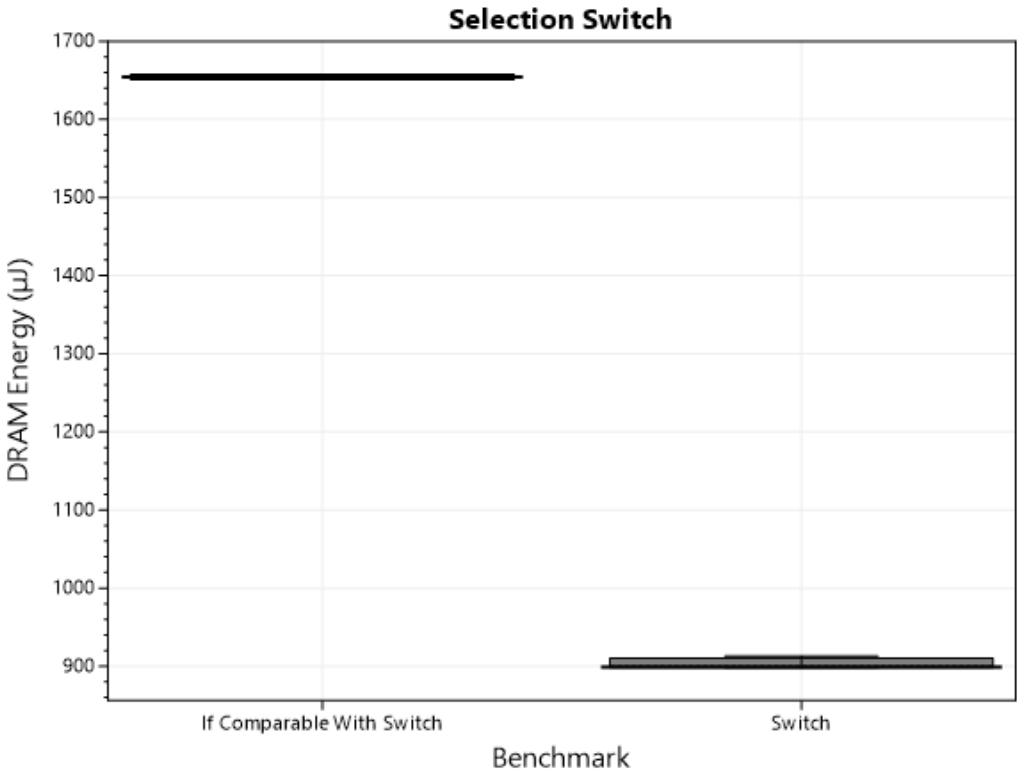

Figure A.50: Boxplot showing how efficient different Selection Switch types are with regards to DRAM Energy.

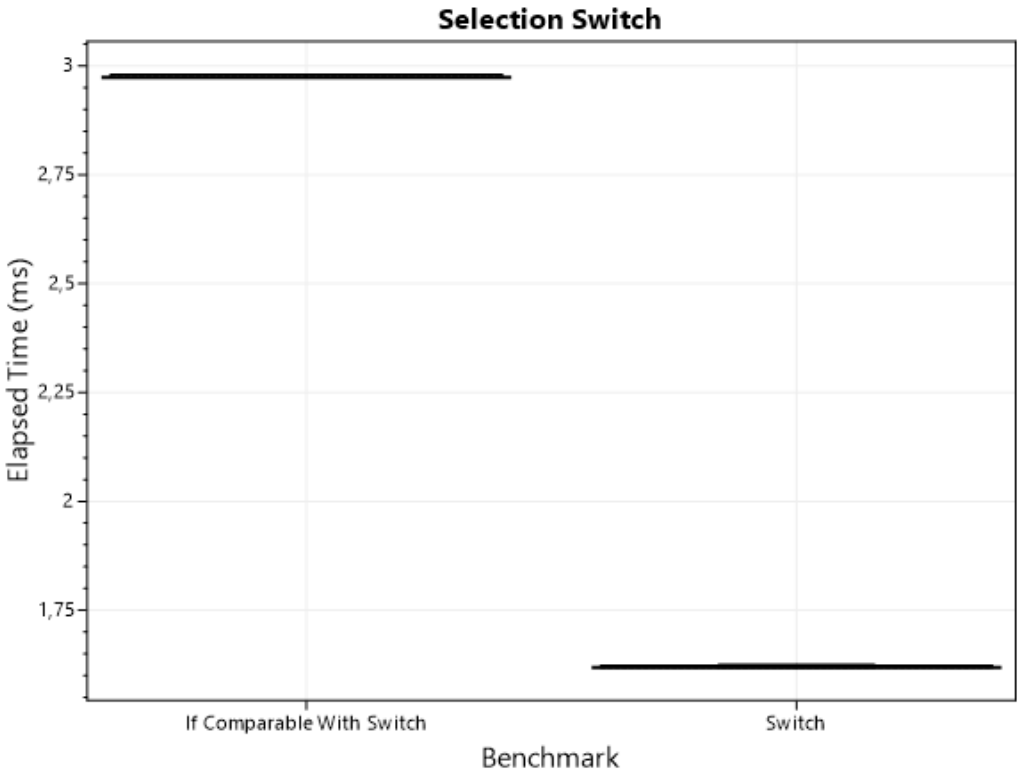

Figure A.51: Boxplot showing how efficient different Selection Switch types are with regards to Elapsed Time.

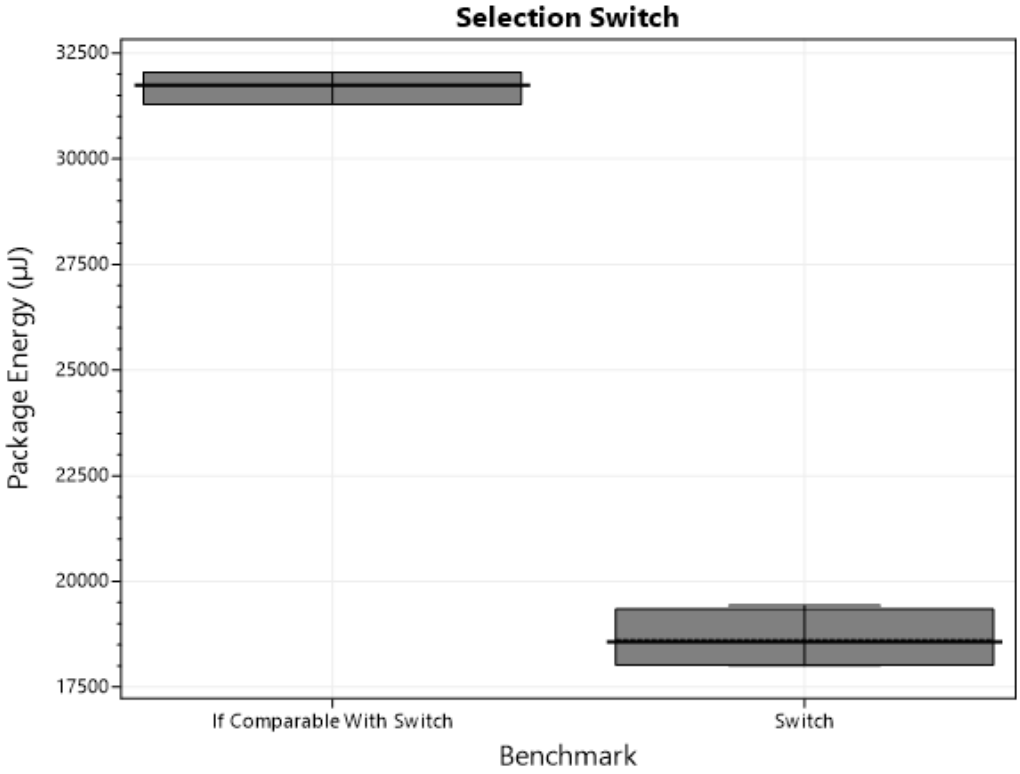

Figure A.52: Boxplot showing how efficient different Selection Switch types are with regards to Package Energy.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
If Comparable With Switch	2,973944	31.751,188	1.653,296
Switch	1,618432	18.620,357	900,318

Table A.61: Table showing the elapsed time and energy measurement for each Selection Switch.

Elapsed Time p-Values	If Comparable With Switch	Switch
If Comparable With Switch	-	<0,05
Switch	<0,05	-

Table A.62: Table showing the p-values for the group Selection Switch with regards to Elapsed Time.

Package Energy *p*-Values	If Comparable With Switch	Switch
If Comparable With Switch	-	<0,05
Switch	<0,05	-

Table A.63: Table showing the *p*-values for the group Selection Switch with regards to Package Energy.

DRAM Energy *p*-Values	If Comparable With Switch	Switch
If Comparable With Switch	-	<0,05
Switch	<0,05	-

Table A.64: Table showing the *p*-values for the group Selection Switch with regards to DRAM Energy.

A.3.5 Loops

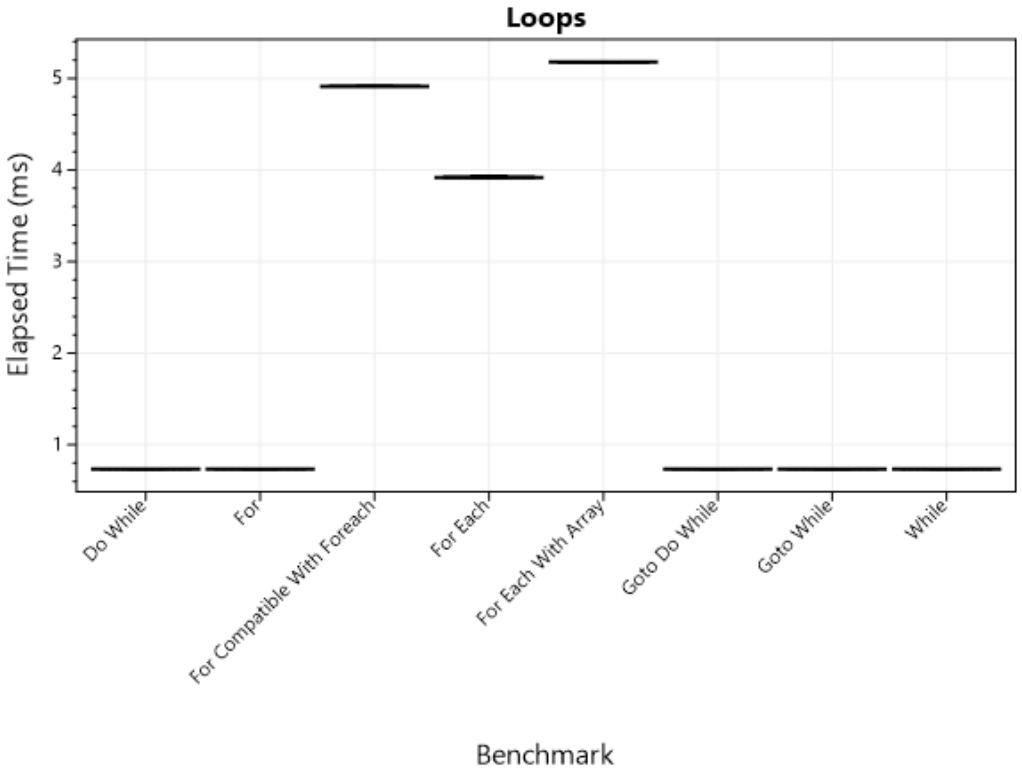

Figure A.53: Boxplot showing how efficient different Loops types are with regards to Elapsed Time.

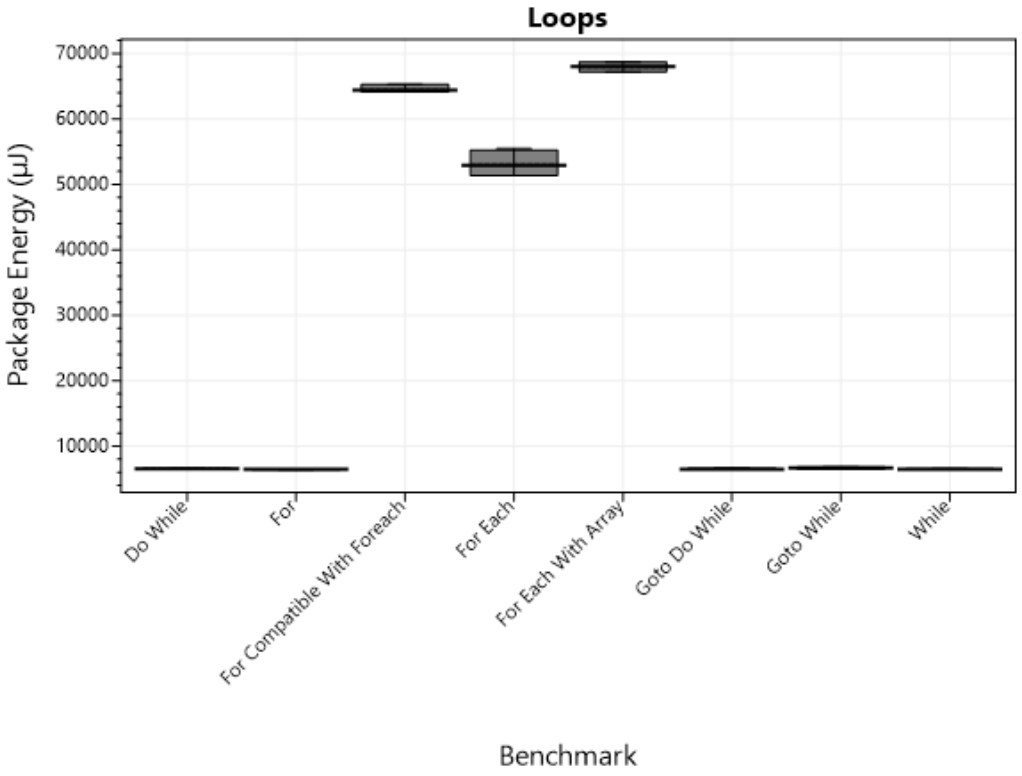

Figure A.54: Boxplot showing how efficient different Loops types are with regards to Package Energy.

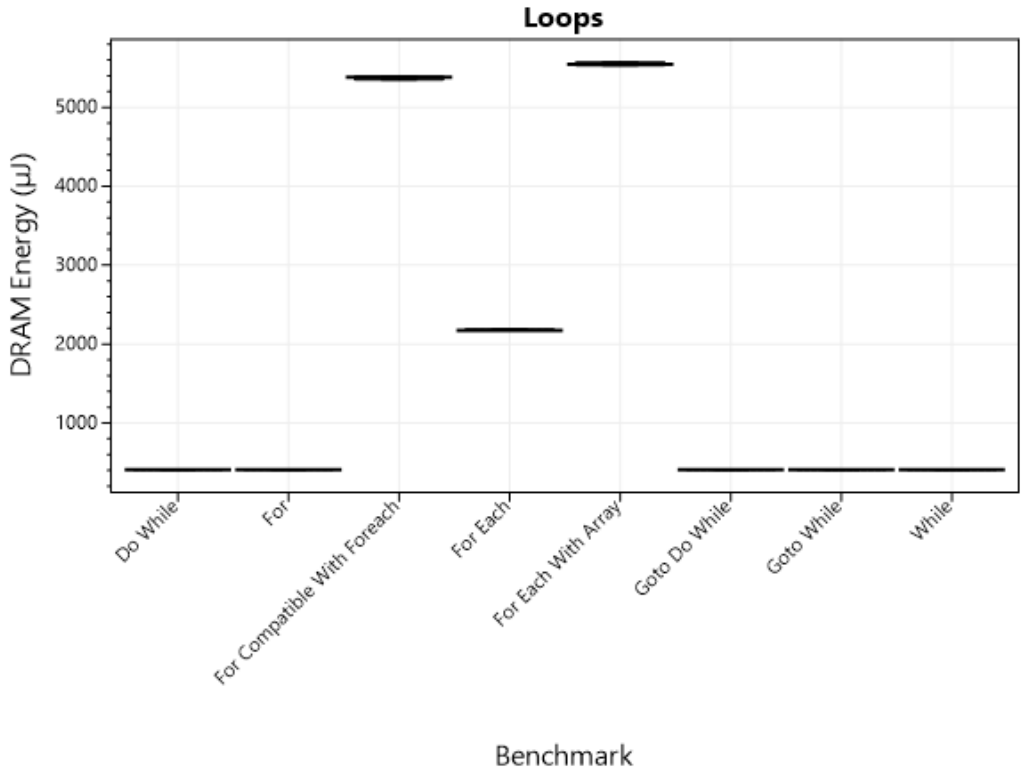

Figure A.55: Boxplot showing how efficient different Loops types are with regards to Dram Energy.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Do While	0,733968	6.523,162	408,200
For	0,734117	6.466,835	408,352
For Compatible With Foreach	4,913834	64.484,708	5.375,867
For Each	3,915805	53.030,853	2.177,913
For Each With Array	5,177604	67.874,405	5.548,147
Goto Do While	0,733960	6.501,446	408,473
Goto While	0,733934	6.645,797	408,238
While	0,733934	6.503,446	408,443

Table A.65: Table showing the elapsed time and energy measurement for each Loops.

Elapsed Time *p*-Values	Do While	While	For	For Each	For Compatible With Foreach	For Each With Array	Goto Do While	Goto While
Do While	-	<0,05	0,465	<0,05	<0,05	<0,05	0,665	<0,05
While	<0,05	-	0,373	<0,05	<0,05	<0,05	0,129	0,951
For	0,465	0,373	-	<0,05	<0,05	<0,05	0,443	0,153
For Each	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
For Compatible With Foreach	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05
For Each With Array	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Goto Do While	0,665	0,129	0,443	<0,05	<0,05	<0,05	-	0,105
Goto While	<0,05	0,951	0,153	<0,05	<0,05	<0,05	0,105	-

Table A.66: Table showing the *p*-values for the group Loops with regards to Elapsed Time.

Package Energy *p*-Values	Do While	While	For	For Each	For Compatible With Foreach	For Each With Array	Goto Do While	Goto While
Do While	-	0,171	<0,05	<0,05	<0,05	<0,05	0,298	<0,05
While	0,171	-	<0,05	<0,05	<0,05	<0,05	0,924	<0,05
For	<0,05	<0,05	-	<0,05	<0,05	<0,05	0,114	<0,05
For Each	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
For Compatible With Foreach	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05
For Each With Array	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Goto Do While	0,298	0,924	0,114	<0,05	<0,05	<0,05	-	<0,05
Goto While	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-

Table A.67: Table showing the *p*-values for the group Loops with regards to Package Energy.

DRAM Energy *p*-Values	Do While	While	For	For Each	For Compatible With Foreach	For Each With Array	Goto Do While	Goto While
Do While	-	0,386	0,527	<0,05	<0,05	<0,05	0,359	0,833
While	0,386	-	0,755	<0,05	<0,05	<0,05	0,929	0,325
For	0,527	0,755	-	<0,05	<0,05	<0,05	0,694	0,544
For Each	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
For Compatible With Foreach	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05
For Each With Array	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Goto Do While	0,359	0,929	0,694	<0,05	<0,05	<0,05	-	0,280
Goto While	0,833	0,325	0,544	<0,05	<0,05	<0,05	0,280	-

Table A.68: Table showing the *p*-values for the group Loops with regards to DRAM Energy.

LINQ

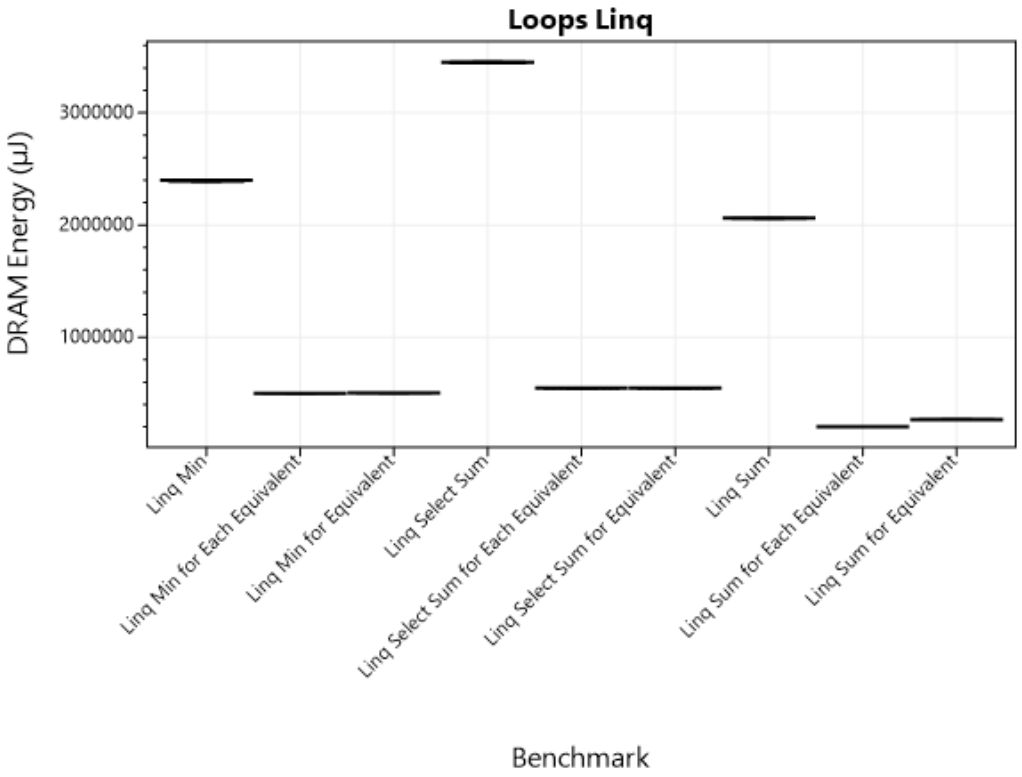

Figure A.56: Boxplot showing how efficient different Loops Linq types are with regards to DRAM Energy.

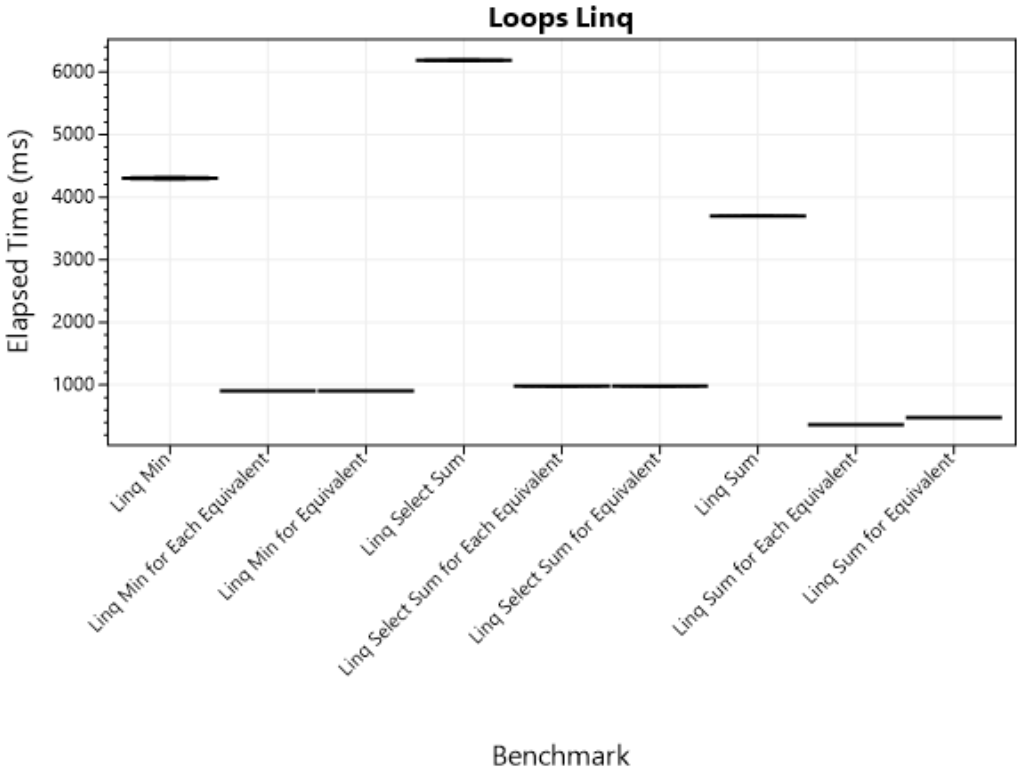

Figure A.57: Boxplot showing how efficient different Loops Linq types are with regards to Elapsed Time.

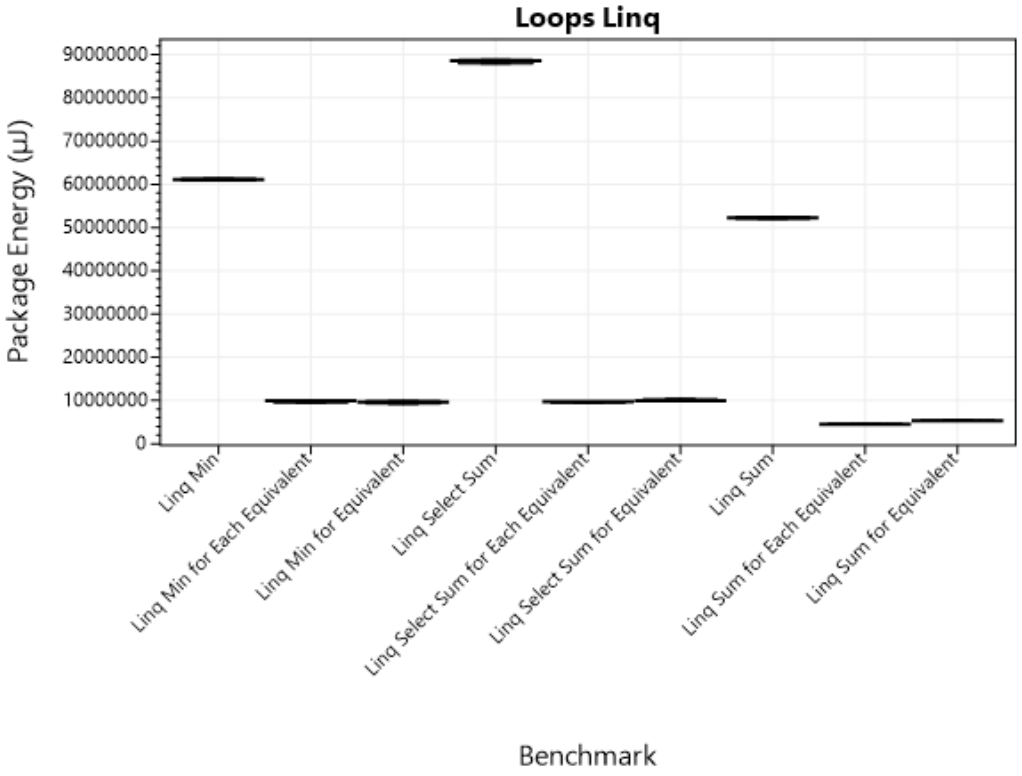

Figure A.58: Boxplot showing how efficient different Loops Linq types are with regards to Package Energy.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Linq Min	4.305,490723	61.124.822,320	2.397.591,146
Linq Min for Each Equivalent	901,496448	9.844.806,885	501.251,343
Linq Min for Equivalent	904,147862	9.597.473,338	502.756,173
Linq Select Sum	6.188,868815	88.412.171,766	3.446.462,674
Linq Select Sum for Each Equivalent	978,779856	9.658.162,944	544.465,129
Linq Select Sum for Equivalent	978,846325	9.946.287,883	544.398,984
Linq Sum	3.698,099772	52.233.071,560	2.059.501,139
Linq Sum for Each Equivalent	362,270846	4.462.221,955	201.459,373
Linq Sum for Equivalent	478,879886	5.327.824,063	266.554,091

Table A.69: Table showing the elapsed time and energy measurement for each Loops Linq.

Elapsed Time *p*-Values	Linq Sum	Linq Sum for Each Equivalent	Linq Sum for Equivalent	Linq Min	Linq Min for Each Equivalent	Linq Min for Equivalent	Linq Select Sum	Linq Select Sum for Each Equivalent	Linq Select Sum for Equivalent
Linq Sum	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Linq Sum for Each Equivalent	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Linq Sum for Equivalent	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Linq Min	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Linq Min for Each Equivalent	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Linq Min for Equivalent	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05
Linq Select Sum	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Linq Select Sum for Each Equivalent	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	0,616
Linq Select Sum for Equivalent	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	0,616	-

Table A.70: Table showing the *p*-values for the group Loops Linq with regards to Elapsed Time.

Package Energy *p*-Values	Linq Sum	Linq Sum for Each Equivalent	Linq Sum for Equivalent	Linq Min	Linq Min for Each Equivalent	Linq Min for Equivalent	Linq Select Sum	Linq Select Sum for Each Equivalent	Linq Select Sum for Equivalent
Linq Sum	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Linq Sum for Each Equivalent	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Linq Sum for Equivalent	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Linq Min	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Linq Min for Each Equivalent	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Linq Min for Equivalent	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	0,296	<0,05
Linq Select Sum	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Linq Select Sum for Each Equivalent	<0,05	<0,05	<0,05	<0,05	<0,05	0,296	<0,05	-	<0,05
Linq Select Sum for Equivalent	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-

Table A.71: Table showing the *p*-values for the group Loops Linq with regards to Package Energy.

DRAM Energy *p*-Values	Linq Sum	Linq Sum for Each Equivalent	Linq Sum for Equivalent	Linq Min	Linq Min for Each Equivalent	Linq Min for Equivalent	Linq Select Sum	Linq Select Sum for Each Equivalent	Linq Select Sum for Equivalent
Linq Sum	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Linq Sum for Each Equivalent	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Linq Sum for Equivalent	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Linq Min	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Linq Min for Each Equivalent	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Linq Min for Equivalent	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05
Linq Select Sum	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Linq Select Sum for Each Equivalent	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	0,838
Linq Select Sum for Equivalent	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	0,838	-

Table A.72: Table showing the *p*-values for the group Loops Linq with regards to DRAM Energy.

A.3.6 Collections

List Creation

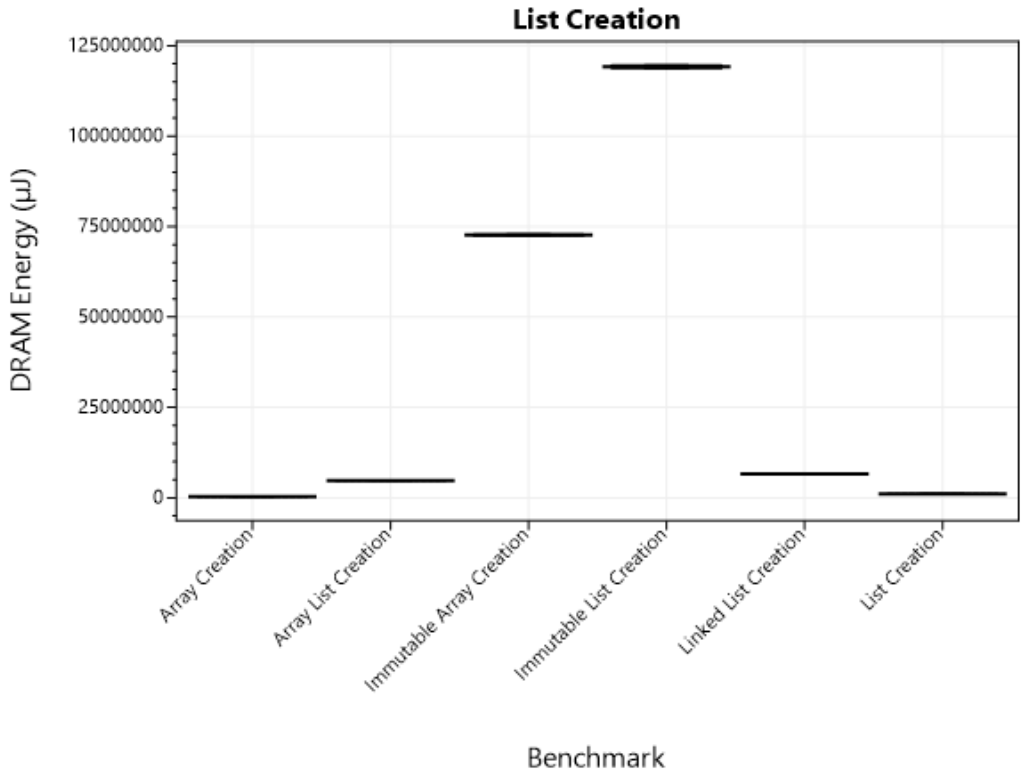

Figure A.59: Boxplot showing how efficient different List Creation types are with regards to DRAM Energy.

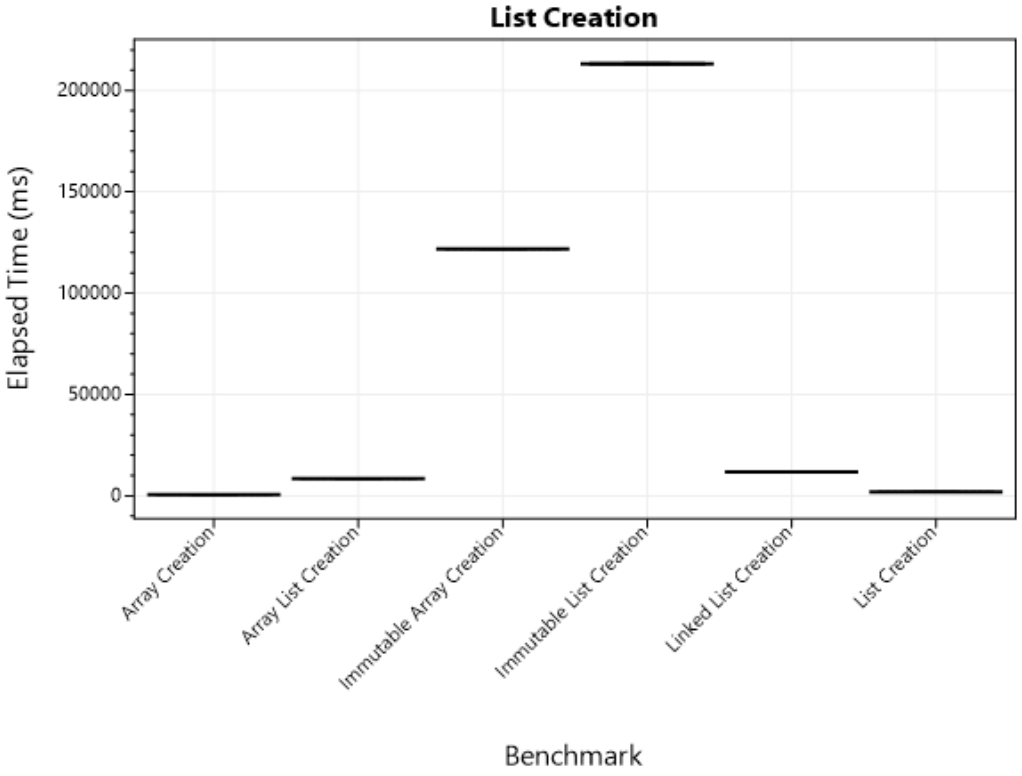

Figure A.60: Boxplot showing how efficient different List Creation types are with regards to Elapsed Time.

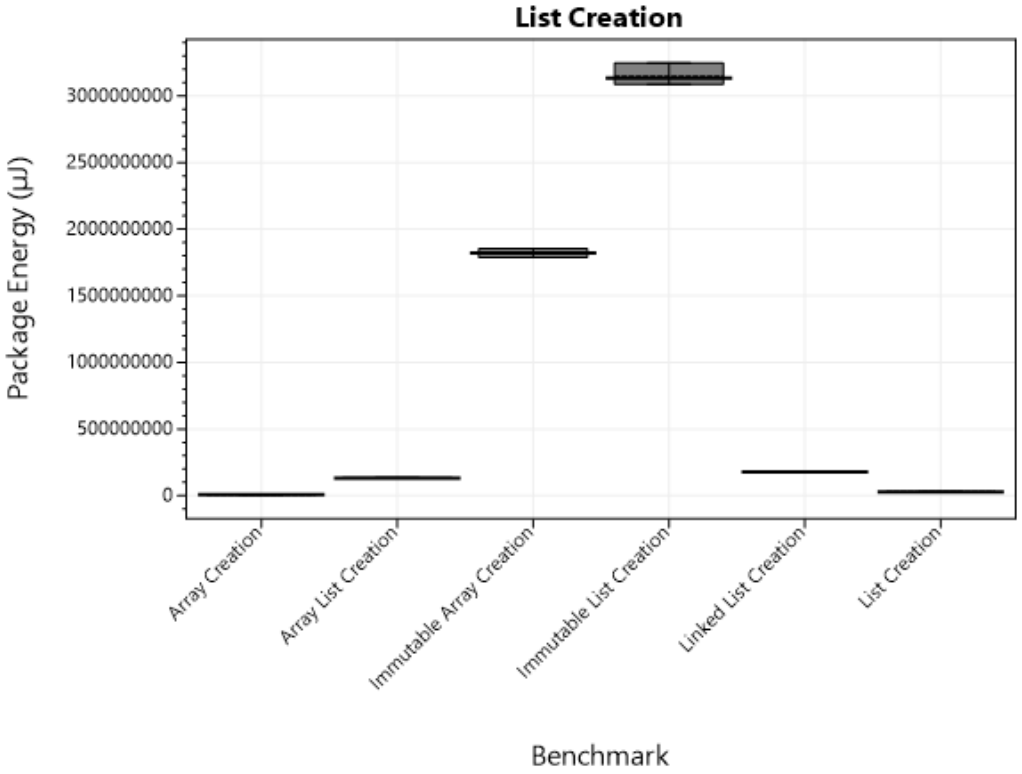

Figure A.61: Boxplot showing how efficient different List Creation types are with regards to Package Energy.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Array Creation	531,356894	7.263.987,160	306.578,954
Array List Creation	8.420,845147	132.747.527,445	4.789.323,478
Immutable Array Creation	121.797,539062	1.821.554.185,268	72.627.232,143
Immutable List Creation	213.035,526620	3.148.107.957,176	119.000.607,639
Linked List Creation	11.705,586480	179.121.763,780	6.611.794,705
List Creation	1.851,999316	27.697.289,167	1.057.061,986

Table A.73: Table showing the elapsed time and energy measurement for each List Creation.

Elapsed Time *p*-Values	Array Creation	Array List Creation	Immutable Array Creation	Immutable List Creation	Linked List Creation	List Creation
Array Creation	-	<0,05	<0,05	<0,05	<0,05	<0,05
Array List Creation	<0,05	-	<0,05	<0,05	<0,05	<0,05
Immutable Array Creation	<0,05	<0,05	-	<0,05	<0,05	<0,05
Immutable List Creation	<0,05	<0,05	<0,05	-	<0,05	<0,05
Linked List Creation	<0,05	<0,05	<0,05	<0,05	-	<0,05
List Creation	<0,05	<0,05	<0,05	<0,05	<0,05	-

Table A.74: Table showing the *p*-values for the group List Creation with regards to Elapsed Time.

Package Energy *p*-Values	Array Creation	Array List Creation	Immutable Array Creation	Immutable List Creation	Linked List Creation	List Creation
Array Creation	-	<0,05	<0,05	<0,05	<0,05	<0,05
Array List Creation	<0,05	-	<0,05	<0,05	<0,05	<0,05
Immutable Array Creation	<0,05	<0,05	-	<0,05	<0,05	<0,05
Immutable List Creation	<0,05	<0,05	<0,05	-	<0,05	<0,05
Linked List Creation	<0,05	<0,05	<0,05	<0,05	-	<0,05
List Creation	<0,05	<0,05	<0,05	<0,05	<0,05	-

Table A.75: Table showing the *p*-values for the group List Creation with regards to Package Energy.

DRAM Energy *p*-Values	Array Creation	Array List Creation	Immutable Array Creation	Immutable List Creation	Linked List Creation	List Creation
Array Creation	-	<0,05	<0,05	<0,05	<0,05	<0,05
Array List Creation	<0,05	-	<0,05	<0,05	<0,05	<0,05
Immutable Array Creation	<0,05	<0,05	-	<0,05	<0,05	<0,05
Immutable List Creation	<0,05	<0,05	<0,05	-	<0,05	<0,05
Linked List Creation	<0,05	<0,05	<0,05	<0,05	-	<0,05
List Creation	<0,05	<0,05	<0,05	<0,05	<0,05	-

Table A.76: Table showing the *p*-values for the group List Creation with regards to DRAM Energy.

List Get

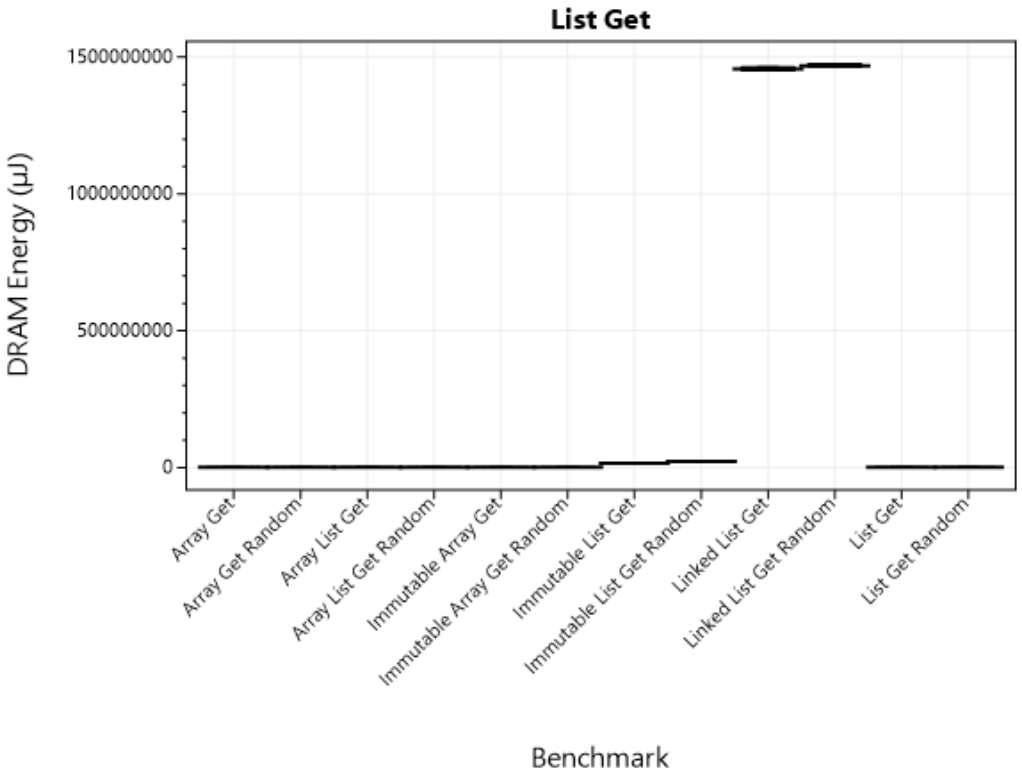

Figure A.62: Boxplot showing how efficient different List Get types are with regards to DRAM Energy.

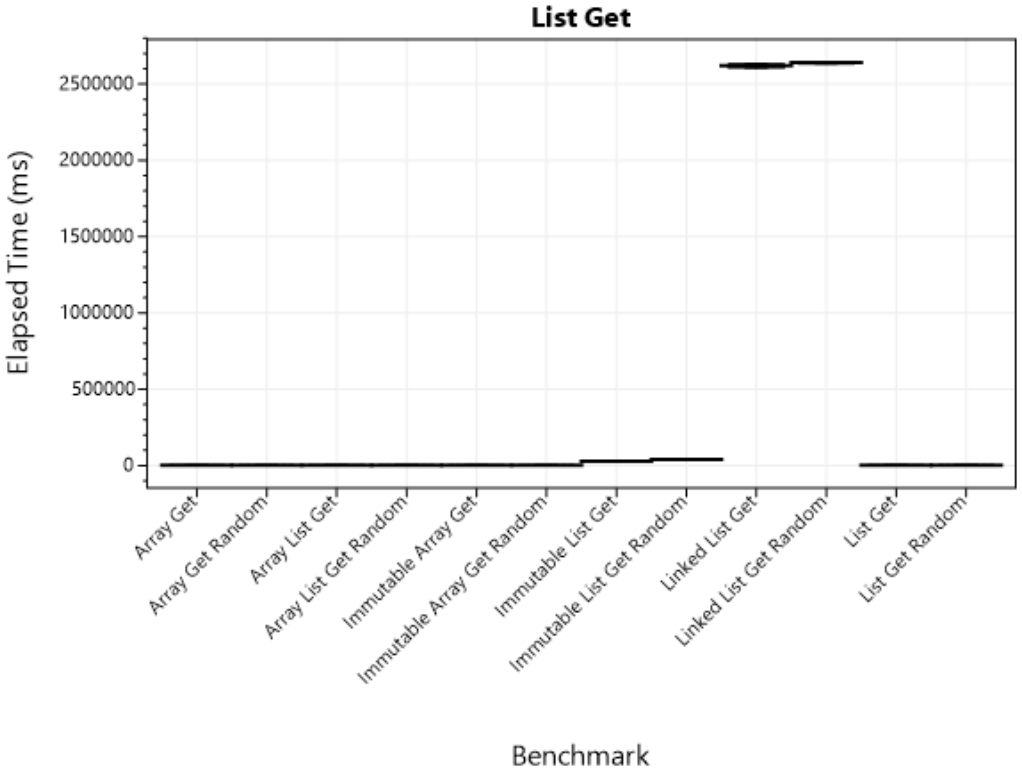

Figure A.63: Boxplot showing how efficient different List Get types are with regards to Elapsed Time.

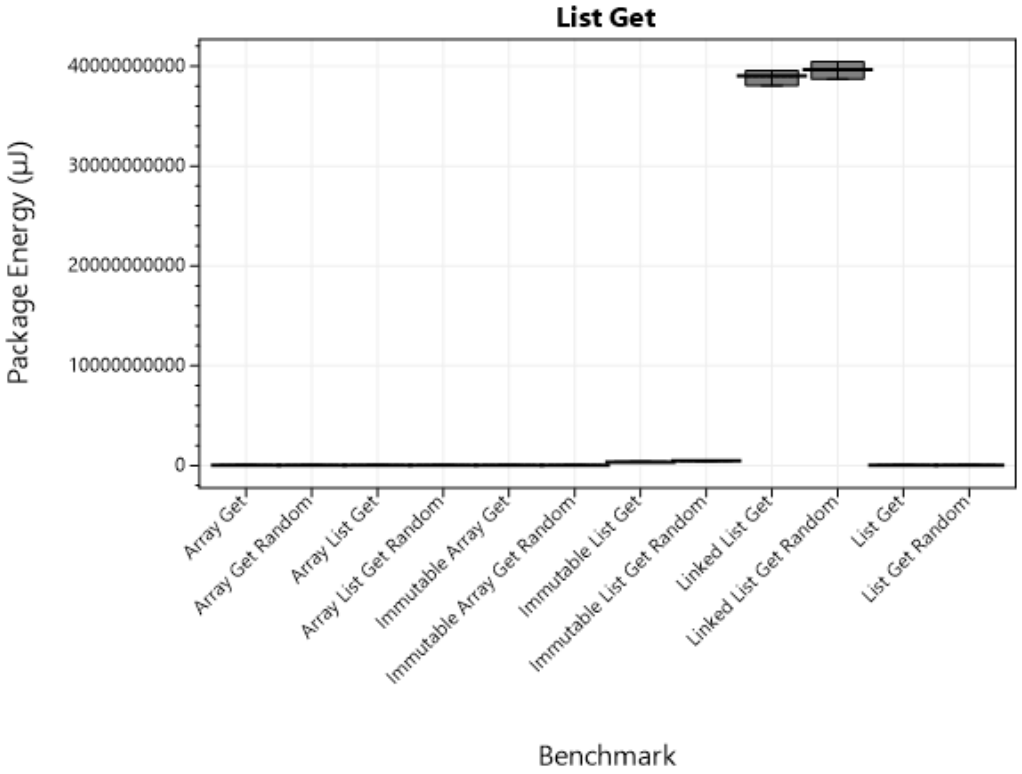

Figure A.64: Boxplot showing how efficient different List Get types are with regards to Package Energy.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Array Get	617,467346	8.794.373,745	343.657,260
Array Get Random	617,463871	8.744.610,426	343.377,686
Array List Get	2.306,950468	35.512.334,487	1.283.994,428
Array List Get Random	2.140,269572	32.721.541,341	1.190.599,908
Immutable Array Get	638,782772	9.042.632,718	355.391,778
Immutable Array Get Random	638,834178	9.154.411,825	355.370,924
Immutable List Get	28.293,397352	359.966.427,951	15.739.062,500
Immutable List Get Random	39.624,610460	476.131.944,444	22.045.713,976
Linked List Get	2.618.687,375000	38.967.090.000,000	1.457.019.687,500
Linked List Get Random	2.639.230,050000	39.638.372.000,000	1.469.158.250,000
List Get	741,488156	10.531.627,401	412.543,742
List Get Random	741,498125	10.633.192,952	412.429,979

Table A.77: Table showing the elapsed time and energy measurement for each List Get.

Elapsed Time *p*-Values	Array Get	Array Get Random	Array List Get	Array List Get Random	Immutable Array Get	Immutable Array Get Random	Immutable List Get	Immutable List Get Random	Linked List Get	Linked List Get Random	List Get	List Get Random
Array Get	-	0,844	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Array Get Random	0,844	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Array List Get	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Array List Get Random	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Immutable Array Get	<0,05	<0,05	<0,05	<0,05	-	0,276	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Immutable Array Get Random	<0,05	<0,05	<0,05	<0,05	0,276	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Immutable List Get	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Immutable List Get Random	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Linked List Get	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05
Linked List Get Random	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
List Get	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	0,644
List Get Random	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	0,644	-

Table A.78: Table showing the *p*-values for the group List Get with regards to Elapsed Time.

Package Energy *p*-Values	Array Get	Array Get Random	Array List Get	Array List Get Random	Immutable Array Get
Array Get	-	<0,05	<0,05	<0,05	<0,05
Array Get Random	<0,05	-	<0,05	<0,05	<0,05
Array List Get	<0,05	<0,05	-	<0,05	<0,05
Array List Get Random	<0,05	<0,05	<0,05	-	<0,05
Immutable Array Get	<0,05	<0,05	<0,05	<0,05	-
Immutable Array Get Random	<0,05	<0,05	<0,05	<0,05	<0,05
Immutable List Get	<0,05	<0,05	<0,05	<0,05	<0,05
Immutable List Get Random	<0,05	<0,05	<0,05	<0,05	<0,05
List Get	<0,05	<0,05	<0,05	<0,05	<0,05
List Get Random	<0,05	<0,05	<0,05	<0,05	<0,05

Package Energy *p*-Values	Immutable Array Get Random	Immutable List Get	Immutable List Get Random	List Get	List Get Random
Array Get	<0,05	<0,05	<0,05	<0,05	<0,05
Array Get Random	<0,05	<0,05	<0,05	<0,05	<0,05
Array List Get	<0,05	<0,05	<0,05	<0,05	<0,05
Array List Get Random	<0,05	<0,05	<0,05	<0,05	<0,05
Immutable Array Get	<0,05	<0,05	<0,05	<0,05	
Immutable Array Get Random	-	<0,05	<0,05	<0,05	<0,05
Immutable List Get	<0,05	-	<0,05	<0,05	<0,05
Immutable List Get Random	<0,05	<0,05	-	<0,05	<0,05
List Get	<0,05	<0,05	<0,05	-	<0,05
List Get Random	<0,05	<0,05	<0,05	<0,05	-

Table A.79: Tables showing the *p*-values for the group List Get with regards to Package Energy.

DRAM Energy *p*-Values	Array Get	Array Get Random	Array List Get	Array List Get Random	Immutable Array Get	Immutable Array Get Random	Immutable List Get	Immutable List Get Random	Linked List Get	Linked List Get Random	List Get	List Get Random
Array Get	-	0,166	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Array Get Random	0,166	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Array List Get	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Array List Get Random	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Immutable Array Get	<0,05	<0,05	<0,05	<0,05	-	0,925	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Immutable Array Get Random	<0,05	<0,05	<0,05	<0,05	0,925	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Immutable List Get	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Immutable List Get Random	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Linked List Get	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05
Linked List Get Random	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
List Get	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	0,286
List Get Random	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	0,286	-

Table A.80: Table showing the *p*-values for the group List Get with regards to DRAM Energy.

List Copy

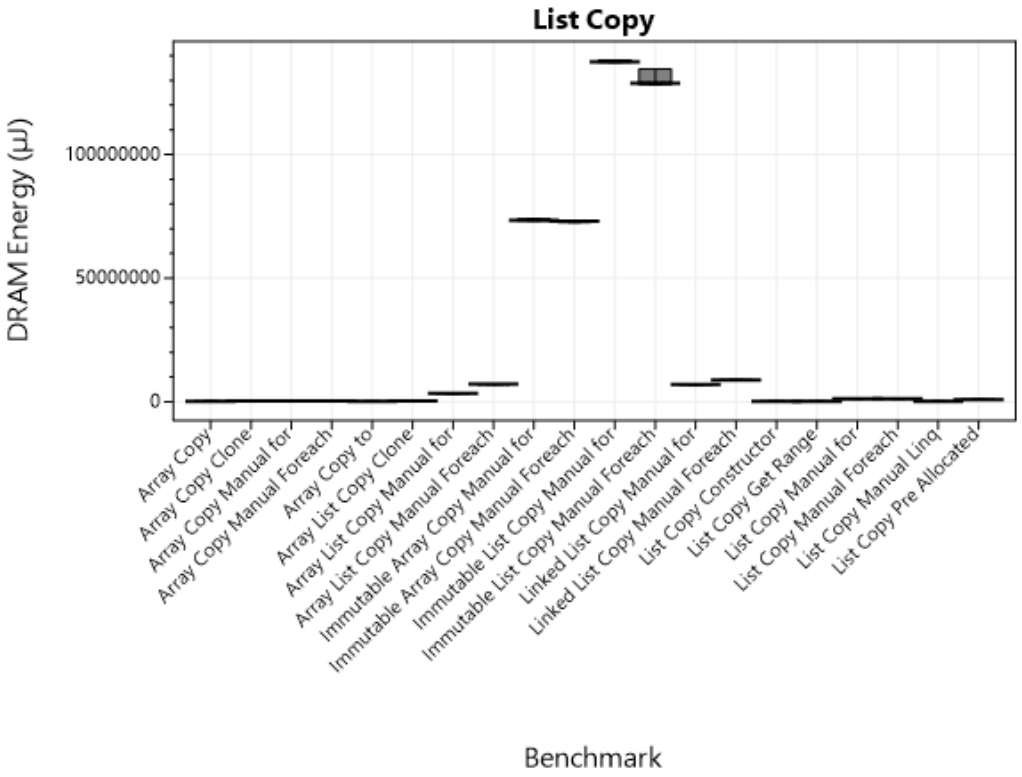

Figure A.65: Boxplot showing how efficient different List Copy types are with regards to DRAM Energy.

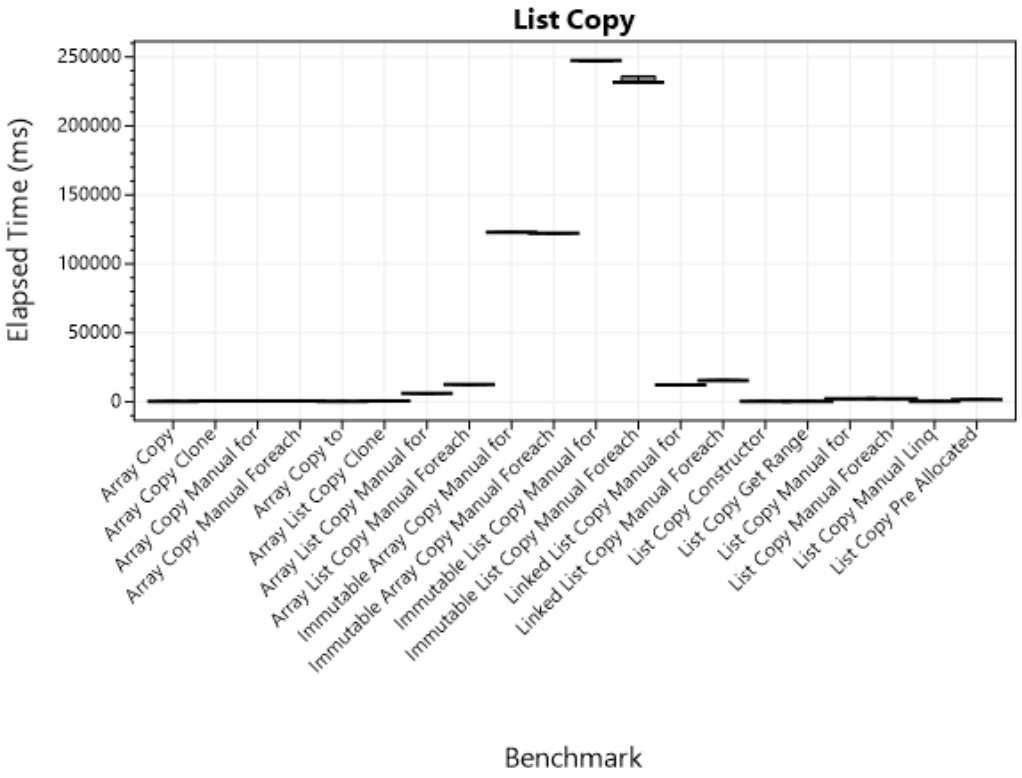

Figure A.66: Boxplot showing how efficient different List Copy types are with regards to Elapsed Time.

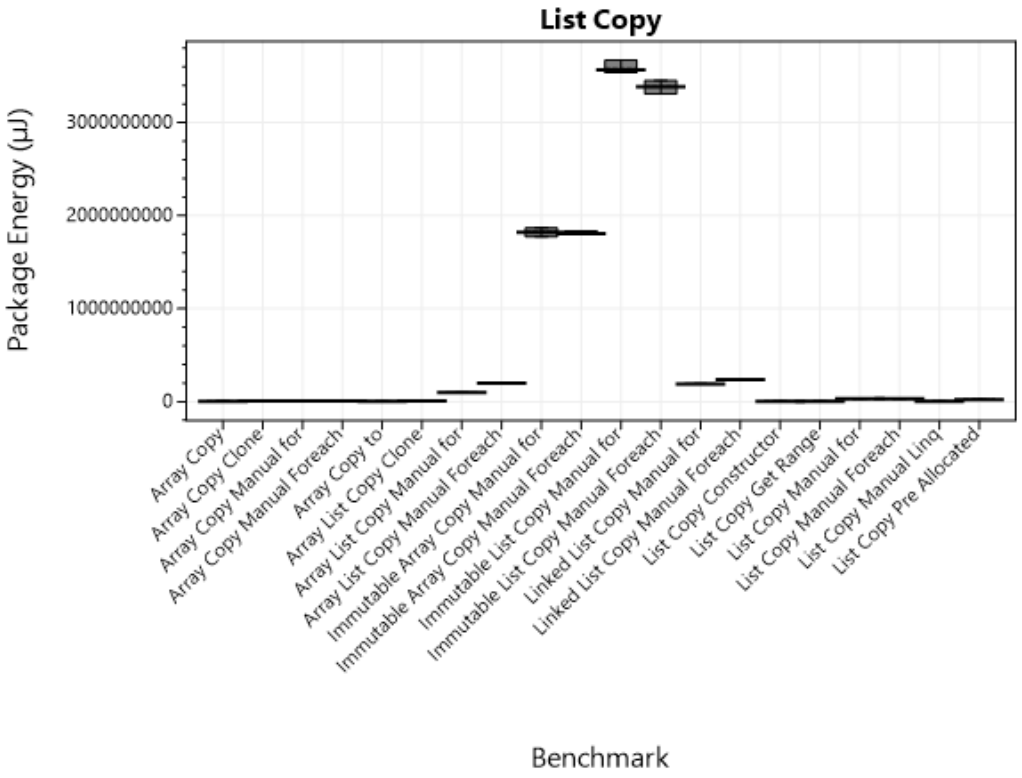

Figure A.67: Boxplot showing how efficient different List Copy types are with regards to Package Energy.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Array Copy	216,408693	3.121.486,185	130.368,265
Array Copy Clone	360,102231	5.441.618,195	220.644,623
Array Copy Manual for	635,253174	9.185.991,028	363.985,962
Array Copy Manual Foreach	636,243404	9.583.979,739	364.328,942
Array Copy to	215,045644	3.140.696,746	128.653,611
Array List Copy Clone	534,230268	8.062.278,157	314.329,603
Array List Copy Manual for	5.890,928467	95.623.337,402	3.319.149,170
Array List Copy Manual Foreach	12.437,360026	199.956.433,105	7.034.077,962
Immutable Array Copy Manual for	122.754,895368	1.823.348.242,187	73.371.428,571
Immutable Array Copy Manual Foreach	122.126,164062	1.814.758.906,250	72.905.859,375
Immutable List Copy Manual for	247.215,997243	3.581.458.639,706	137.628.814,338
Immutable List Copy Manual Foreach	231.815,328947	3.391.733.963,816	129.305.386,513
Linked List Copy Manual for	12.221,221830	188.772.468,450	6.897.385,817
Linked List Copy Manual Foreach	15.467,489692	236.531.157,769	8.706.564,670
List Copy Constructor	230,448949	3.435.943,126	140.742,686
List Copy Get Range	227,289248	3.315.948,809	138.935,070
List Copy Manual for	2.076,674805	31.438.595,920	1.184.928,385
List Copy Manual Foreach	2.099,194472	32.131.084,527	1.199.001,736
List Copy Manual Linq	229,872768	3.403.422,981	140.119,286
List Copy Pre Allocated	1.391,106479	20.463.207,186	789.469,381

Table A.81: Table showing the elapsed time and energy measurement for each List Copy.

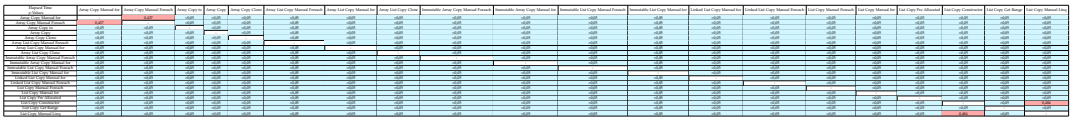

Table A.82: Table showing the *p*-values for the group List Copy with regards to Elapsed Time.

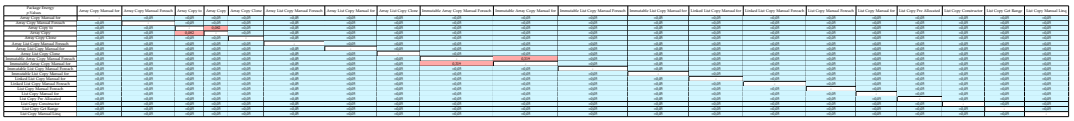

Table A.83: Table showing the *p*-values for the group List Copy with regards to Package Energy.

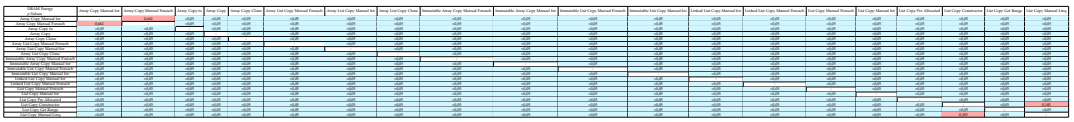

Table A.84: Table showing the *p*-values for the group List Copy with regards to DRAM Energy.

List Insertion

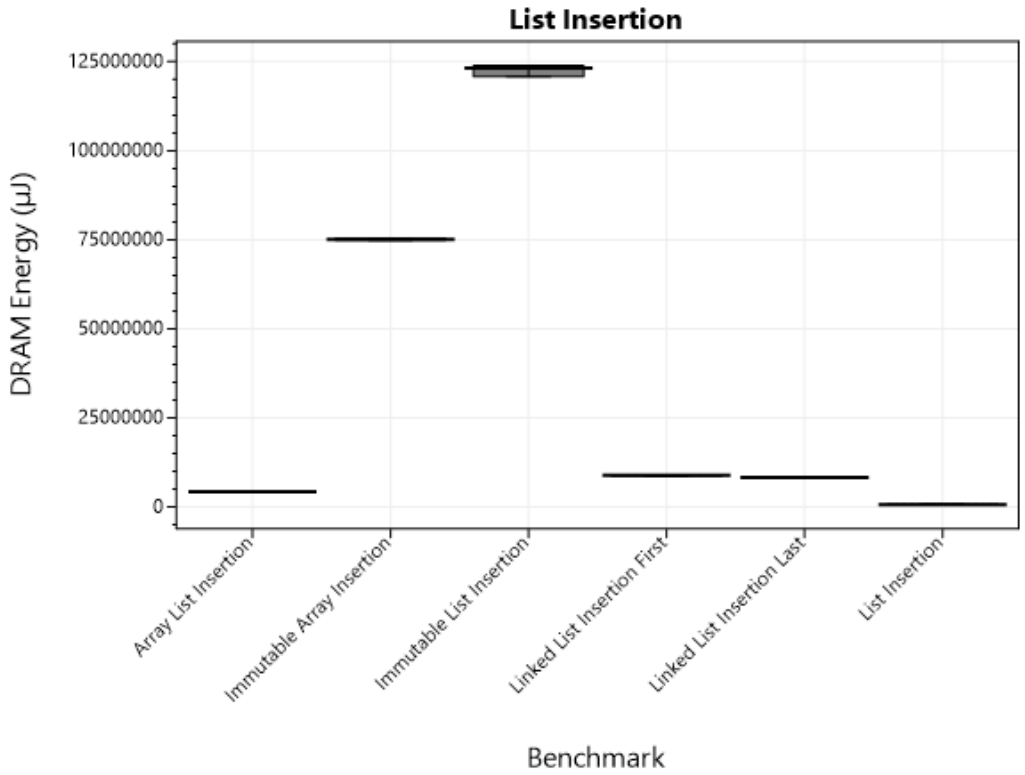

Figure A.68: Boxplot showing how efficient different List Insertion types are with regards to DRAM Energy.

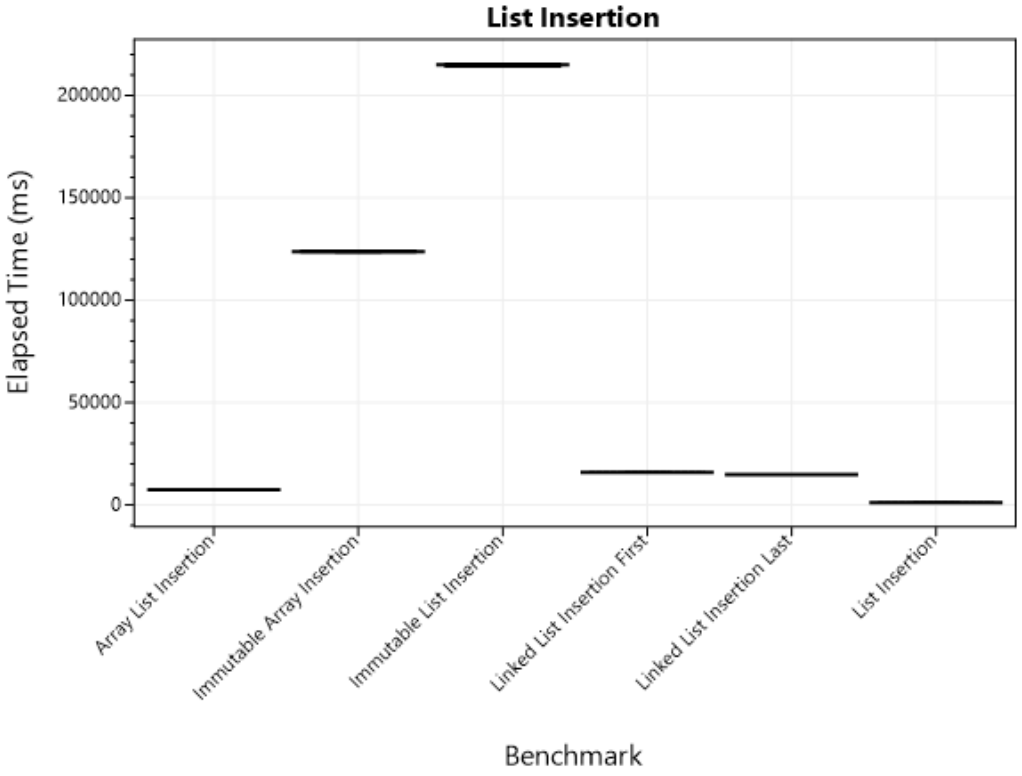

Figure A.69: Boxplot showing how efficient different List Insertion types are with regards to Elapsed Time.

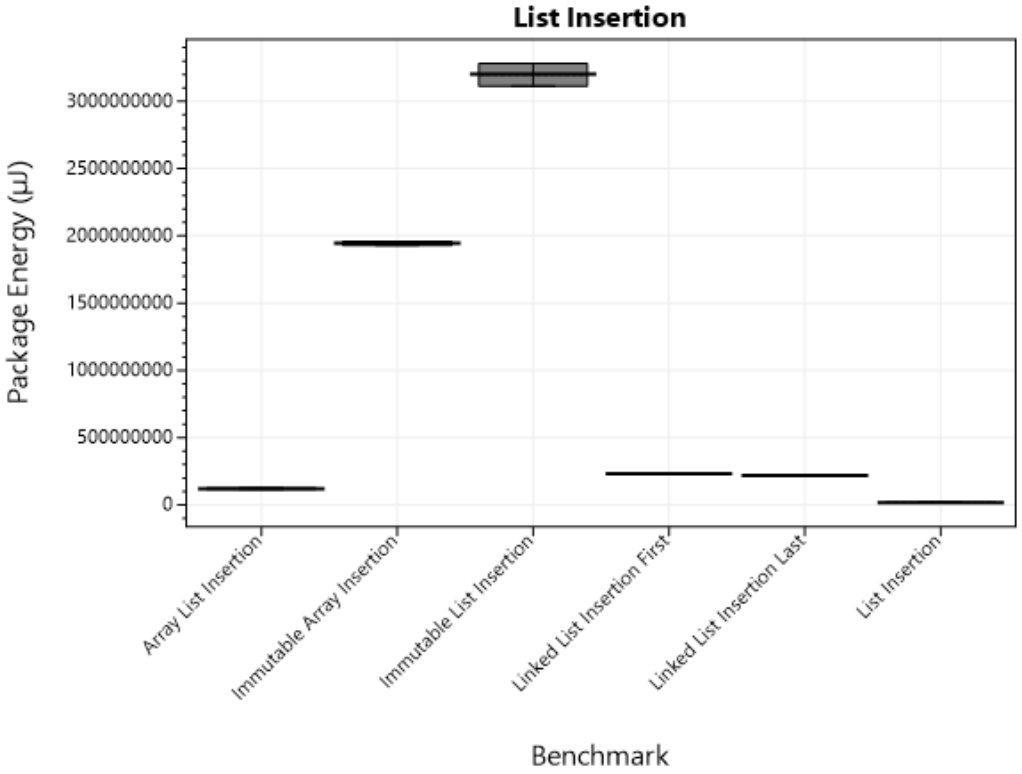

Figure A.70: Boxplot showing how efficient different List Insertion types are with regards to Package Energy.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Array List Insertion	7.560,993490	118.082.856,988	4.291.722,548
Immutable Array Insertion	123.686,914062	1.943.715.755,208	75.138.758,681
Immutable List Insertion	214.848,097747	3.197.835.901,163	122.871.784,157
Linked List Insertion First	15.886,229980	232.091.845,703	8.837.456,055
Linked List Insertion Last	14.829,519531	219.373.139,648	8.253.730,469
List Insertion	1.218,500137	17.845.165,761	678.032,176

Table A.85: Table showing the elapsed time and energy measurement for each List Insertion.

Elapsed Time *p*-Values	Array List Insertion	Immutable Array Insertion	Immutable List Insertion	Linked List Insertion Last	Linked List Insertion First	List Insertion
Array List Insertion	-	<0,05	<0,05	<0,05	<0,05	<0,05
Immutable Array Insertion	<0,05	-	<0,05	<0,05	<0,05	<0,05
Immutable List Insertion	<0,05	<0,05	-	<0,05	<0,05	<0,05
Linked List Insertion Last	<0,05	<0,05	<0,05	-	<0,05	<0,05
Linked List Insertion First	<0,05	<0,05	<0,05	<0,05	-	<0,05
List Insertion	<0,05	<0,05	<0,05	<0,05	<0,05	-

Table A.86: Table showing the *p*-values for the group List Insertion with regards to Elapsed Time.

Package Energy *p*-Values	Array List Insertion	Immutable Array Insertion	Immutable List Insertion	Linked List Insertion Last	Linked List Insertion First	List Insertion
Array List Insertion	-	<0,05	<0,05	<0,05	<0,05	<0,05
Immutable Array Insertion	<0,05	-	<0,05	<0,05	<0,05	<0,05
Immutable List Insertion	<0,05	<0,05	-	<0,05	<0,05	<0,05
Linked List Insertion Last	<0,05	<0,05	<0,05	-	<0,05	<0,05
Linked List Insertion First	<0,05	<0,05	<0,05	<0,05	-	<0,05
List Insertion	<0,05	<0,05	<0,05	<0,05	<0,05	-

Table A.87: Table showing the *p*-values for the group List Insertion with regards to Package Energy.

DRAM Energy *p*-Values	Array List Insertion	Immutable Array Insertion	Immutable List Insertion	Linked List Insertion Last	Linked List Insertion First	List Insertion
Array List Insertion	-	<0,05	<0,05	<0,05	<0,05	<0,05
Immutable Array Insertion	<0,05	-	<0,05	<0,05	<0,05	<0,05
Immutable List Insertion	<0,05	<0,05	-	<0,05	<0,05	<0,05
Linked List Insertion Last	<0,05	<0,05	<0,05	-	<0,05	<0,05
Linked List Insertion First	<0,05	<0,05	<0,05	<0,05	-	<0,05
List Insertion	<0,05	<0,05	<0,05	<0,05	<0,05	-

Table A.88: Table showing the *p*-values for the group List Insertion with regards to DRAM Energy.

List Removal

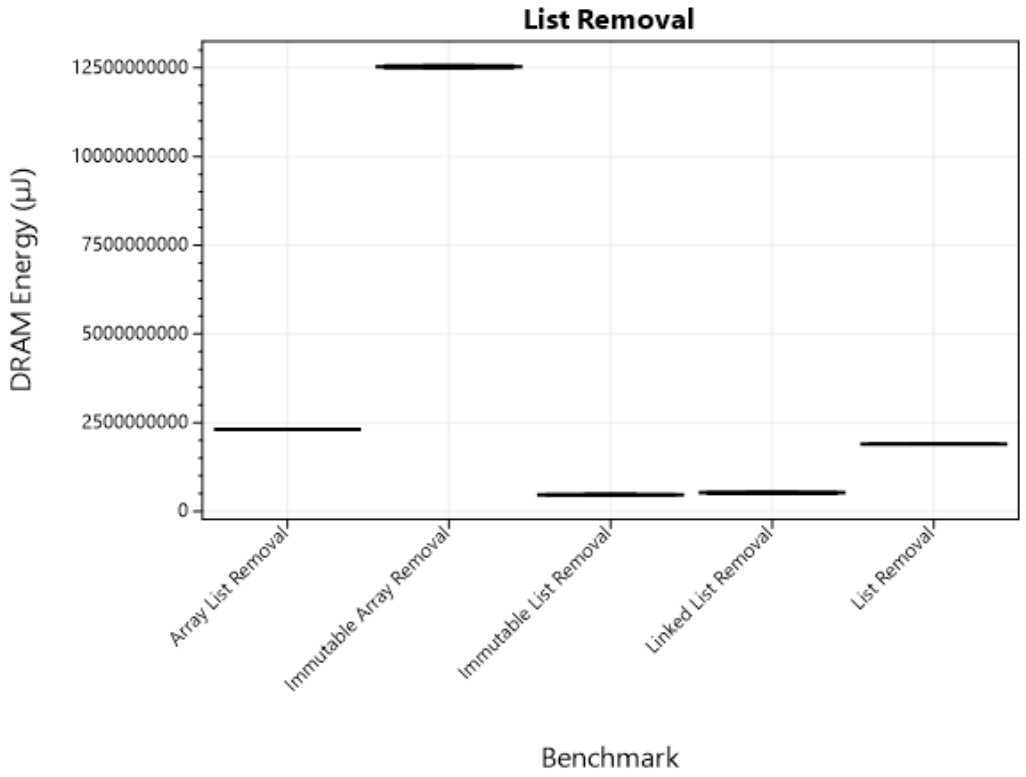

Figure A.71: Boxplot showing how efficient different List Removal types are with regards to DRAM Energy.

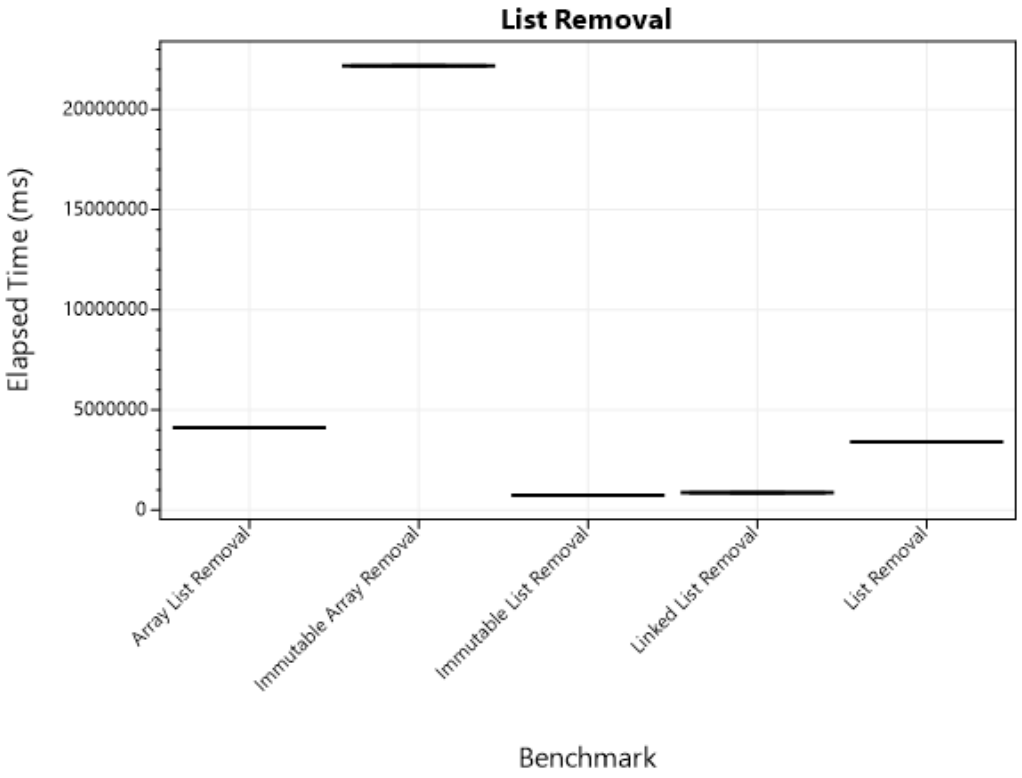

Figure A.72: Boxplot showing how efficient different List Removal types are with regards to Elapsed Time.

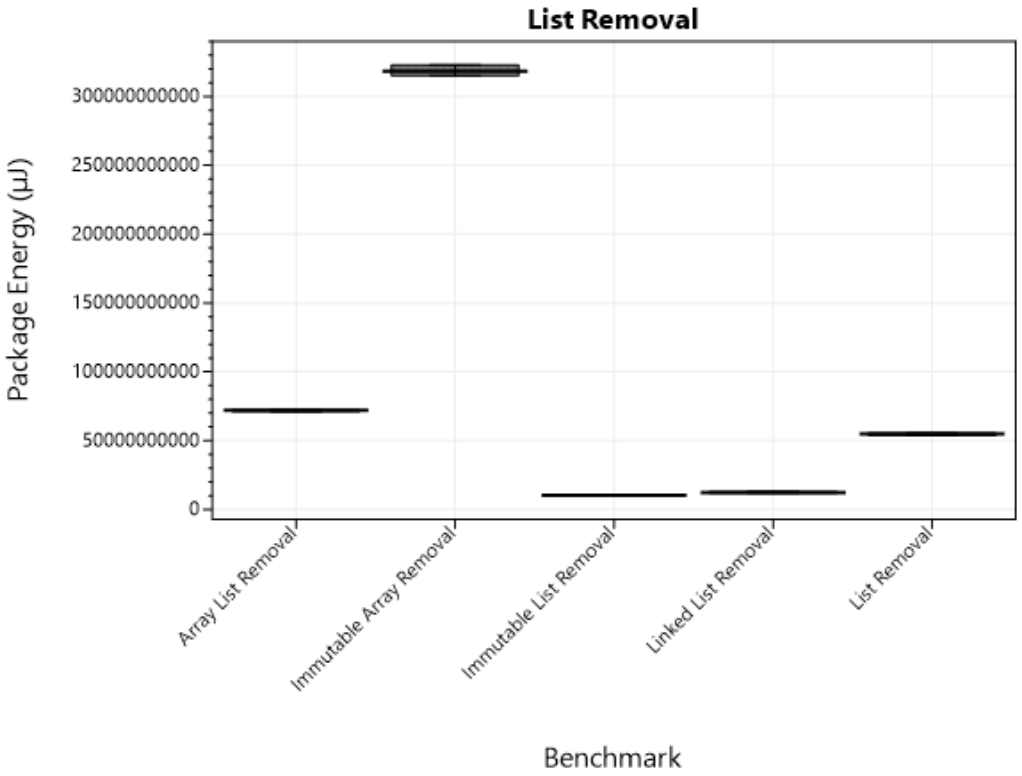

Figure A.73: Boxplot showing how efficient different List Removal types are with regards to Package Energy.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Array List Removal	4.117.686,666667	71.745.456.250,000	2.303.393.055,556
Immutable Array Removal	22.181.604,479167	318.774.679.166,667	12.531.503.125,000
Immutable List Removal	728.103,041858	10.353.928.913,417	464.979.773,509
Linked List Removal	852.388,915071	12.352.086.630,544	513.789.201,109
List Removal	3.400.574,174107	54.740.547.991,071	1.893.229.464,286

Table A.89: Table showing the elapsed time and energy measurement for each List Removal.

Elapsed Time *p*-Values	Array List Removal	Immutable Array Removal	Immutable List Removal	Linked List Removal	List Removal
Array List Removal	-	<0,05	<0,05	<0,05	<0,05
Immutable Array Removal	<0,05	-	<0,05	<0,05	<0,05
Immutable List Removal	<0,05	<0,05	-	<0,05	<0,05
Linked List Removal	<0,05	<0,05	<0,05	-	<0,05
List Removal	<0,05	<0,05	<0,05	<0,05	-

Table A.90: Table showing the *p*-values for the group List Removal with regards to Elapsed Time.

Package Energy *p*-Values	Array List Removal	Immutable Array Removal	Immutable List Removal	Linked List Removal	List Removal
Array List Removal	-	<0,05	<0,05	<0,05	<0,05
Immutable Array Removal	<0,05	-	<0,05	<0,05	<0,05
Immutable List Removal	<0,05	<0,05	-	<0,05	<0,05
Linked List Removal	<0,05	<0,05	<0,05	-	<0,05
List Removal	<0,05	<0,05	<0,05	<0,05	-

Table A.91: Table showing the *p*-values for the group List Removal with regards to Package Energy.

DRAM Energy *p*-Values	Array List Removal	Immutable Array Removal	Immutable List Removal	Linked List Removal	List Removal
Array List Removal	-	<0,05	<0,05	<0,05	<0,05
Immutable Array Removal	<0,05	-	<0,05	<0,05	<0,05
Immutable List Removal	<0,05	<0,05	-	<0,05	<0,05
Linked List Removal	<0,05	<0,05	<0,05	-	<0,05
List Removal	<0,05	<0,05	<0,05	<0,05	-

Table A.92: Table showing the *p*-values for the group List Removal with regards to DRAM Energy.

Set Creation

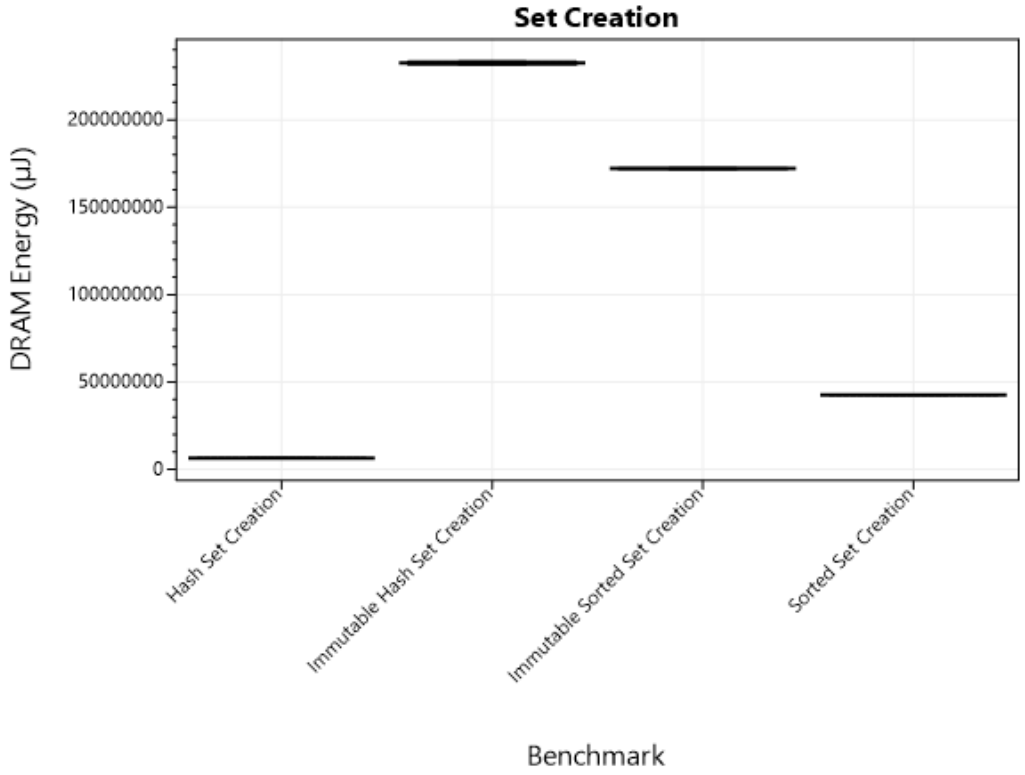

Figure A.74: Boxplot showing how efficient different Set Creation types are with regards to DRAM Energy.

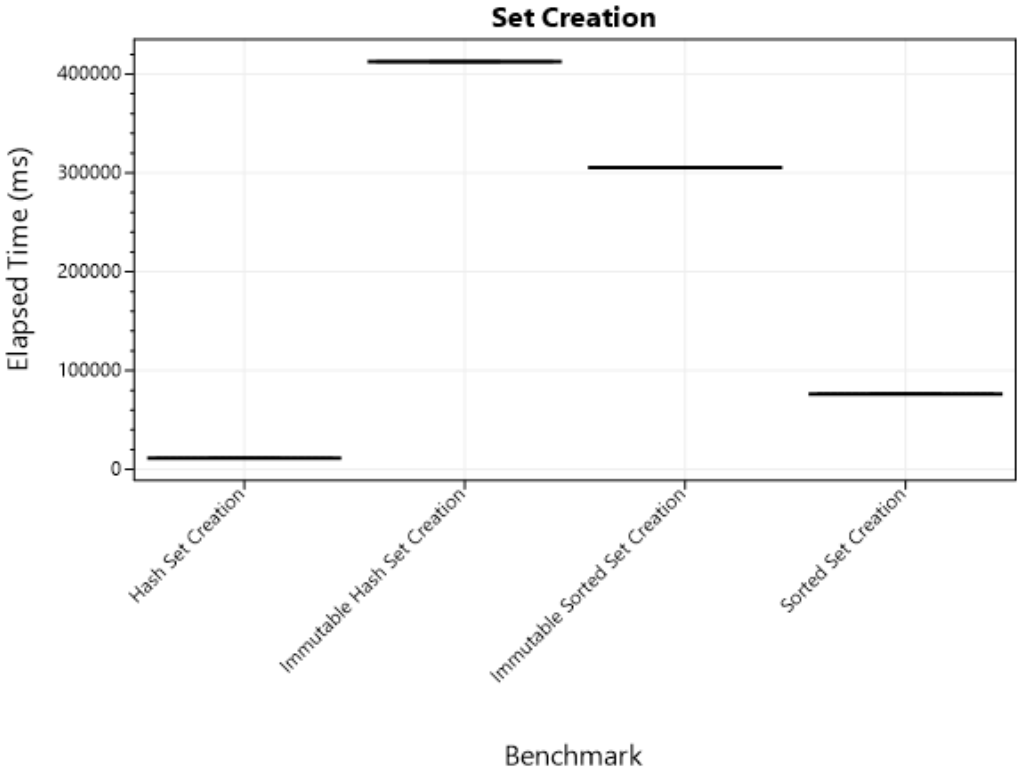

Figure A.75: Boxplot showing how efficient different Set Creation types are with regards to Elapsed Time.

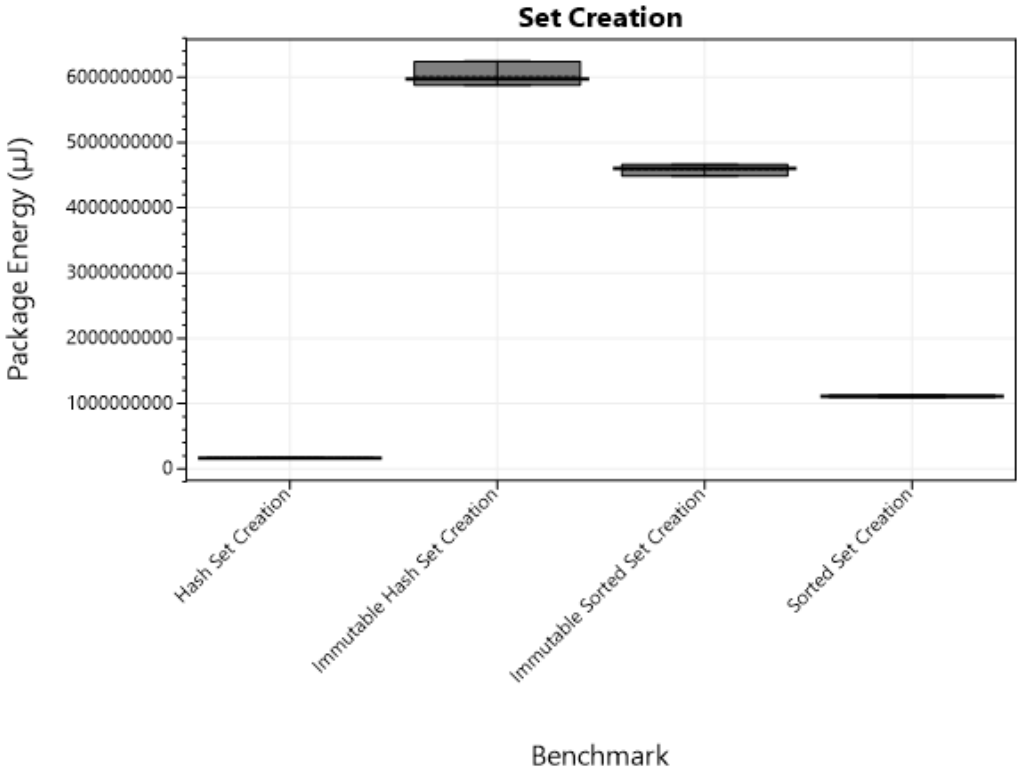

Figure A.76: Boxplot showing how efficient different Set Creation types are with regards to Package Energy.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Hash Set Creation	11.444,244249	163.936.528,863	6.500.699,870
Immutable Hash Set Creation	412.457,141335	6.007.757.883,523	232.414.169,034
Immutable Sorted Set Creation	305.442,459077	4.591.083.110,119	172.117.299,107
Sorted Set Creation	76.436,072049	1.109.430.316,840	42.731.098,090

Table A.93: Table showing the elapsed time and energy measurement for each Set Creation.

Elapsed Time *p*-Values	Hash Set Creation	Immutable Hash Set Creation	Immutable Sorted Set Creation	Sorted Set Creation
Hash Set Creation	-	<0,05	<0,05	<0,05
Immutable Hash Set Creation	<0,05	-	<0,05	<0,05
Immutable Sorted Set Creation	<0,05	<0,05	-	<0,05
Sorted Set Creation	<0,05	<0,05	<0,05	-

Table A.94: Table showing the *p*-values for the group Set Creation with regards to Elapsed Time.

Package Energy *p*-Values	Hash Set Creation	Immutable Hash Set Creation	Immutable Sorted Set Creation	Sorted Set Creation
Hash Set Creation	-	<0,05	<0,05	<0,05
Immutable Hash Set Creation	<0,05	-	<0,05	<0,05
Immutable Sorted Set Creation	<0,05	<0,05	-	<0,05
Sorted Set Creation	<0,05	<0,05	<0,05	-

Table A.95: Table showing the *p*-values for the group Set Creation with regards to Package Energy.

DRAM Energy *p*-Values	Hash Set Creation	Immutable Hash Set Creation	Immutable Sorted Set Creation	Sorted Set Creation
Hash Set Creation	-	<0,05	<0,05	<0,05
Immutable Hash Set Creation	<0,05	-	<0,05	<0,05
Immutable Sorted Set Creation	<0,05	<0,05	-	<0,05
Sorted Set Creation	<0,05	<0,05	<0,05	-

Table A.96: Table showing the *p*-values for the group Set Creation with regards to DRAM Energy.

Set Get

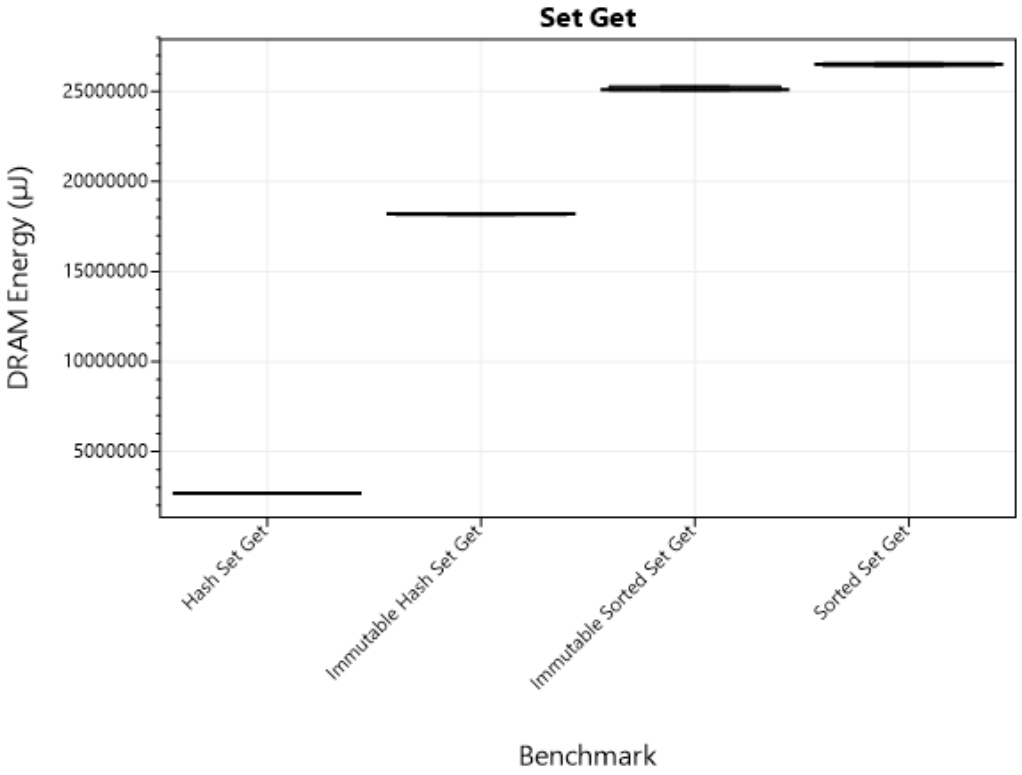

Figure A.77: Boxplot showing how efficient different Set Get types are with regards to DRAM Energy.

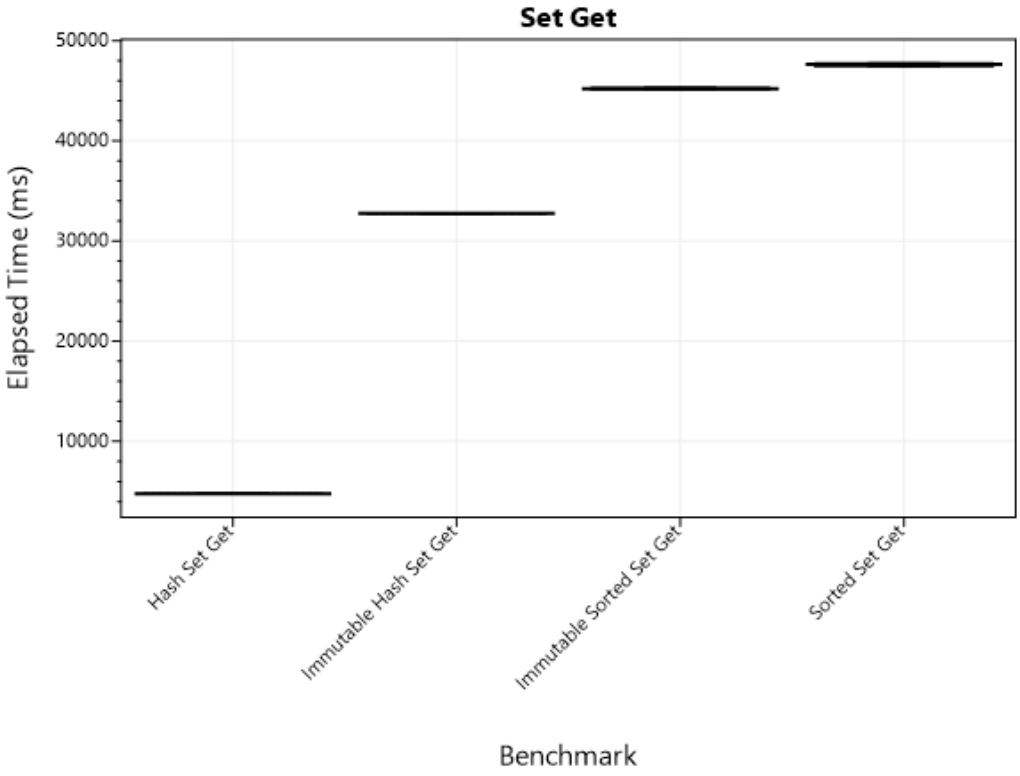

Figure A.78: Boxplot showing how efficient different Set Get types are with regards to Elapsed Time.

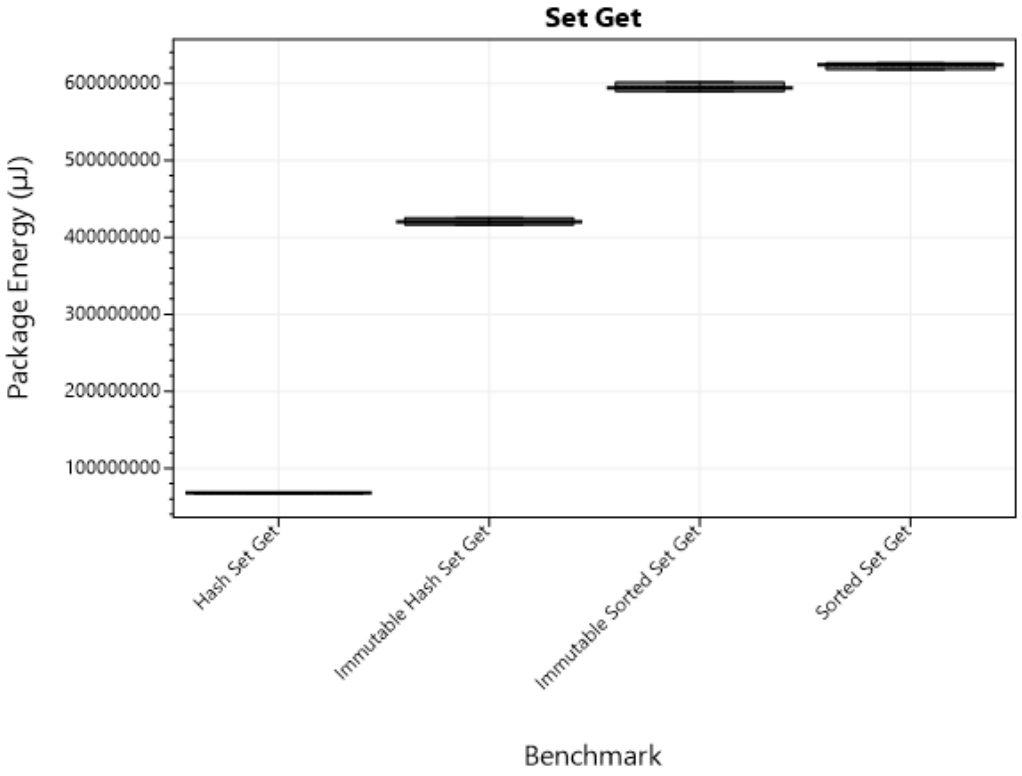

Figure A.79: Boxplot showing how efficient different Set Get types are with regards to Package Energy.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Hash Set Get	4.792,735053	67.619.639,757	2.665.976,291
Immutable Hash Set Get	32.736,826172	419.871.920,072	18.198.182,091
Immutable Sorted Set Get	45.180,626085	594.950.835,503	25.134.559,462
Sorted Set Get	47.600,834418	623.191.460,503	26.484.928,385

Table A.97: Table showing the elapsed time and energy measurement for each Set Get.

Elapsed Time *p*-Values	Hash Set Get	Immutable Hash Set Get	Immutable Sorted Set Get	Sorted Set Get
Hash Set Get	-	<0,05	<0,05	<0,05
Immutable Hash Set Get	<0,05	-	<0,05	<0,05
Immutable Sorted Set Get	<0,05	<0,05	-	<0,05
Sorted Set Get	<0,05	<0,05	<0,05	-

Table A.98: Table showing the *p*-values for the group Set Get with regards to Elapsed Time.

Package Energy *p*-Values	Hash Set Get	Immutable Hash Set Get	Immutable Sorted Set Get	Sorted Set Get
Hash Set Get	-	<0,05	<0,05	<0,05
Immutable Hash Set Get	<0,05	-	<0,05	<0,05
Immutable Sorted Set Get	<0,05	<0,05	-	<0,05
Sorted Set Get	<0,05	<0,05	<0,05	-

Table A.99: Table showing the *p*-values for the group Set Get with regards to Package Energy.

DRAM Energy *p*-Values	Hash Set Get	Immutable Hash Set Get	Immutable Sorted Set Get	Sorted Set Get
Hash Set Get	-	<0,05	<0,05	<0,05
Immutable Hash Set Get	<0,05	-	<0,05	<0,05
Immutable Sorted Set Get	<0,05	<0,05	-	<0,05
Sorted Set Get	<0,05	<0,05	<0,05	-

Table A.100: Table showing the *p*-values for the group Set Get with regards to DRAM Energy.

Set Copy

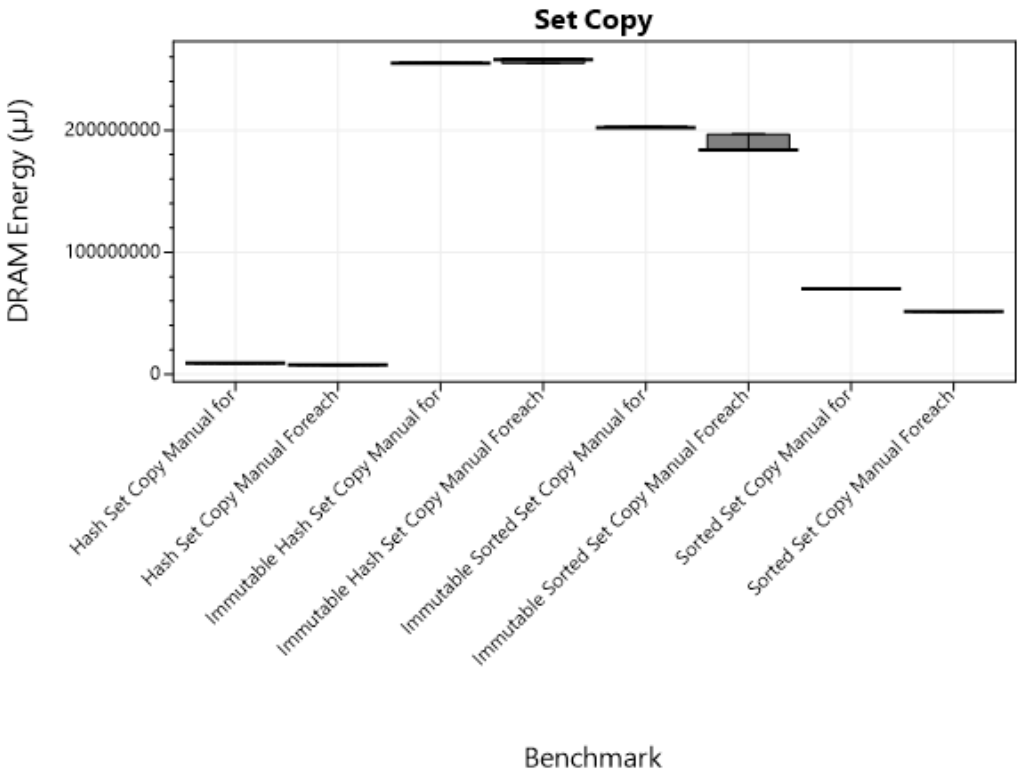

Figure A.80: Boxplot showing how efficient different Set Copy types are with regards to DRAM Energy.

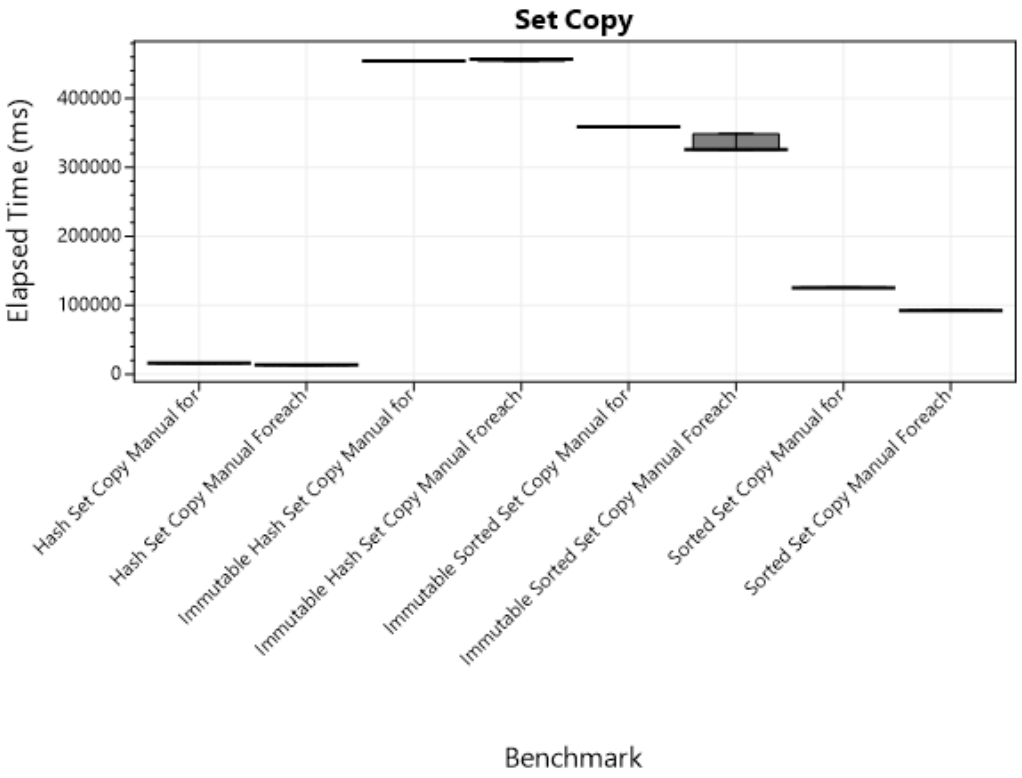

Figure A.81: Boxplot showing how efficient different Set Copy types are with regards to Elapsed Time.

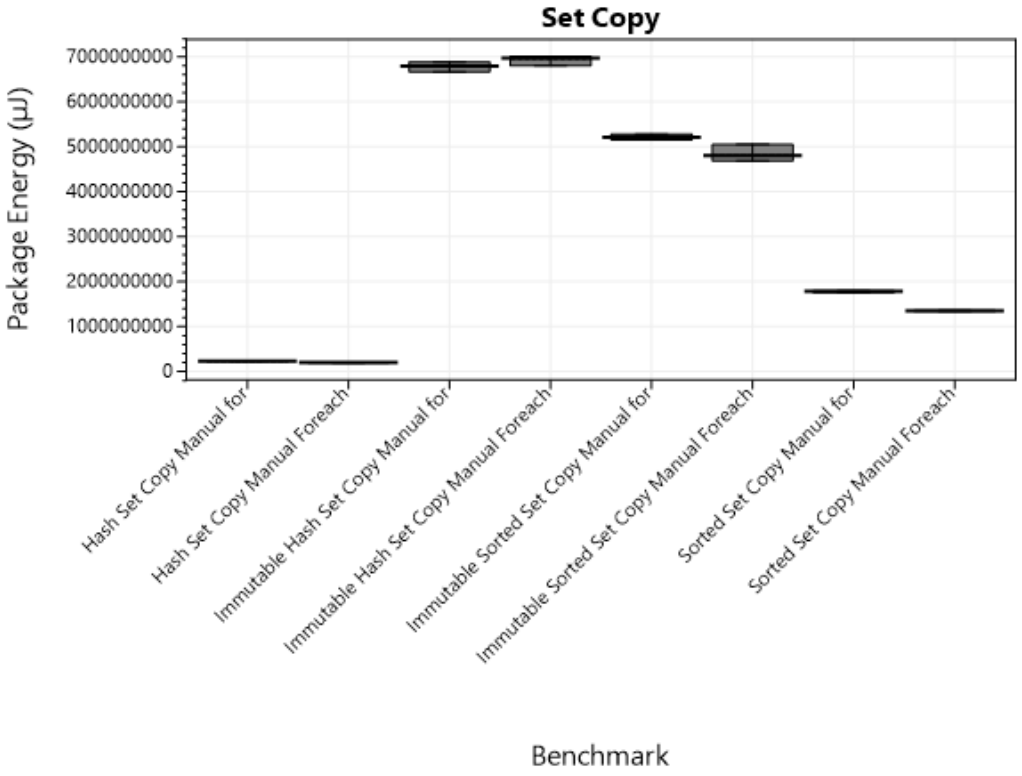

Figure A.82: Boxplot showing how efficient different Set Copy types are with regards to Package Energy.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Hash Set Copy Manual for	16.304,306098	235.434.765,625	9.228.759,766
Hash Set Copy Manual Foreach	13.596,051758	200.385.533,203	7.710.794,922
Immutable Hash Set Copy Manual for	454.078,125000	6.797.356.960,227	255.237.215,909
Immutable Hash Set Copy Manual Foreach	456.778,850446	6.950.067.745,536	257.355.133,929
Immutable Sorted Set Copy Manual for	358.804,756944	5.213.533.420,139	202.210.243,056
Immutable Sorted Set Copy Manual Foreach	326.751,615234	4.807.357.421,875	184.385.292,969
Sorted Set Copy Manual for	125.329,787326	1.784.640.625,000	70.004.817,708
Sorted Set Copy Manual Foreach	92.074,437934	1.346.576.736,111	51.438.671,875

Table A.101: Table showing the elapsed time and energy measurement for each Set Copy.

Elapsed Time *p*-Values	Hash Set Copy Manual Foreach	Hash Set Copy Manual for	Immutable Hash Set Copy Manual Foreach	Immutable Hash Set Copy Manual for	Immutable Sorted Set Copy Manual Foreach	Immutable Sorted Set Copy Manual for	Sorted Set Copy Manual Foreach	Sorted Set Copy Manual for
Hash Set Copy Manual Foreach	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Hash Set Copy Manual for	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Immutable Hash Set Copy Manual Foreach	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Immutable Hash Set Copy Manual for	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Immutable Sorted Set Copy Manual Foreach	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05
Immutable Sorted Set Copy Manual for	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Sorted Set Copy Manual Foreach	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05
Sorted Set Copy Manual for	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-

Table A.102: Table showing the *p*-values for the group Set Copy with regards to Elapsed Time.

Package Energy p-Values	Hash Set Copy Manual Foreach	Hash Set Copy Manual for	Immutable Hash Set Copy Manual Foreach	Immutable Hash Set Copy Manual for	Immutable Sorted Set Copy Manual Foreach	Immutable Sorted Set Copy Manual for	Sorted Set Copy Manual Foreach	Sorted Set Copy Manual for
Hash Set Copy Manual Foreach	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Hash Set Copy Manual for	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Immutable Hash Set Copy Manual Foreach	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Immutable Hash Set Copy Manual for	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Immutable Sorted Set Copy Manual Foreach	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05
Immutable Sorted Set Copy Manual for	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Sorted Set Copy Manual Foreach	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05
Sorted Set Copy Manual for	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-

Table A.103: Table showing the *p*-values for the group Set Copy with regards to Package Energy.

DRAM Energy p-Values	Hash Set Copy Manual Foreach	Hash Set Copy Manual for	Immutable Hash Set Copy Manual Foreach	Immutable Hash Set Copy Manual for	Immutable Sorted Set Copy Manual Foreach	Immutable Sorted Set Copy Manual for	Sorted Set Copy Manual Foreach	Sorted Set Copy Manual for
Hash Set Copy Manual Foreach	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Hash Set Copy Manual for	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Immutable Hash Set Copy Manual Foreach	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Immutable Hash Set Copy Manual for	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Immutable Sorted Set Copy Manual Foreach	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05
Immutable Sorted Set Copy Manual for	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Sorted Set Copy Manual Foreach	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05
Sorted Set Copy Manual for	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-

Table A.104: Table showing the *p*-values for the group Set Copy with regards to DRAM Energy.

Set Insertion

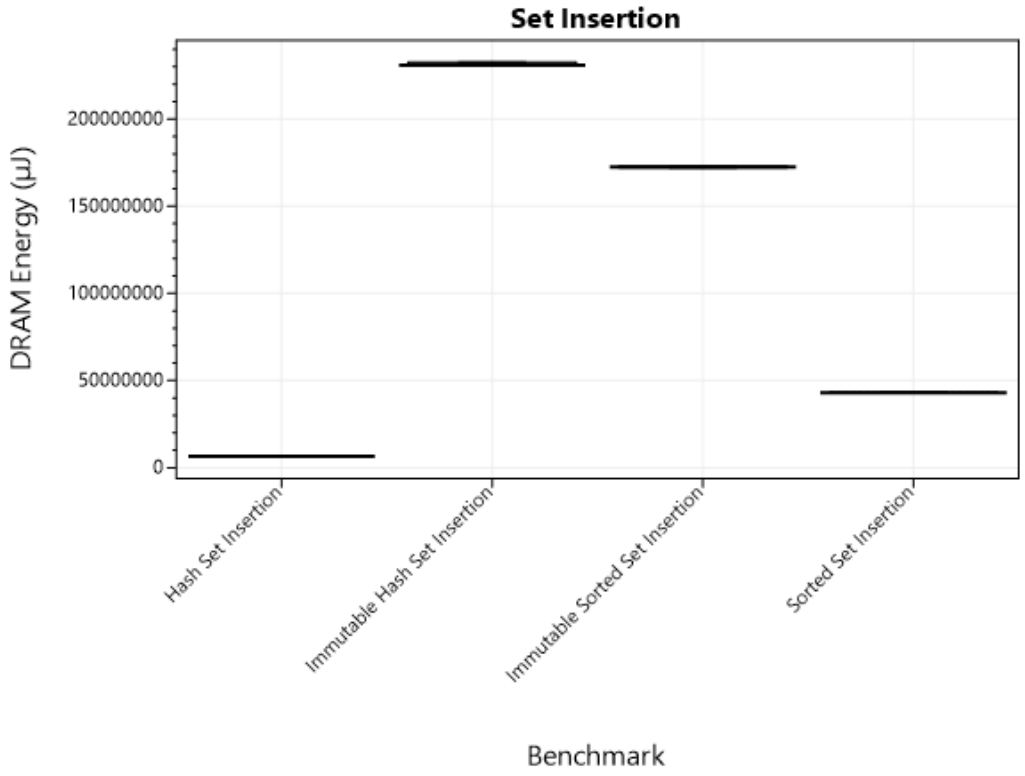

Figure A.83: Boxplot showing how efficient different Set Insertion types are with regards to DRAM Energy.

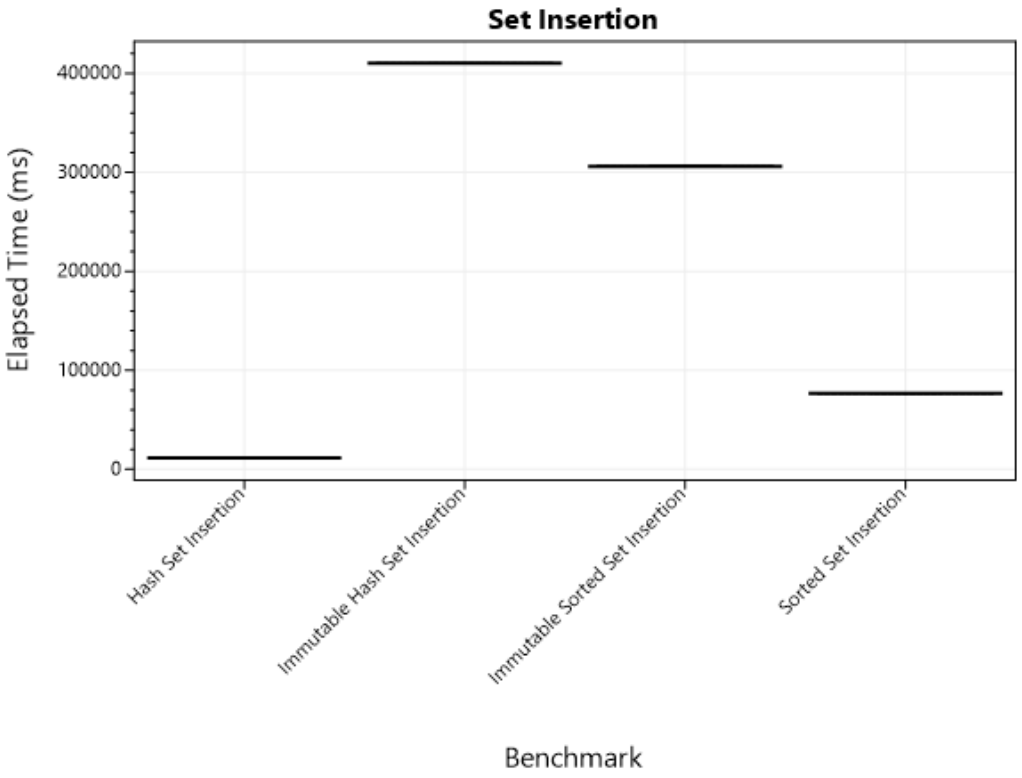

Figure A.84: Boxplot showing how efficient different Set Insertion types are with regards to Elapsed Time.

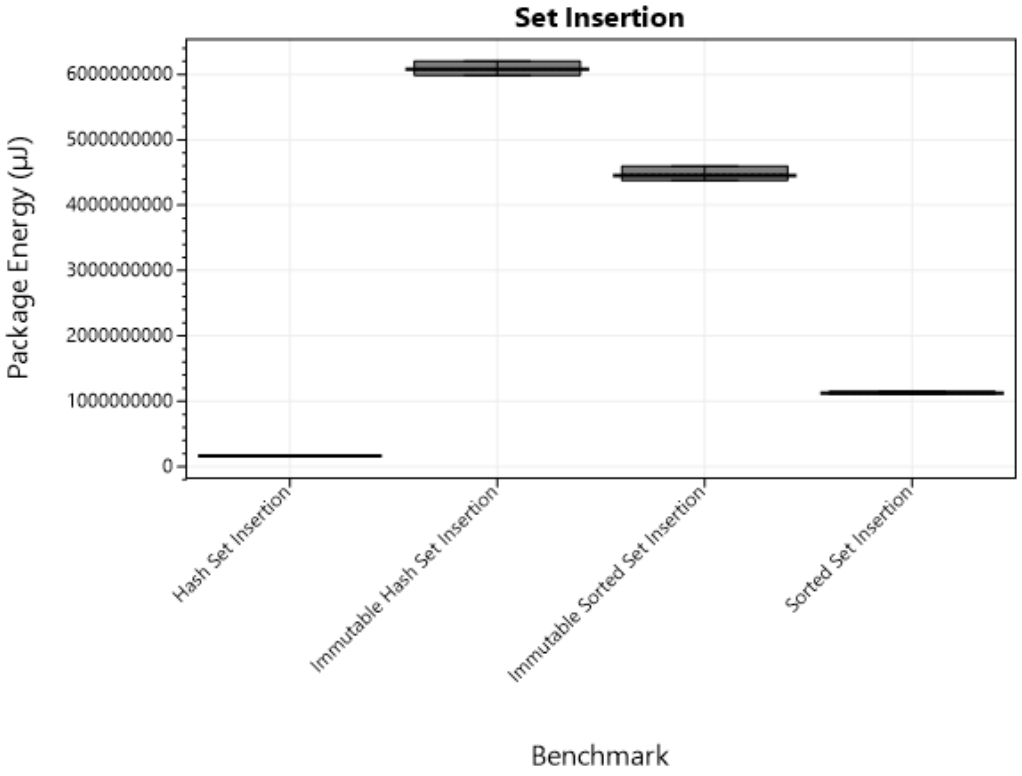

Figure A.85: Boxplot showing how efficient different Set Insertion types are with regards to Package Energy.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Hash Set Insertion	11.356,735026	162.558.336,046	6.452.682,834
Immutable Hash Set Insertion	410.127,760417	6.079.054.315,476	231.174.627,976
Immutable Sorted Set Insertion	306.020,373884	4.465.258.593,750	172.461.356,027
Sorted Set Insertion	76.844,421875	1.125.280.390,625	42.936.699,219

Table A.105: Table showing the elapsed time and energy measurement for each Set Insertion.

Elapsed Time *p*-Values	Hash Set Insertion	Immutable Hash Set Insertion	Immutable Sorted Set Insertion	Sorted Set Insertion
Hash Set Insertion	-	<0,05	<0,05	<0,05
Immutable Hash Set Insertion	<0,05	-	<0,05	<0,05
Immutable Sorted Set Insertion	<0,05	<0,05	-	<0,05
Sorted Set Insertion	<0,05	<0,05	<0,05	-

Table A.106: Table showing the *p*-values for the group Set Insertion with regards to Elapsed Time.

Package Energy *p*-Values	Hash Set Insertion	Immutable Hash Set Insertion	Immutable Sorted Set Insertion	Sorted Set Insertion
Hash Set Insertion	-	<0,05	<0,05	<0,05
Immutable Hash Set Insertion	<0,05	-	<0,05	<0,05
Immutable Sorted Set Insertion	<0,05	<0,05	-	<0,05
Sorted Set Insertion	<0,05	<0,05	<0,05	-

Table A.107: Table showing the *p*-values for the group Set Insertion with regards to Package Energy.

DRAM Energy *p*-Values	Hash Set Insertion	Immutable Hash Set Insertion	Immutable Sorted Set Insertion	Sorted Set Insertion
Hash Set Insertion	-	<0,05	<0,05	<0,05
Immutable Hash Set Insertion	<0,05	-	<0,05	<0,05
Immutable Sorted Set Insertion	<0,05	<0,05	-	<0,05
Sorted Set Insertion	<0,05	<0,05	<0,05	-

Table A.108: Table showing the *p*-values for the group Set Insertion with regards to DRAM Energy.

Set Removal

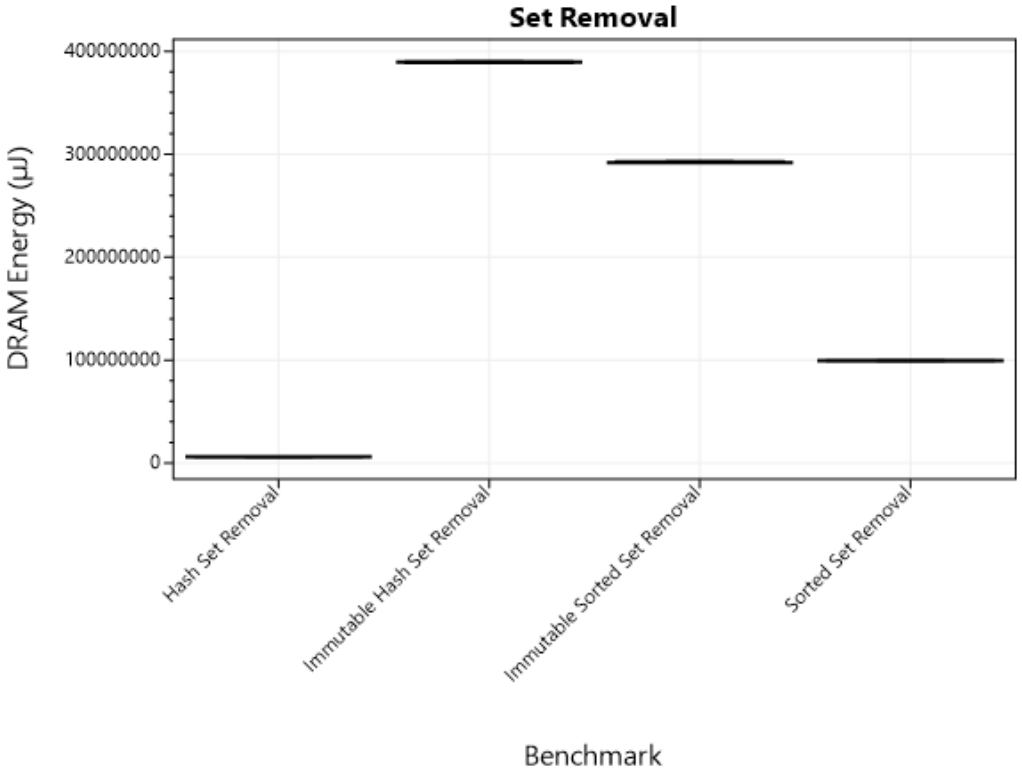

Figure A.86: Boxplot showing how efficient different Set Removal types are with regards to DRAM Energy.

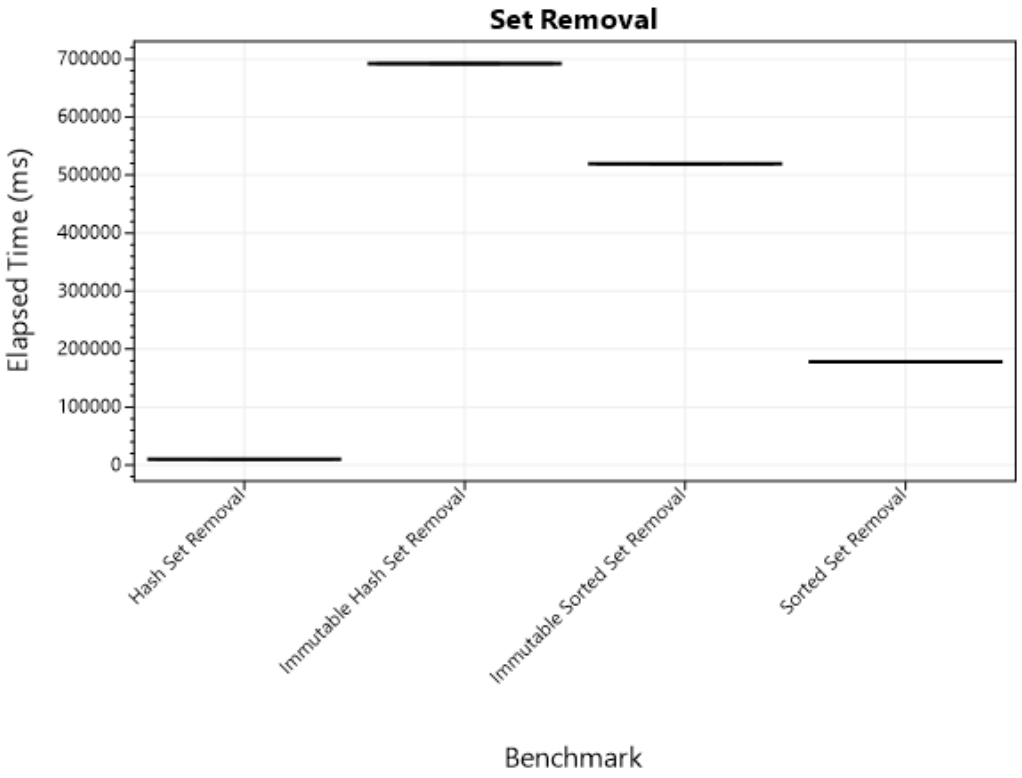

Figure A.87: Boxplot showing how efficient different Set Removal types are with regards to Elapsed Time.

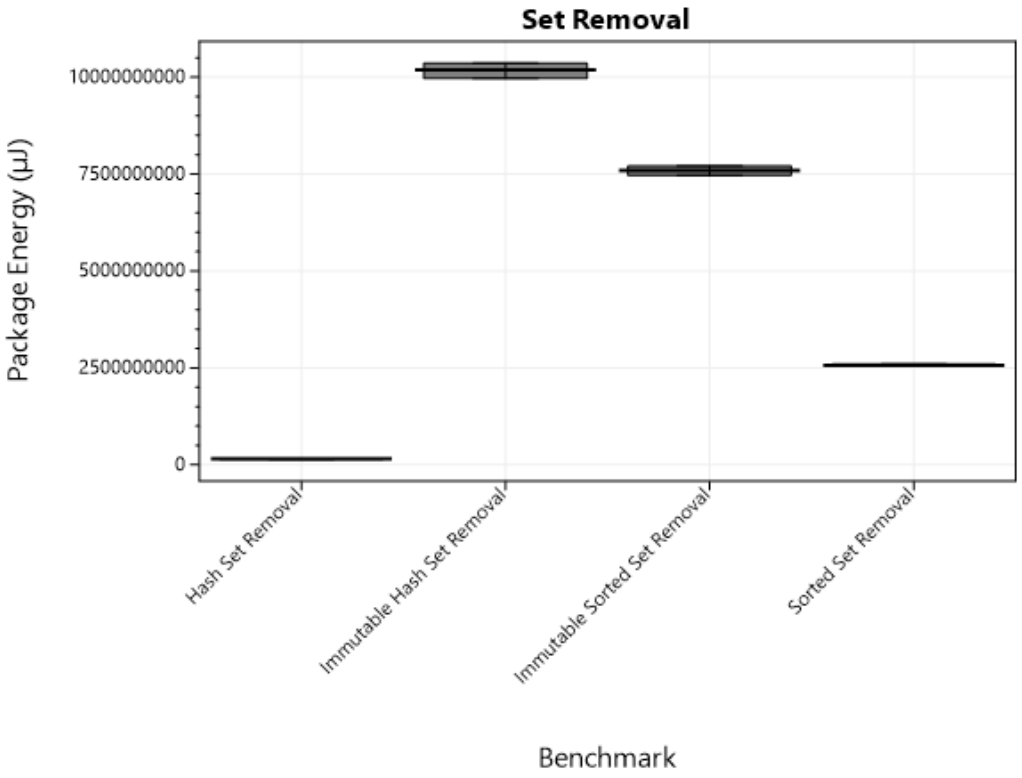

Figure A.88: Boxplot showing how efficient different Set Removal types are with regards to Package Energy.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Hash Set Removal	10.561,858798	149.928.328,081	5.875.310,724
Immutable Hash Set Removal	692.034,578125	10.175.922.031,250	389.341.328,125
Immutable Sorted Set Removal	519.039,176136	7.591.739.772,727	291.988.210,227
Sorted Set Removal	178.111,510417	2.571.655.642,361	99.340.798,611

Table A.109: Table showing the elapsed time and energy measurement for each Set Removal.

Elapsed Time *p*-Values	Hash Set Removal	Immutable Hash Set Removal	Immutable Sorted Set Removal	Sorted Set Removal
Hash Set Removal	-	<0,05	<0,05	<0,05
Immutable Hash Set Removal	<0,05	-	<0,05	<0,05
Immutable Sorted Set Removal	<0,05	<0,05	-	<0,05
Sorted Set Removal	<0,05	<0,05	<0,05	-

Table A.110: Table showing the *p*-values for the group Set Removal with regards to Elapsed Time.

Package Energy *p*-Values	Hash Set Removal	Immutable Hash Set Removal	Immutable Sorted Set Removal	Sorted Set Removal
Hash Set Removal	-	<0,05	<0,05	<0,05
Immutable Hash Set Removal	<0,05	-	<0,05	<0,05
Immutable Sorted Set Removal	<0,05	<0,05	-	<0,05
Sorted Set Removal	<0,05	<0,05	<0,05	-

Table A.111: Table showing the *p*-values for the group Set Removal with regards to Package Energy.

DRAM Energy *p*-Values	Hash Set Removal	Immutable Hash Set Removal	Immutable Sorted Set Removal	Sorted Set Removal
Hash Set Removal	-	<0,05	<0,05	<0,05
Immutable Hash Set Removal	<0,05	-	<0,05	<0,05
Immutable Sorted Set Removal	<0,05	<0,05	-	<0,05
Sorted Set Removal	<0,05	<0,05	<0,05	-

Table A.112: Table showing the *p*-values for the group Set Removal with regards to DRAM Energy.

Table Creation

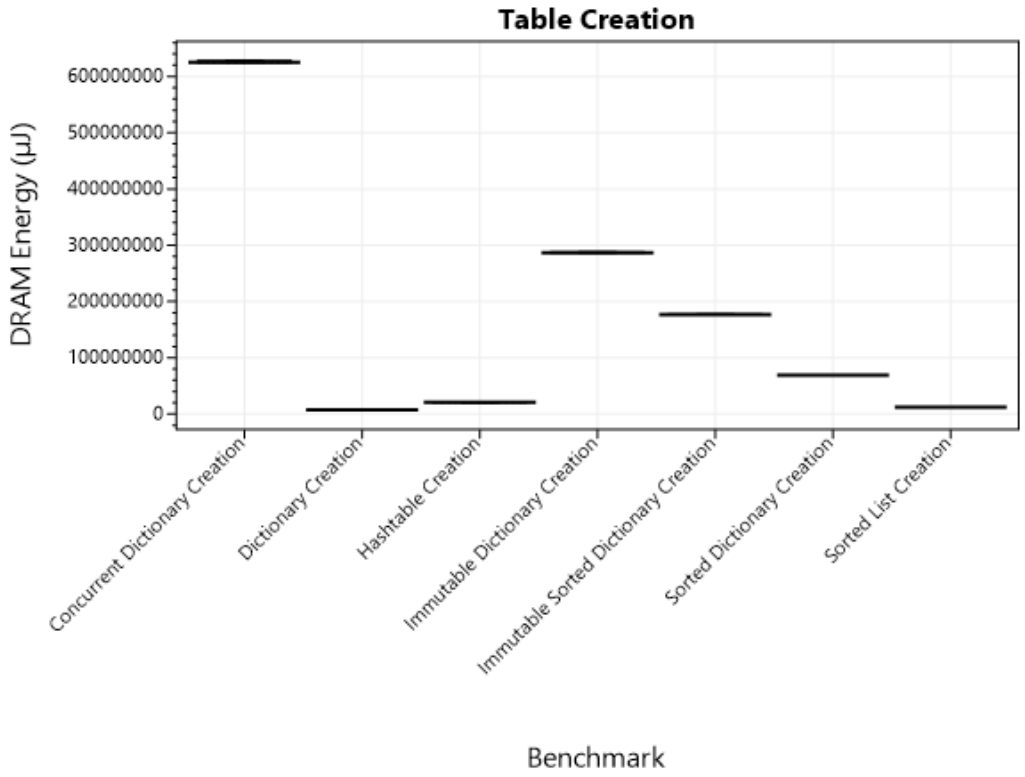

Figure A.89: Boxplot showing how efficient different Table Creation types are with regards to DRAM Energy.

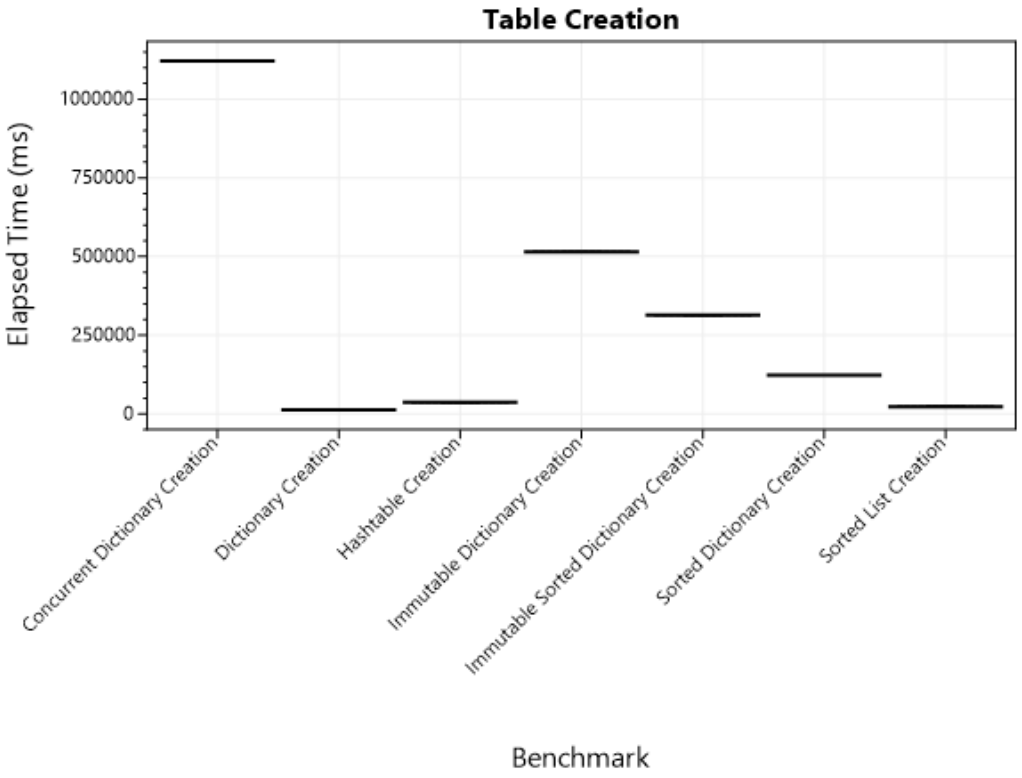

Figure A.90: Boxplot showing how efficient different Table Creation types are with regards to Elapsed Time.

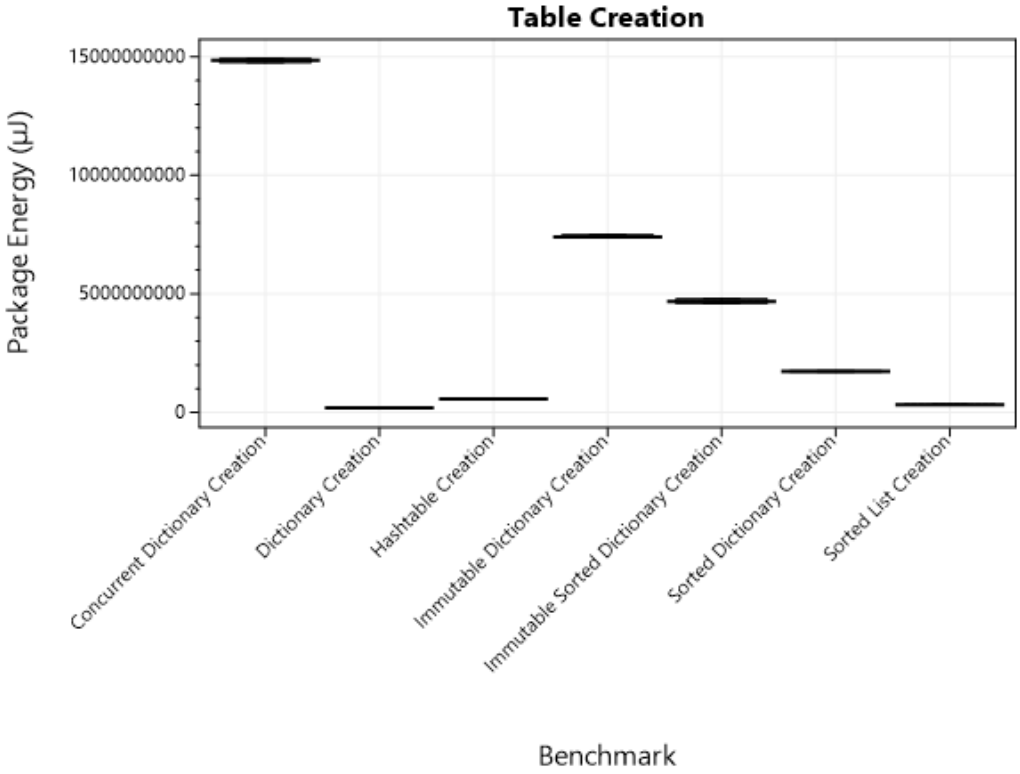

Figure A.91: Boxplot showing how efficient different Table Creation types are with regards to Package Energy.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Concurrent Dictionary Creation	1.121.929,791667	14.845.217.013,889	625.312.152,778
Dictionary Creation	12.564,241536	187.864.089,627	7.212.147,352
Hashtable Creation	37.045,715061	569.785.177,951	21.063.563,368
Immutable Dictionary Creation	514.113,767361	7.400.769.097,222	287.086.979,167
Immutable Sorted Dictionary Creation	313.663,977273	4.679.674.396,307	176.665.553,977
Sorted Dictionary Creation	123.507,240513	1.730.808.872,768	69.004.966,518
Sorted List Creation	22.188,491753	319.061.187,066	12.399.723,307

Table A.113: Table showing the elapsed time and energy measurement for each Table Creation.

Elapsed Time *p*-Values	Concurrent Dictionary Creation	Dictionary Creation	Hashtable Creation	Immutable Dictionary Creation	Immutable Sorted Dictionary Creation	Sorted Dictionary Creation	Sorted List Creation
Concurrent Dictionary Creation	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Dictionary Creation	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Hashtable Creation	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Immutable Dictionary Creation	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05
Immutable Sorted Dictionary Creation	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Sorted Dictionary Creation	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05
Sorted List Creation	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-

Table A.114: Table showing the *p*-values for the group Table Creation with regards to Elapsed Time.

Package Energy *p*-Values	Concurrent Dictionary Creation	Dictionary Creation	Hashtable Creation	Immutable Dictionary Creation	Immutable Sorted Dictionary Creation	Sorted Dictionary Creation	Sorted List Creation
Concurrent Dictionary Creation	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Dictionary Creation	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Hashtable Creation	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Immutable Dictionary Creation	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05
Immutable Sorted Dictionary Creation	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Sorted Dictionary Creation	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05
Sorted List Creation	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-

Table A.115: Table showing the *p*-values for the group Table Creation with regards to Package Energy.

DRAM Energy *p*-Values	Concurrent Dictionary Creation	Dictionary Creation	Hashtable Creation	Immutable Dictionary Creation	Immutable Sorted Dictionary Creation	Sorted Dictionary Creation	Sorted List Creation
Concurrent Dictionary Creation	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Dictionary Creation	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Hashtable Creation	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Immutable Dictionary Creation	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05
Immutable Sorted Dictionary Creation	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Sorted Dictionary Creation	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05
Sorted List Creation	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-

Table A.116: Table showing the *p*-values for the group Table Creation with regards to DRAM Energy.

Table Get

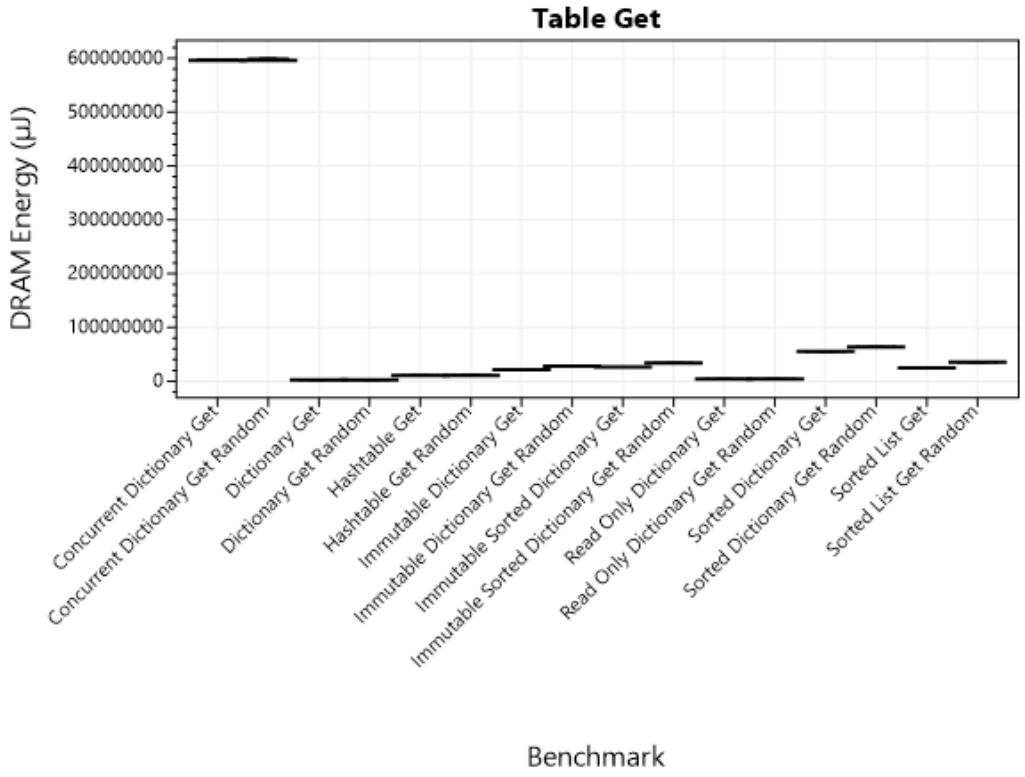

Figure A.92: Boxplot showing how efficient different Table Get types are with regards to DRAM Energy.

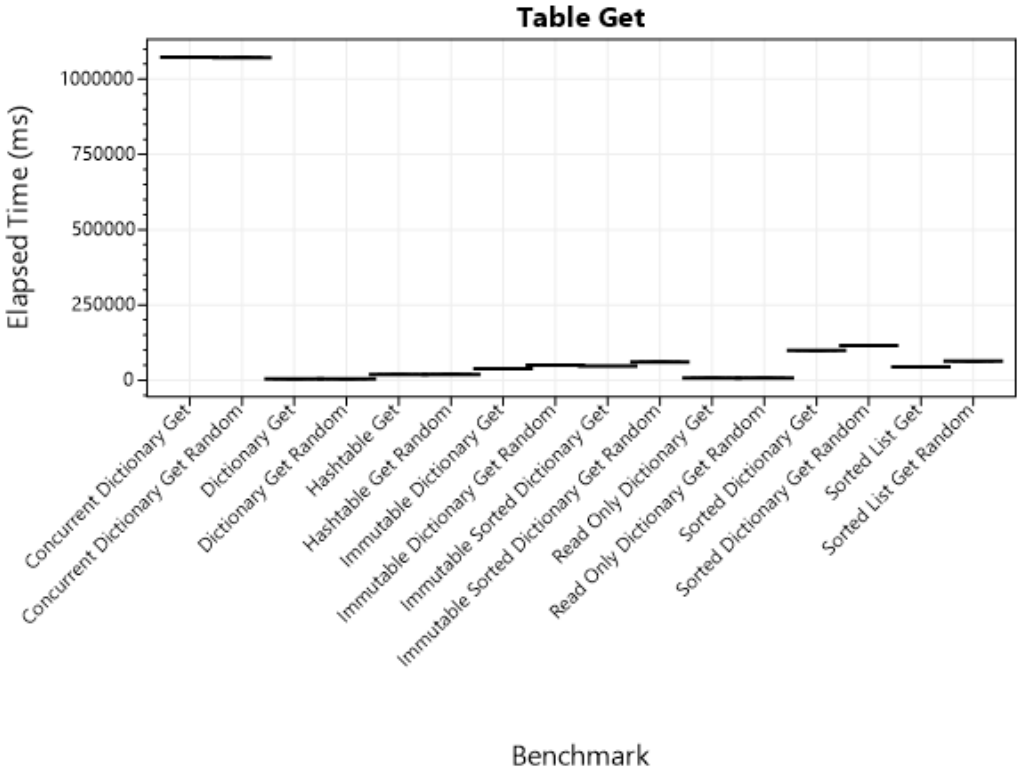

Figure A.93: Boxplot showing how efficient different Table Get types are with regards to Elapsed Time.

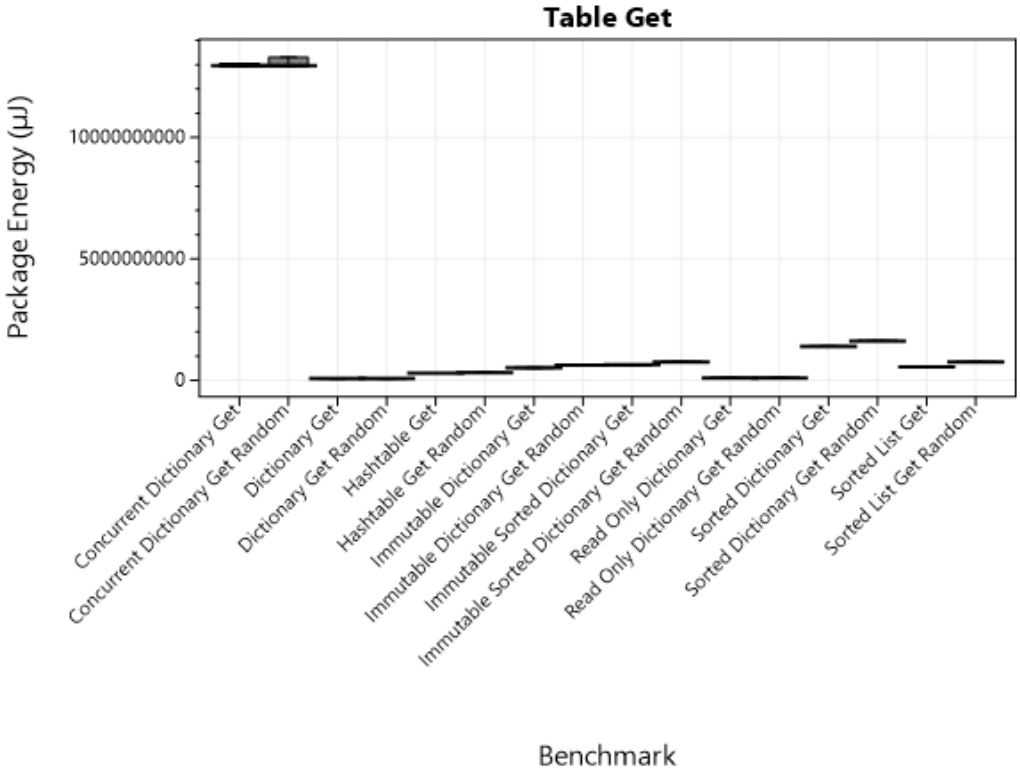

Figure A.94: Boxplot showing how efficient different Table Get types are with regards to Package Energy.

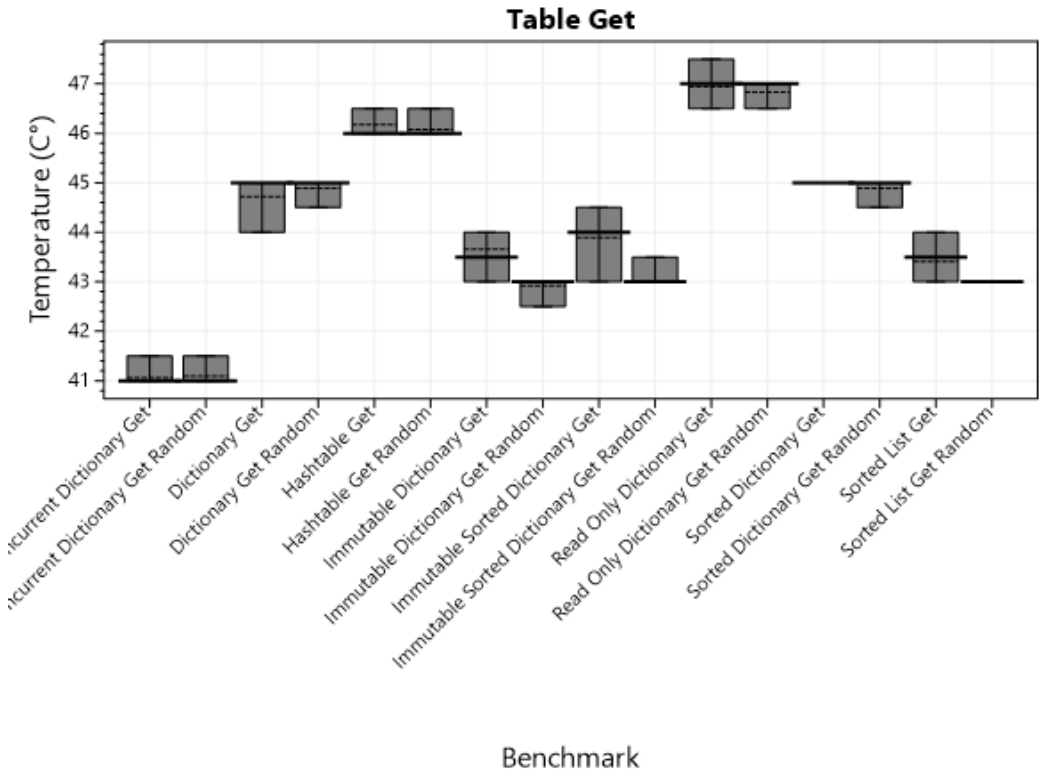

Figure A.95: Boxplot showing how efficient different Table Get types are with regards to Temperature.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Concurrent Dictionary Get	1.072.901,458333	12.962.393.055,556	596.977.430,556
Concurrent Dictionary Get Random	1.072.031,818182	12.994.715.909,091	596.755.681,818
Dictionary Get	4.843,913710	69.332.938,639	2.695.199,924
Dictionary Get Random	4.844,832628	69.141.790,093	2.695.448,134
Hashtable Get	18.620,078823	300.895.396,205	10.472.509,766
Hashtable Get Random	19.502,656250	315.457.230,319	10.948.655,349
Immutable Dictionary Get	37.928,138428	512.774.717,204	21.094.523,112
Immutable Dictionary Get Random	50.475,181588	622.144.283,150	28.075.000,000
Immutable Sorted Dictionary Get	47.234,032411	645.217.337,943	26.275.274,493
Immutable Sorted Dictionary Get Random	60.901,821181	752.983.029,514	33.865.416,667
Read Only Dictionary Get	6.952,353516	104.625.971,680	3.865.769,531
Read Only Dictionary Get Random	6.937,612305	104.983.878,581	3.857.071,940
Sorted Dictionary Get	99.192,005208	1.403.638.151,042	55.151.388,889
Sorted Dictionary Get Random	114.615,112847	1.626.500.651,042	63.774.175,347
Sorted List Get	44.417,494081	552.496.978,575	24.705.158,026
Sorted List Get Random	63.285,692274	757.550.585,937	35.190.516,493

Table A.117: Table showing the elapsed time and energy measurement for each Table Get.

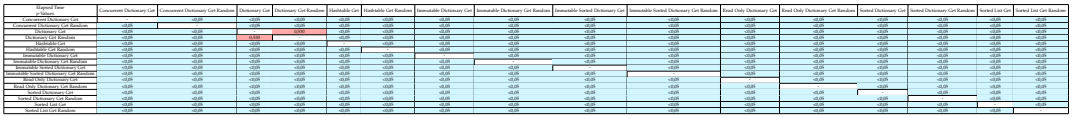

Table A.118: Table showing the *p*-values for the group Table Get with regards to Elapsed Time.

Package Energy *p*-Values	Concurrent Dictionary Get	Concurrent Dictionary Get Random	Dictionary Get	Dictionary Get Random
Concurrent Dictionary Get	-	0,441	<0,05	<0,05
Concurrent Dictionary Get Random	0,441	-	<0,05	<0,05
Dictionary Get	<0,05	<0,05	-	0,405
Dictionary Get Random	<0,05	<0,05	0,405	-
Hashtable Get	<0,05	<0,05	<0,05	<0,05
Hashtable Get Random	<0,05	<0,05	<0,05	<0,05
Immutable Dictionary Get	<0,05	<0,05	<0,05	<0,055
Immutable Dictionary Get Random	<0,05	<0,05	<0,05	<0,05
Immutable Sorted Dictionary Get	<0,05	<0,05	<0,05	<0,05
Immutable Sorted Dictionary Get Random	<0,05	<0,05	<0,05	<0,05
Read Only Dictionary Get	<0,05	<0,05	<0,05	<0,05
Read Only Dictionary Get Random	<0,05	<0,05	<0,05	<0,05
Sorted Dictionary Get	<0,05	<0,05	<0,05	<0,05
Sorted Dictionary Get Random	<0,05	<0,05	<0,05	<0,05
Sorted List Get	<0,05	<0,05	<0,05	<0,05
Sorted List Get Random	<0,05	<0,05	<0,05	<0,05

Package Energy *p*-Values	Hashtable Get	Hashtable Get Random	Immutable Dictionary Get	Immutable Dictionary Get Random
Concurrent Dictionary Get	<0,05	<0,05	<0,05	<0,05
Concurrent Dictionary Get Random	<0,05	<0,05	<0,05	<0,05
Dictionary Get	<0,05	<0,05	<0,05	<0,05
Dictionary Get Random	<0,05	<0,05	<0,05	<0,05
Hashtable Get	-	<0,05	<0,05	<0,05
Hashtable Get Random	<0,05	-	<0,05	<0,05
Immutable Dictionary Get	<0,05	<0,05	-	<0,05
Immutable Dictionary Get Random	<0,05	<0,05	<0,05	-
Immutable Sorted Dictionary Get	<0,05	<0,05	<0,05	<0,05
Immutable Sorted Dictionary Get Random	<0,05	<0,05	<0,05	<0,05
Read Only Dictionary Get	<0,05	<0,05	<0,05	<0,05
Read Only Dictionary Get Random	<0,05	<0,05	<0,05	<0,05
Sorted Dictionary Get	<0,05	<0,05	<0,05	<0,05
Sorted Dictionary Get Random	<0,05	<0,05	<0,05	<0,05
Sorted List Get	<0,05	<0,05	<0,05	<0,05
Sorted List Get Random	<0,05	<0,05	<0,05	<0,05

Package Energy *p*-Values	Immutable Sorted Dictionary Get	Immutable Sorted Dictionary Get Random	Read Only Dictionary Get	Read Only Dictionary Get Random
Concurrent Dictionary Get	<0,05	<0,05	<0,05	<0,05
Concurrent Dictionary Get Random	<0,05	<0,05	<0,05	<0,05
Dictionary Get	<0,05	<0,05	<0,05	<0,05
Dictionary Get Random	<0,05	<0,05	<0,05	<0,05
Hashtable Get	<0,05	<0,05	<0,05	<0,05
Hashtable Get Random	<0,05	<0,05	<0,05	<0,05
Immutable Dictionary Get	<0,05	<0,05	<0,05	<0,05
Immutable Dictionary Get Random	<0,05	<0,05	<0,05	<0,05
Immutable Sorted Dictionary Get	-	<0,05	<0,05	<0,05
Immutable Sorted Dictionary Get Random	<0,05	-	<0,05	<0,05
Read Only Dictionary Get	<0,05	<0,05	-	0,418
Read Only Dictionary Get Random	<0,05	<0,05	0,418	-
Sorted Dictionary Get	<0,05	<0,05	<0,05	<0,05
Sorted Dictionary Get Random	<0,05	<0,05	<0,05	<0,05
Sorted List Get	<0,05	<0,05	<0,05	<0,0
Sorted List Get Random	<0,05	0,296	<0,05	<0,05

Package Energy *p*-Values	Sorted Dictionary Get	Sorted Dictionary Get Random	Sorted List Get	Sorted List Get Random
Concurrent Dictionary Get	<0,05	<0,05	<0,05	<0,05
Concurrent Dictionary Get Random	<0,05	<0,05	<0,05	<0,05
Dictionary Get	<0,05	<0,05	<0,05	<0,05
Dictionary Get Random	<0,05	<0,05	<0,05	<0,05
Hashtable Get	<0,05	<0,05	<0,05	<0,05
Hashtable Get Random	<0,05	<0,05	<0,05	<0,05
Immutable Dictionary Get	<0,05	<0,05	<0,05	<0,05
Immutable Dictionary Get Random	<0,05	<0,05	<0,05	<0,05
Immutable Sorted Dictionary Get	<0,05	<0,05	<0,05	<0,05
Immutable Sorted Dictionary Get Random	<0,05	<0,05	<0,05	0,296
Read Only Dictionary Get	<0,05	<0,05	<0,05	<0,05
Read Only Dictionary Get Random	<0,05	<0,05	<0,05	<0,05
Sorted Dictionary Get	-	<0,05	<0,05	<0,05
Sorted Dictionary Get Random	<0,05	-	<0,05	<0,05
Sorted List Get	<0,05	<0,05	-	<0,05
Sorted List Get Random	<0,05	<0,05	<0,05	-

Table A.119: Tables showing the *p*-values for the group Table Get with regards to Package Energy.

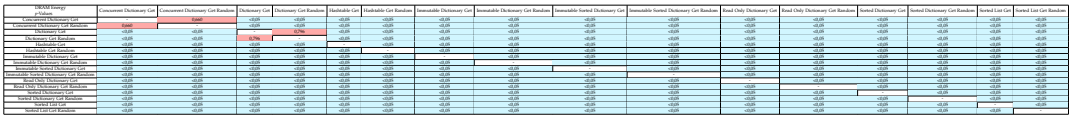

Table A.120: Table showing the *p*-values for the group Table Get with regards to DRAM Energy.

Table Copy

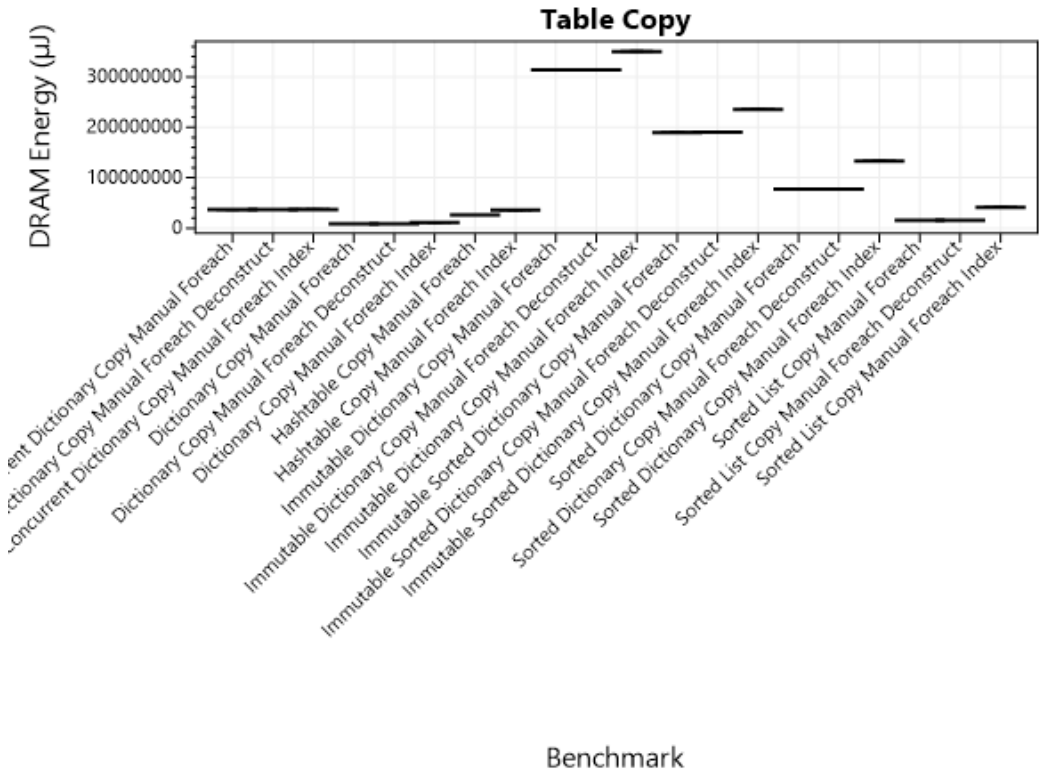

Figure A.96: Boxplot showing how efficient different Table Copy types are with regards to DRAM Energy.

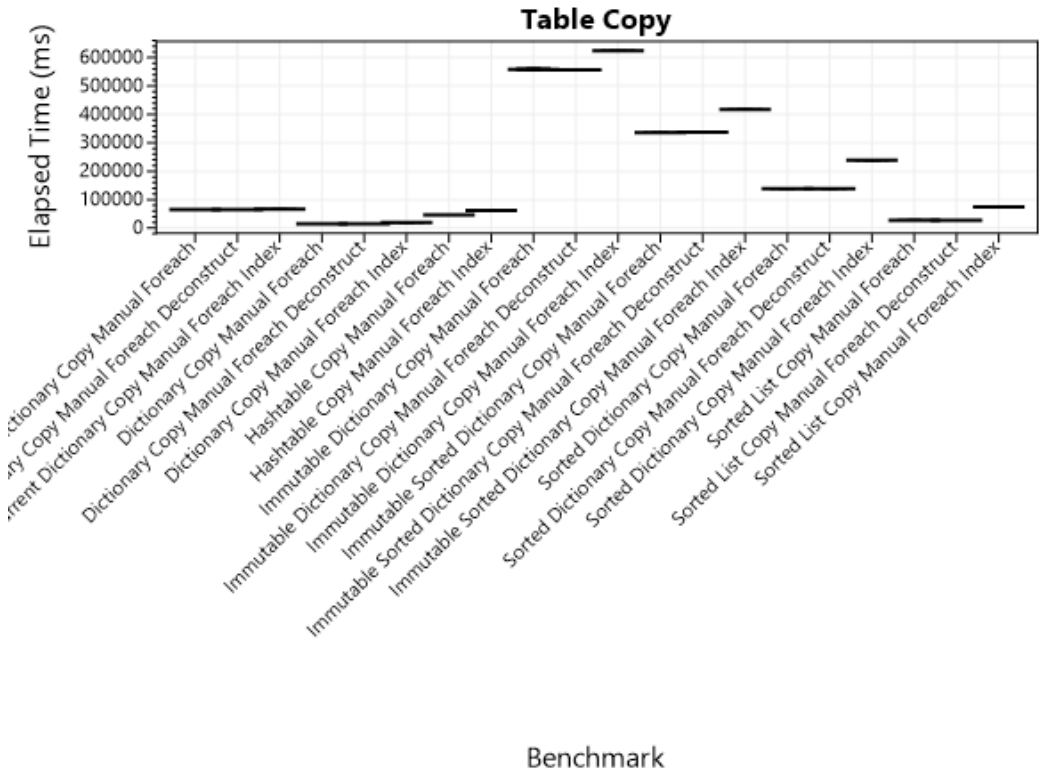

Figure A.97: Boxplot showing how efficient different Table Copy types are with regards to Elapsed Time.

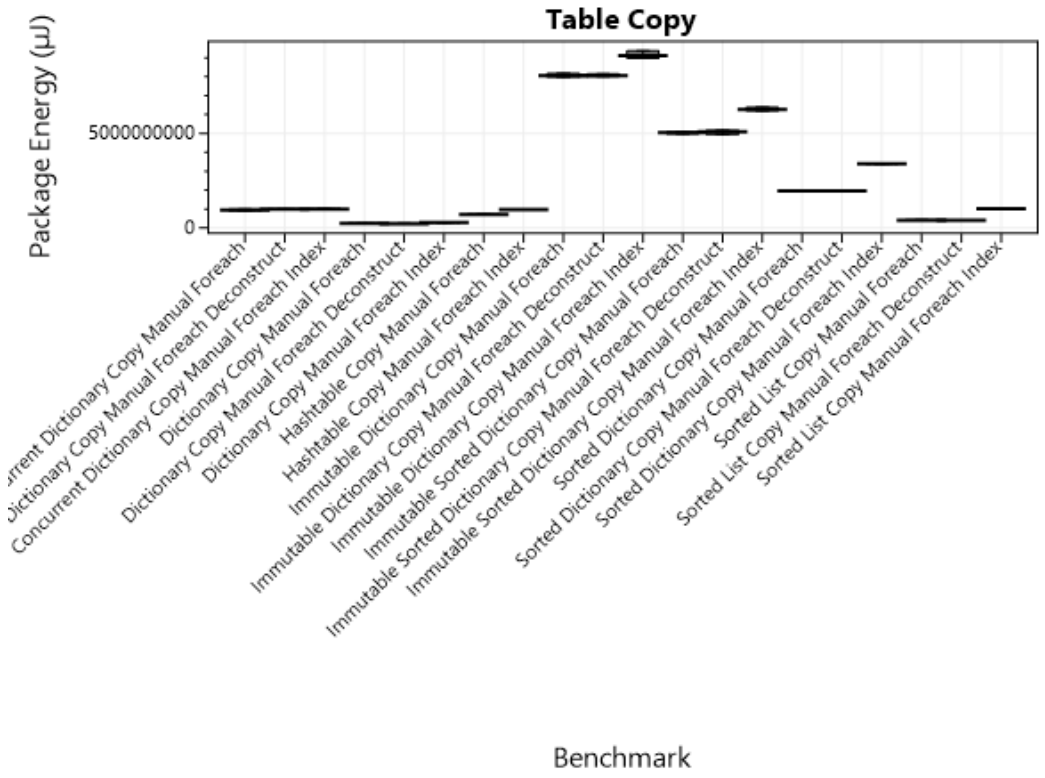

Figure A.98: Boxplot showing how efficient different Table Copy types are with regards to Package Energy.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Concurrent Dictionary Copy Manual Foreach	64.632,826451	950.506.281,888	36.362.392,379
Concurrent Dictionary Copy Manual Foreach Deconstruct	64.955,544705	985.214.561,632	36.577.300,347
Concurrent Dictionary Copy Manual Foreach Index	65.882,224392	1.001.522.352,431	37.137.586,806
Dictionary Copy Manual Foreach	15.162,429504	229.613.082,886	8.668.353,271
Dictionary Copy Manual Foreach Deconstruct	15.129,150879	226.082.670,898	8.609.121,094
Dictionary Copy Manual Foreach Index	19.734,663086	288.810.259,332	11.198.361,545
Hashtable Copy Manual Foreach	45.869,966363	703.693.945,312	25.986.230,469
Hashtable Copy Manual Foreach Index	62.502,873264	967.878.927,951	35.407.725,694
Immutable Dictionary Copy Manual Foreach	558.980,265625	8.060.692.187,500	313.624.687,500
Immutable Dictionary Copy Manual Foreach Deconstruct	557.388,177083	8.052.911.284,722	313.397.743,056
Immutable Dictionary Copy Manual Foreach Index	623.691,847426	9.109.288.694,853	350.446.507,353
Immutable Sorted Dictionary Copy Manual Foreach	336.119,991319	5.031.917.187,500	189.642.968,750
Immutable Sorted Dictionary Copy Manual Foreach Deconstruct	337.114,570312	5.071.705.078,125	190.196.839,489
Immutable Sorted Dictionary Copy Manual Foreach Index	417.556,015625	6.286.748.102,679	235.209.598,214
Sorted Dictionary Copy Manual Foreach	137.926,250000	1.956.346.571,181	77.040.755,208
Sorted Dictionary Copy Manual Foreach Deconstruct	138.039,960938	1.956.338.628,472	77.091.102,431
Sorted Dictionary Copy Manual Foreach Index	238.277,387153	3.395.018.836,806	132.926.041,667
Sorted List Copy Manual Foreach	28.094,923828	407.233.383,789	15.694.760,742
Sorted List Copy Manual Foreach Deconstruct	28.092,382813	401.166.037,326	15.709.092,882
Sorted List Copy Manual Foreach Index	74.134,964718	1.019.207.982,611	41.331.294,103

Table A.121: Table showing the elapsed time and energy measurement for each Table Copy.

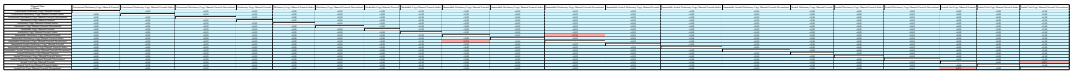

Table A.122: Table showing the *p*-values for the group Table Copy with regards to Elapsed Time.

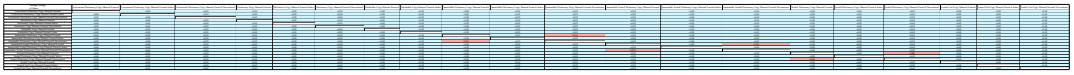

Table A.123: Table showing the *p*-values for the group Table Copy with regards to Package Energy.

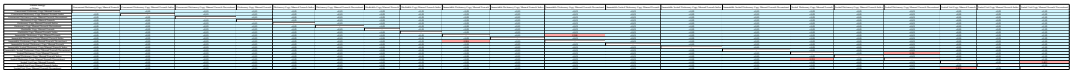

Table A.124: Table showing the *p*-values for the group Table Copy with regards to DRAM Energy.

Table Insertion

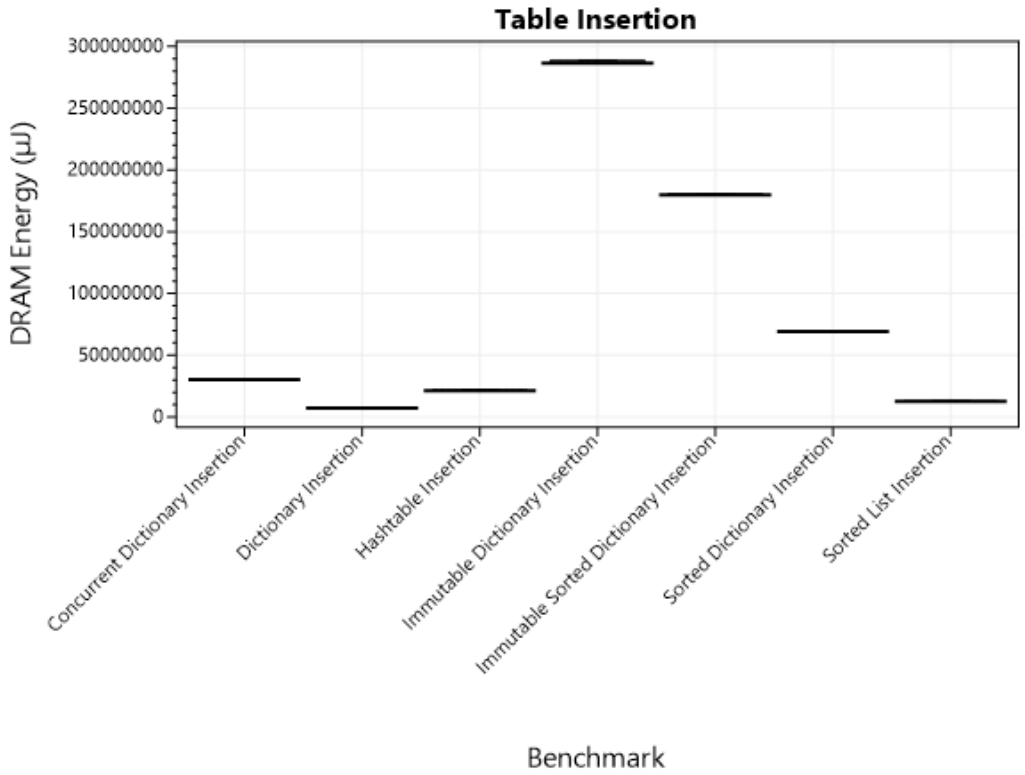

Figure A.99: Boxplot showing how efficient different Table Insertion types are with regards to DRAM Energy.

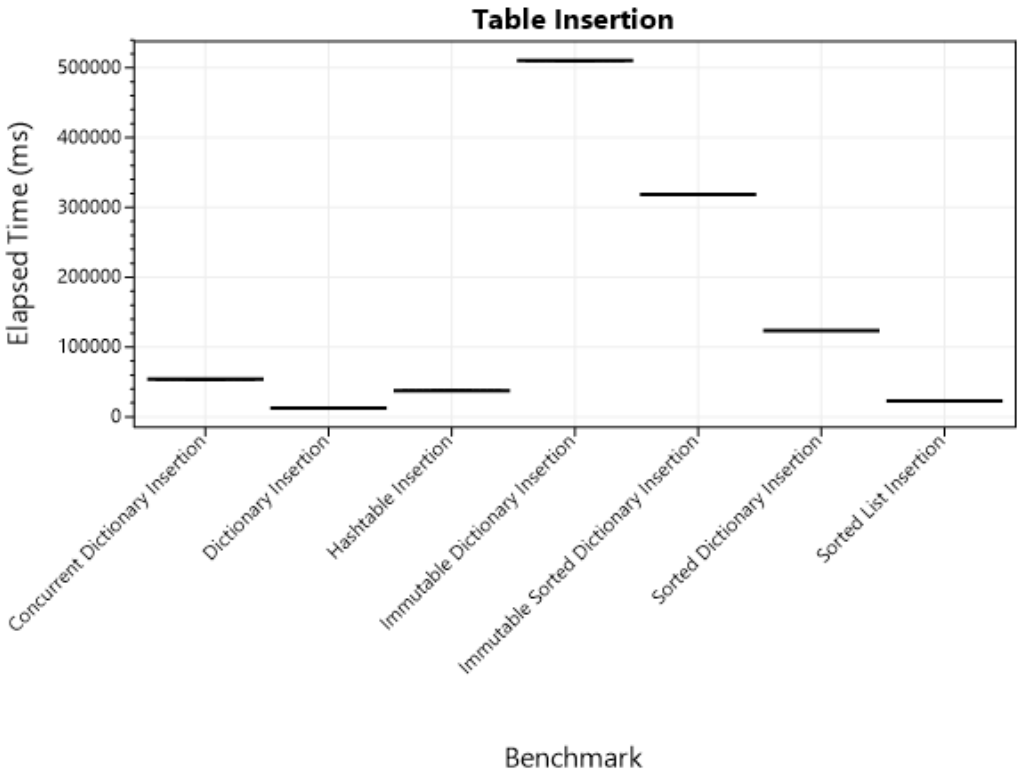

Figure A.100: Boxplot showing how efficient different Table Insertion types are with regards to Elapsed Time.

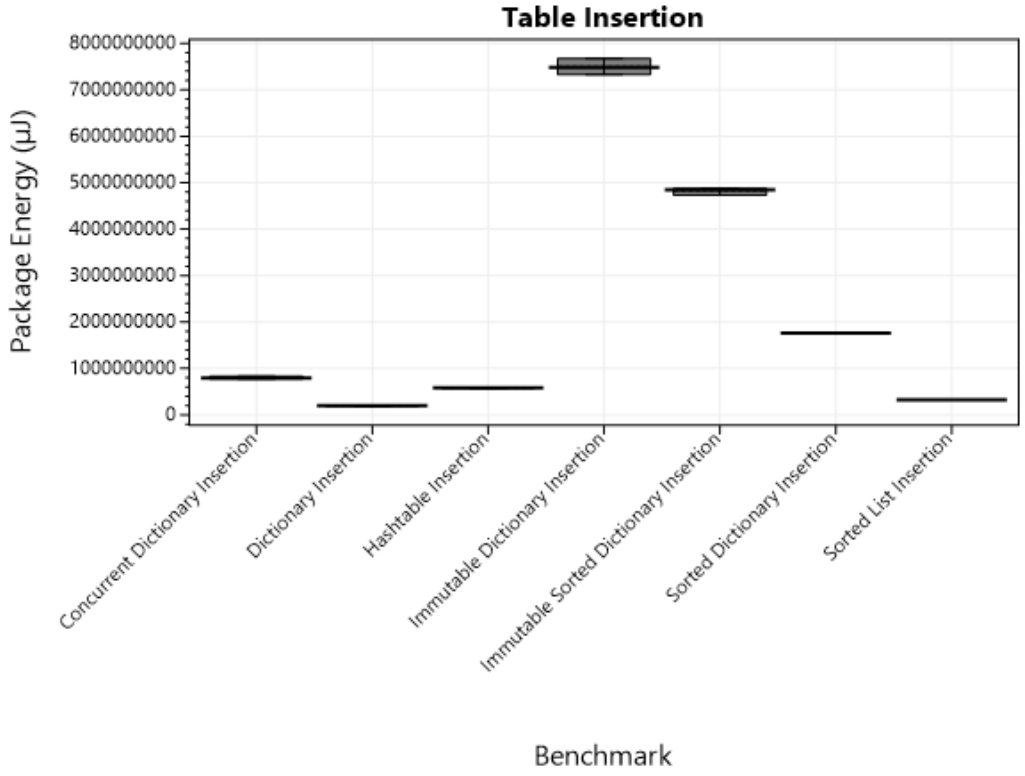

Figure A.101: Boxplot showing how efficient different Table Insertion types are with regards to Package Energy.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Concurrent Dictionary Insertion	53.874,248311	796.316.337,627	30.210.773,860
Dictionary Insertion	12.910,111762	194.847.086,589	7.388.639,323
Hashtable Insertion	37.719,679905	577.019.184,028	21.463.563,368
Immutable Dictionary Insertion	510.088,476563	7.495.426.953,125	286.740.390,625
Immutable Sorted Dictionary Insertion	318.472,486979	4.826.910.872,396	179.818.815,104
Sorted Dictionary Insertion	123.652,565104	1.761.876.996,528	69.053.776,042
Sorted List Insertion	22.492,137044	324.142.144,097	12.582.839,627

Table A.125: Table showing the elapsed time and energy measurement for each Table Insertion.

Elapsed Time *p*-Values	Concurrent Dictionary Insertion	Dictionary Insertion	Hashtable Insertion	Immutable Dictionary Insertion	Immutable Sorted Dictionary Insertion	Sorted Dictionary Insertion	Sorted List Insertion
Concurrent Dictionary Insertion	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Dictionary Insertion	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Hashtable Insertion	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Immutable Dictionary Insertion	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05
Immutable Sorted Dictionary Insertion	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Sorted Dictionary Insertion	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05
Sorted List Insertion	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-

Table A.126: Table showing the *p*-values for the group Table Insertion with regards to Elapsed Time.

Package Energy *p*-Values	Concurrent Dictionary Insertion	Dictionary Insertion	Hashtable Insertion	Immutable Dictionary Insertion	Immutable Sorted Dictionary Insertion	Sorted Dictionary Insertion	Sorted List Insertion
Concurrent Dictionary Insertion	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Dictionary Insertion	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Hashtable Insertion	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Immutable Dictionary Insertion	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05
Immutable Sorted Dictionary Insertion	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Sorted Dictionary Insertion	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05
Sorted List Insertion	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-

Table A.127: Table showing the *p*-values for the group Table Insertion with regards to Package Energy.

DRAM Energy *p*-Values	Concurrent Dictionary Insertion	Dictionary Insertion	Hashtable Insertion	Immutable Dictionary Insertion	Immutable Sorted Dictionary Insertion	Sorted Dictionary Insertion	Sorted List Insertion
Concurrent Dictionary Insertion	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Dictionary Insertion	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Hashtable Insertion	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Immutable Dictionary Insertion	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05
Immutable Sorted Dictionary Insertion	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Sorted Dictionary Insertion	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05
Sorted List Insertion	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-

Table A.128: Table showing the *p*-values for the group Table Insertion with regards to DRAM Energy.

Table Removal

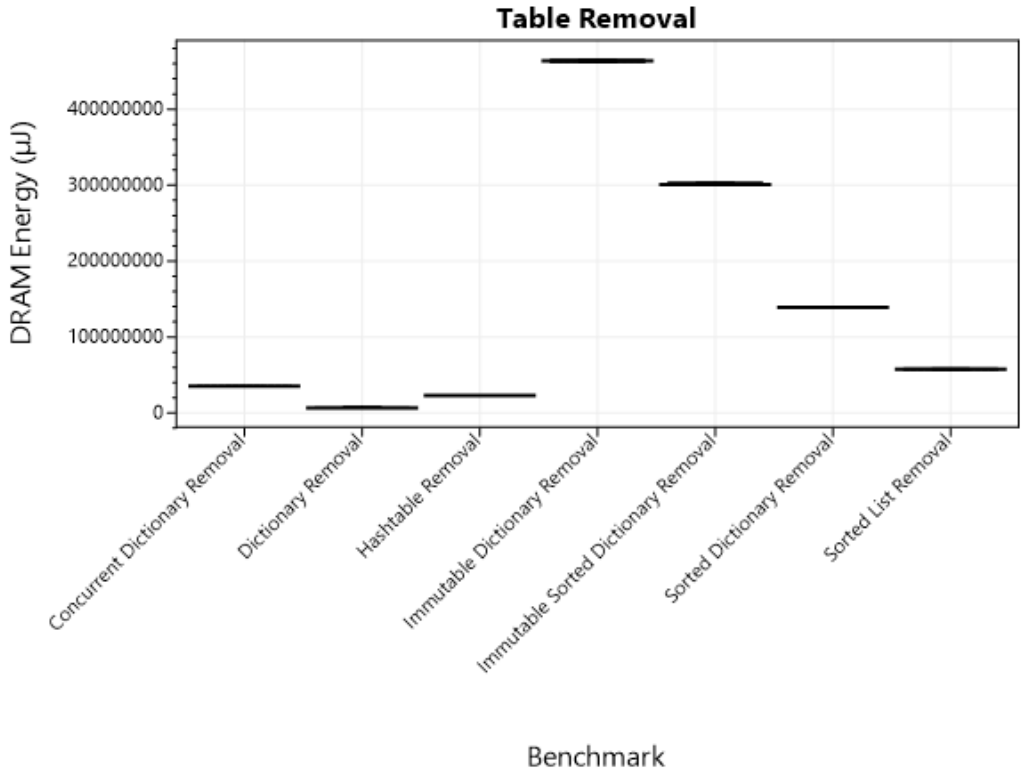

Figure A.102: Boxplot showing how efficient different Table Removal types are with regards to DRAM Energy.

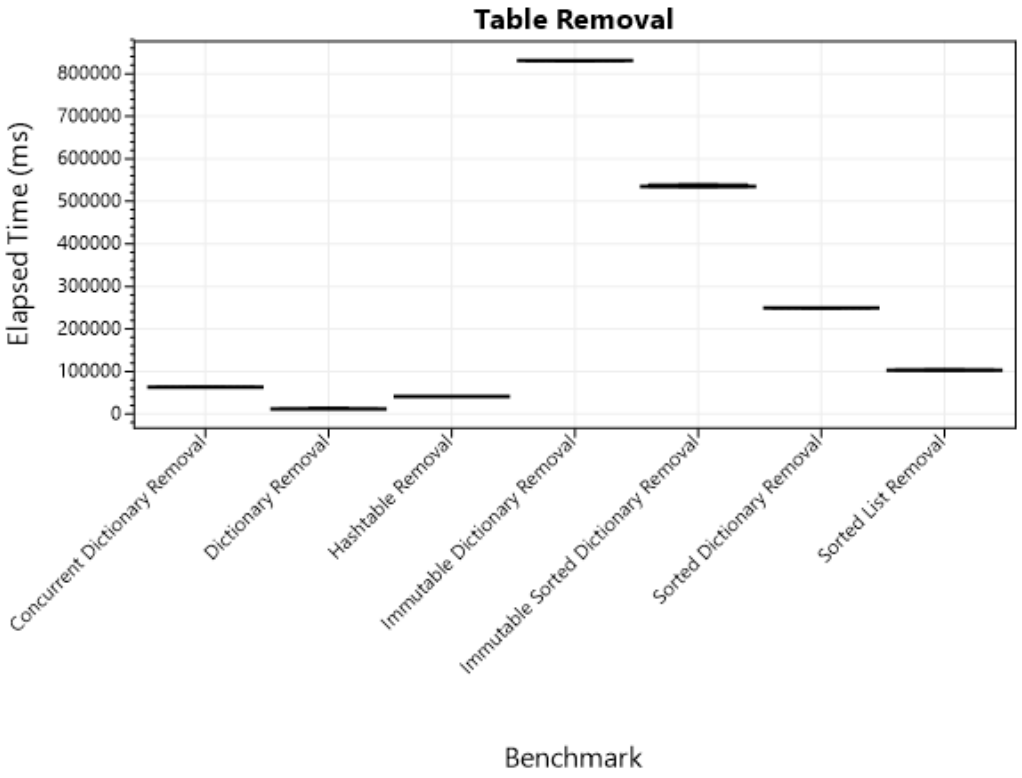

Figure A.103: Boxplot showing how efficient different Table Removal types are with regards to Elapsed Time.

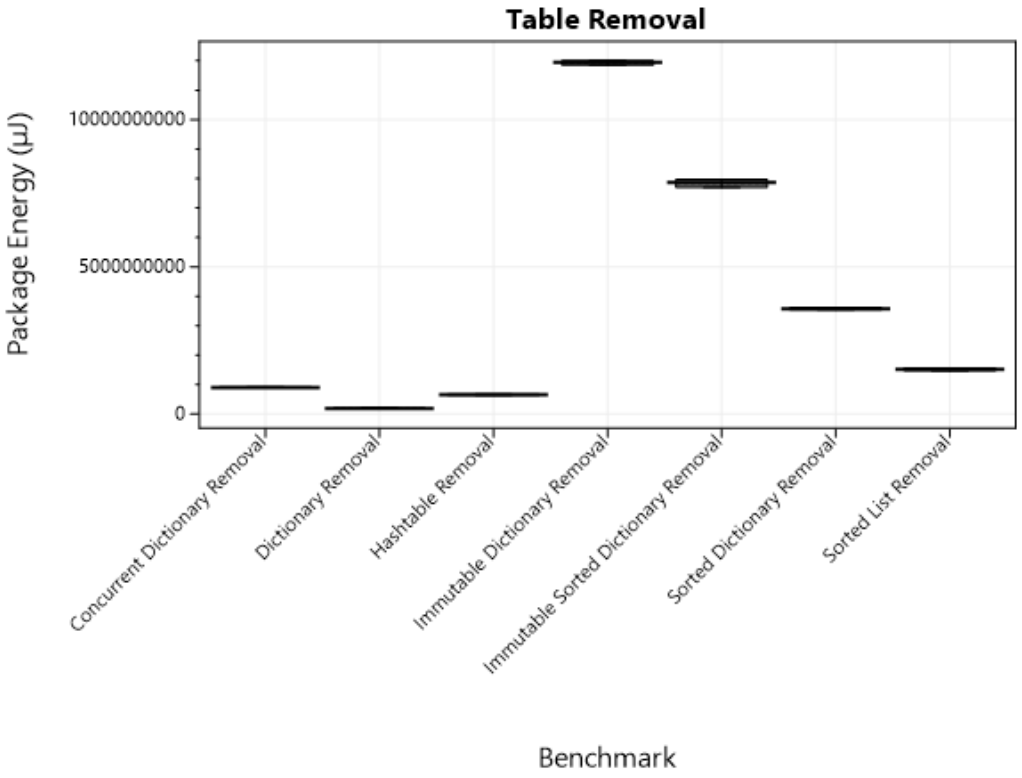

Figure A.104: Boxplot showing how efficient different Table Removal types are with regards to Package Energy.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Concurrent Dictionary Removal	62.810,505642	902.511.328,125	35.142.795,139
Dictionary Removal	12.227,097622	182.749.155,767	6.801.570,292
Hashtable Removal	40.768,450521	651.775.786,675	22.919.921,875
Immutable Dictionary Removal	830.434,756944	11.927.679.861,111	463.251.041,667
Immutable Sorted Dictionary Removal	535.102,812500	7.839.482.514,881	300.756.324,405
Sorted Dictionary Removal	249.131,501736	3.572.502.604,167	138.865.277,778
Sorted List Removal	103.534,748884	1.511.867.075,893	57.596.679,687

Table A.129: Table showing the elapsed time and energy measurement for each Table Removal.

Elapsed Time *p*-Values	Concurrent Dictionary Removal	Dictionary Removal	Hashtable Removal	Immutable Dictionary Removal	Immutable Sorted Dictionary Removal	Sorted Dictionary Removal	Sorted List Removal
Concurrent Dictionary Removal	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Dictionary Removal	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Hashtable Removal	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Immutable Dictionary Removal	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05
Immutable Sorted Dictionary Removal	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Sorted Dictionary Removal	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05
Sorted List Removal	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-

Table A.130: Table showing the *p*-values for the group Table Removal with regards to Elapsed Time.

Package Energy *p*-Values	Concurrent Dictionary Removal	Dictionary Removal	Hashtable Removal	Immutable Dictionary Removal	Immutable Sorted Dictionary Removal	Sorted Dictionary Removal	Sorted List Removal
Concurrent Dictionary Removal	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Dictionary Removal	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Hashtable Removal	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Immutable Dictionary Removal	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05
Immutable Sorted Dictionary Removal	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Sorted Dictionary Removal	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05
Sorted List Removal	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-

Table A.131: Table showing the *p*-values for the group Table Removal with regards to Package Energy.

DRAM Energy *p*-Values	Concurrent Dictionary Removal	Dictionary Removal	Hashtable Removal	Immutable Dictionary Removal	Immutable Sorted Dictionary Removal	Sorted Dictionary Removal	Sorted List Removal
Concurrent Dictionary Removal	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Dictionary Removal	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Hashtable Removal	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Immutable Dictionary Removal	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05
Immutable Sorted Dictionary Removal	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Sorted Dictionary Removal	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05
Sorted List Removal	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-

Table A.132: Table showing the *p*-values for the group Table Removal with regards to DRAM Energy.

A.3.7 String Concatenation

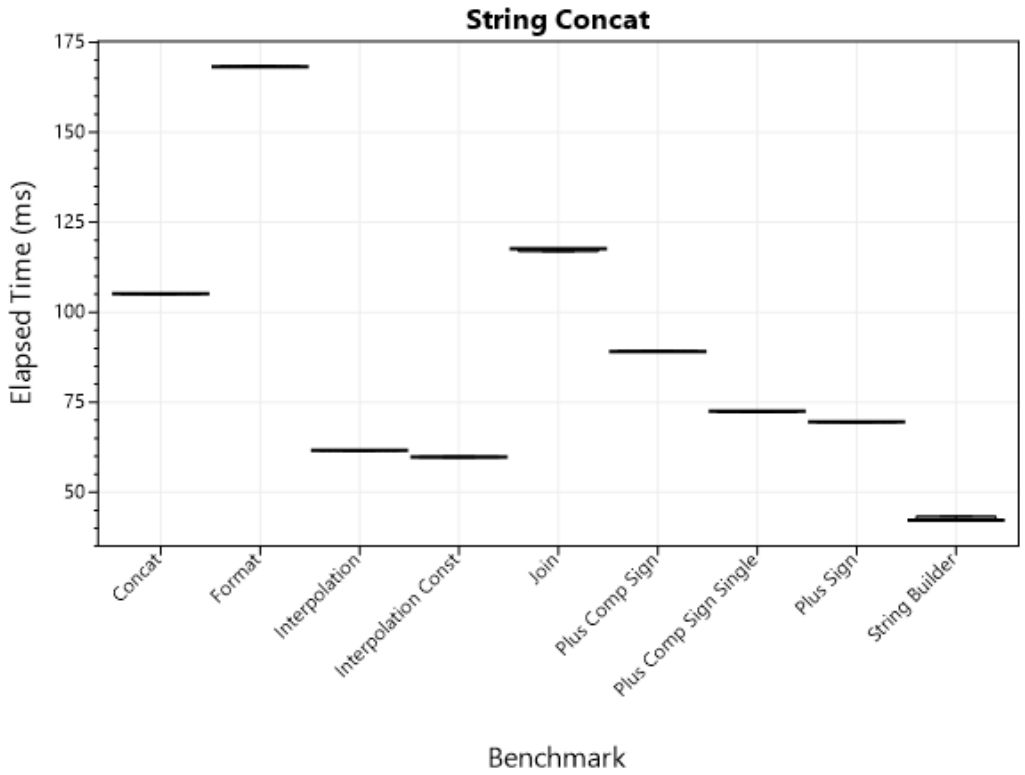

Figure A.105: Boxplot showing how efficient different String Concatenation are with regards to Elapsed Time.

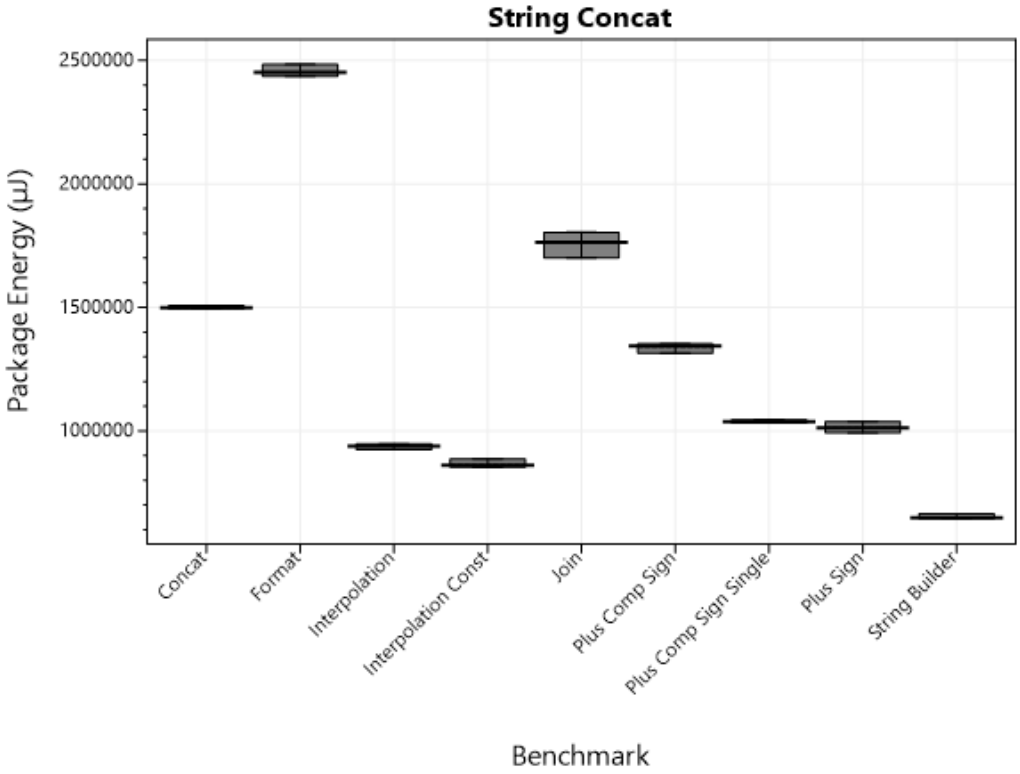

Figure A.106: Boxplot showing how efficient different String Concatenation are with regards to Package Energy.

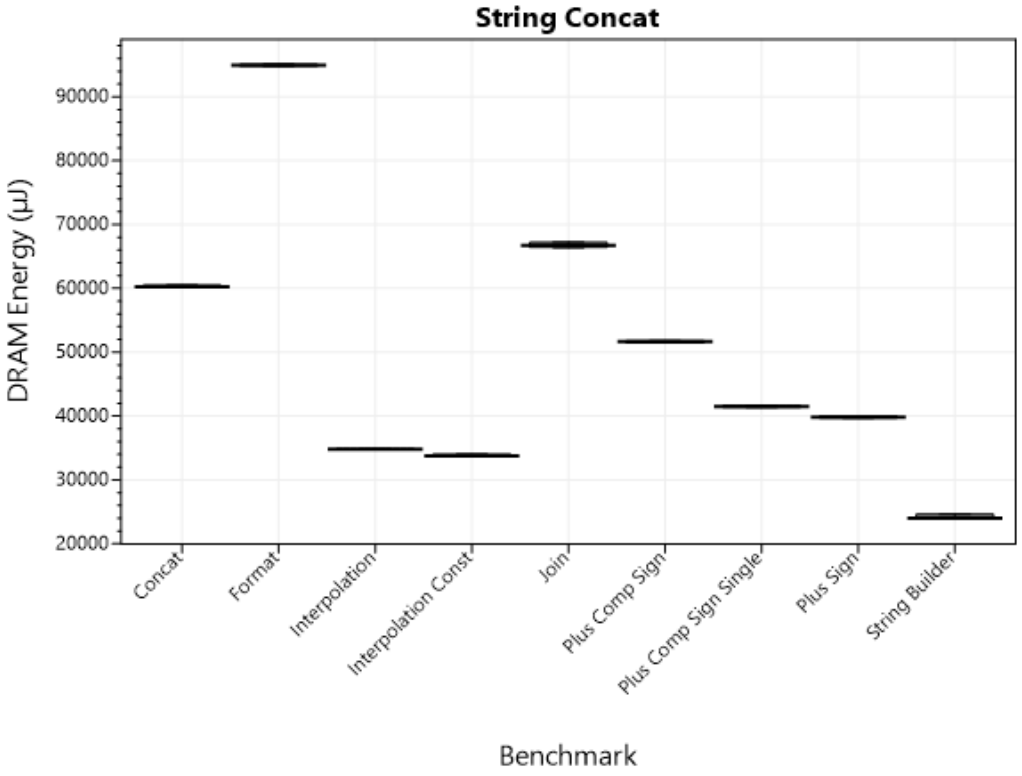

Figure A.107: Boxplot showing how efficient different String Concatenation are with regards to Dram Energy.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Concat	105,095622	1.498.695,967	60.271,962
Format	168,237707	2.452.010,854	94.942,898
Interpolation	61,647413	937.183,656	34.811,974
Interpolation Const	59,775534	862.549,460	33.796,740
Join	117,485496	1.759.847,856	66.768,384
Plus Comp Sign	89,079091	1.339.160,840	51.656,262
Plus Comp Sign Single	72,462775	1.038.106,410	41.498,099
Plus Sign	69,501949	1.012.054,016	39.811,592
String Builder	42,375883	649.596,221	24.073,461

Table A.133: Table showing the elapsed time and energy measurement for each String Concat.

Elapsed Time *p*-Values	Plus Sign	Plus Comp Sign	Plus Comp Sign Single	String Builder	Format	Concat	Join	Interpolation	Interpolation Const
Plus Sign	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Plus Comp Sign	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Plus Comp Sign Single	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
String Builder	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Format	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Concat	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05
Join	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Interpolation	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05
Interpolation Const	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-

Table A.134: Table showing the *p*-values for the group String Concat with regards to Elapsed Time.

Package Energy *p*-Values	Plus Sign	Plus Comp Sign	Plus Comp Sign Single	String Builder	Format	Concat	Join	Interpolation	Interpolation Const
Plus Sign	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Plus Comp Sign	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Plus Comp Sign Single	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
String Builder	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Format	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Concat	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05
Join	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Interpolation	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05
Interpolation Const	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-

Table A.135: Table showing the *p*-values for the group String Concat with regards to Package Energy.

DRAM Energy *p*-Values	Plus Sign	Plus Comp Sign	Plus Comp Sign Single	String Builder	Format	Concat	Join	Interpolation	Interpolation Const
Plus Sign	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Plus Comp Sign	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Plus Comp Sign Single	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
String Builder	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Format	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
Concat	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05
Join	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Interpolation	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05
Interpolation Const	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-

Table A.136: Table showing the *p*-values for the group String Concat with regards to DRAM Energy.

A.3.8 Boxing

Boxed Integer Types

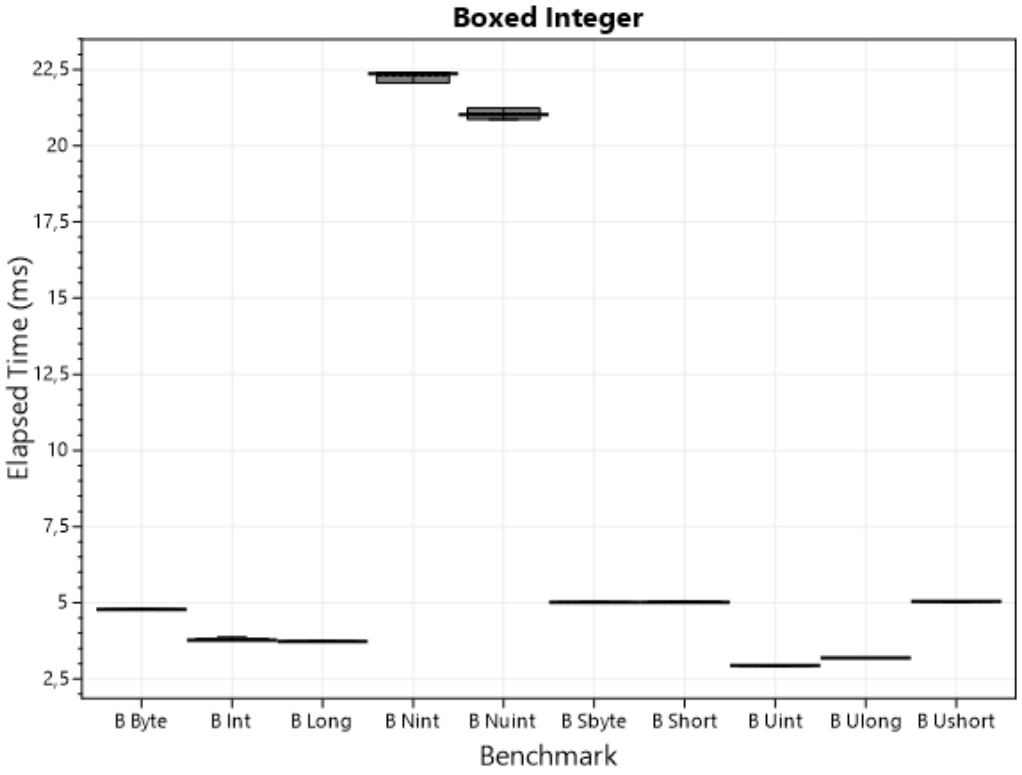

Figure A.108: Boxplot showing how efficient different Boxed Integer types are with regards to Elapsed Time.

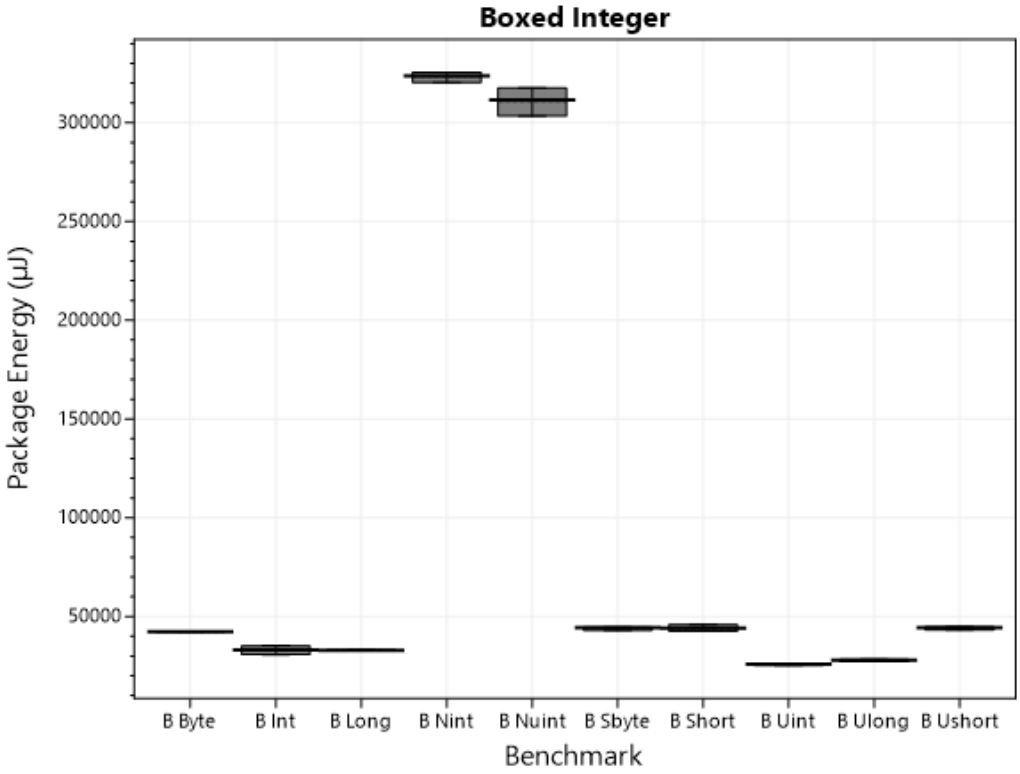

Figure A.109: Boxplot showing how efficient different Boxed Integer types are with regards to Package Energy.

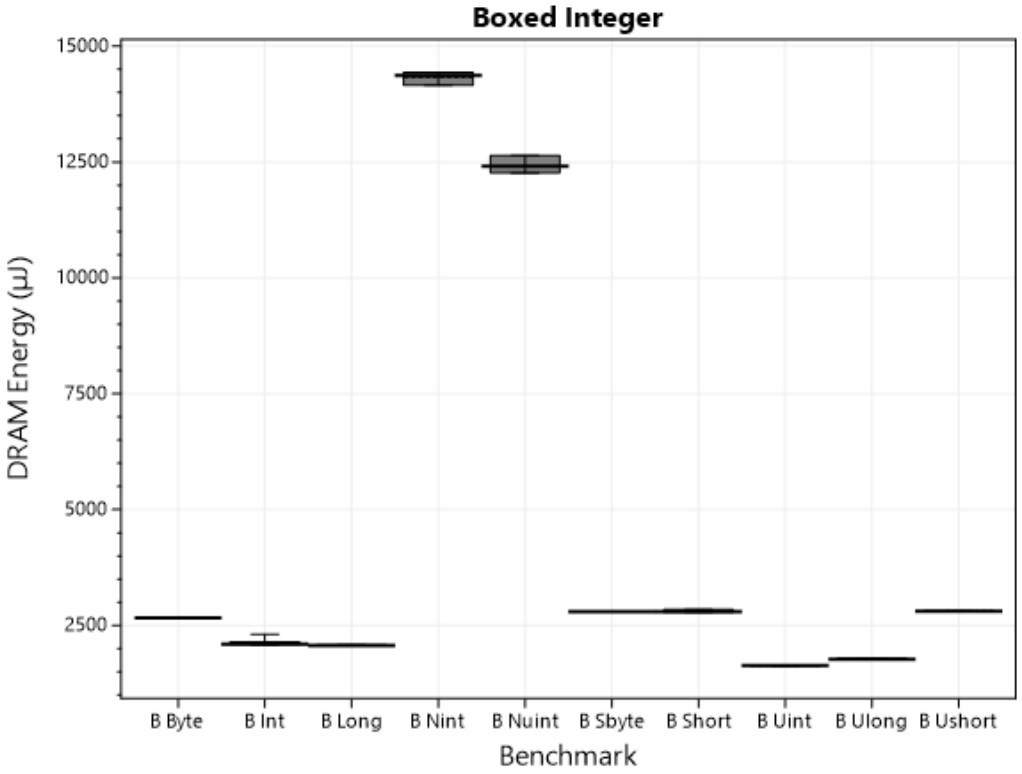

Figure A.110: Boxplot showing how efficient different Boxed Integer types are with regards to Dram Energy.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
B Byte	4,785292	42.353,808	2.661,378
B Int	3,769831	32.909,269	2.099,227
B Long	3,723809	32.944,805	2.069,924
B Nint	22,298800	323.471,758	14.329,073
B Nuint	21,061244	310.822,660	12.400,928
B Sbyte	5,019800	44.132,488	2.792,057
B Short	5,015588	43.931,457	2.790,964
B Uint	2,936213	25.729,813	1.633,539
B Ulong	3,182983	27.830,345	1.771,732
B Ushort	5,042458	44.135,280	2.803,460

Table A.137: Table showing the elapsed time and energy measurement for each Boxed Integer.

Elapsed Time *p*-Values	B Int	B Uint	B Nint	B Nuint	B Long	B Ulong	B Short	B Ushort	B Byte	B Sbyte
B Int	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
B Uint	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
B Nint	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
B Nuint	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
B Long	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
B Ulong	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
B Short	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05
B Ushort	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
B Byte	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05
B Sbyte	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-

Table A.138: Table showing the *p*-values for the group Boxed Integer with regards to Elapsed Time.

Package Energy *p*-Values	B Int	B Uint	B Nint	B Nuint	B Long	B Ulong	B Short	B Ushort	B Byte	B Sbyte
B Int	-	<0,05	<0,05	<0,05	0,884	<0,05	<0,05	<0,05	<0,05	<0,05
B Uint	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
B Nint	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
B Nuint	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
B Long	0,884	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
B Ulong	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
B Short	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	0,251	<0,05	0,309
B Ushort	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	0,251	-	<0,05	0,987
B Byte	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05
B Sbyte	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	0,309	0,987	<0,05	-

Table A.139: Table showing the *p*-values for the group Boxed Integer with regards to Package Energy.

DRAM Energy *p*-Values	B Int	B Uint	B Nint	B Nuint	B Long	B Ulong	B Short	B Ushort	B Byte	B Sbyte
B Int	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
B Uint	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
B Nint	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
B Nuint	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
B Long	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
B Ulong	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05
B Short	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	0,695
B Ushort	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
B Byte	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05
B Sbyte	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	0,695	<0,05	<0,05	-

Table A.140: Table showing the *p*-values for the group Boxed Integer with regards to DRAM Energy.

Boxed Decimal Types

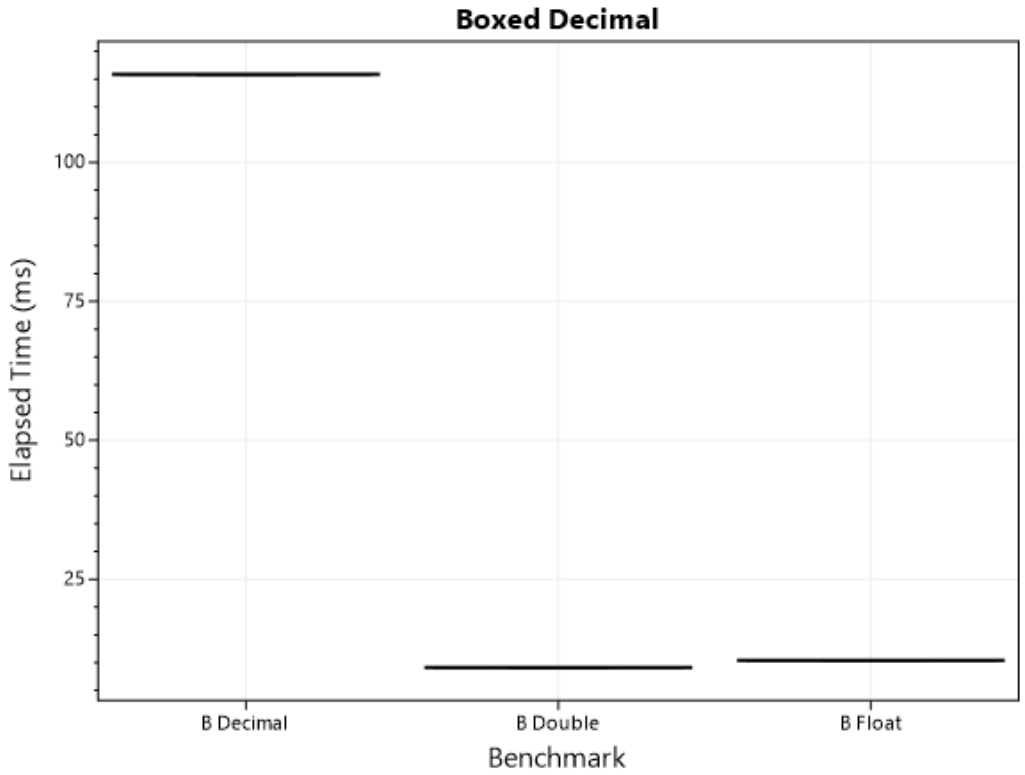

Figure A.111: Boxplot showing how efficient different Boxed Decimal types are with regards to Elapsed Time.

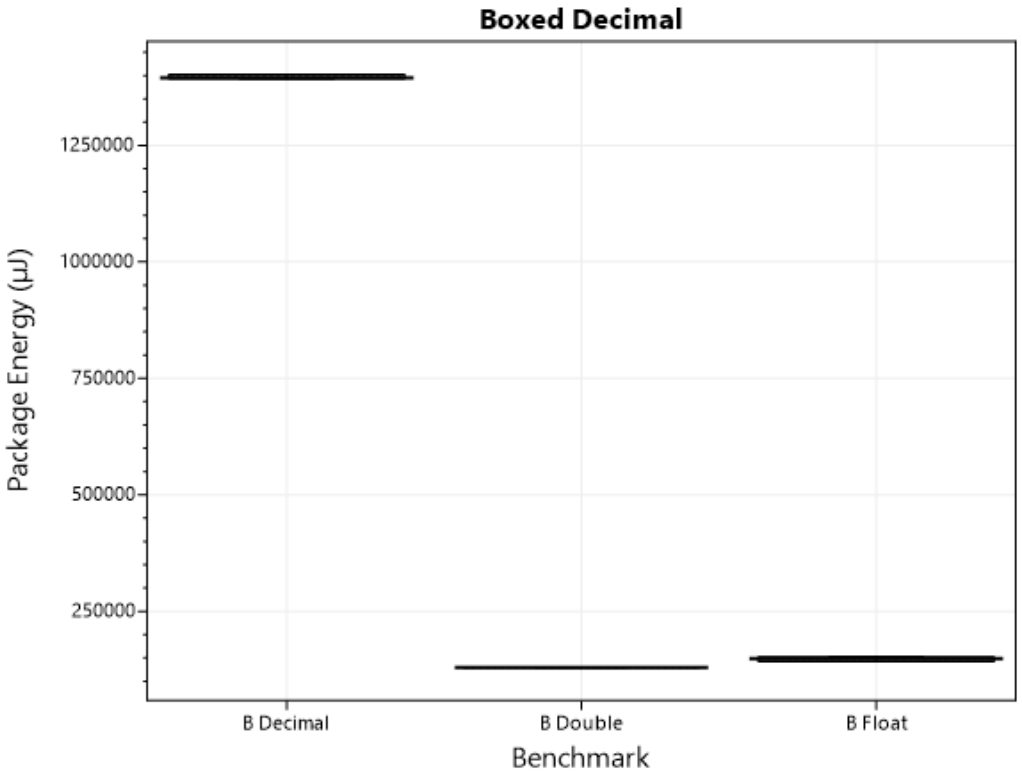

Figure A.112: Boxplot showing how efficient different Boxed Decimal types are with regards to Package Energy.

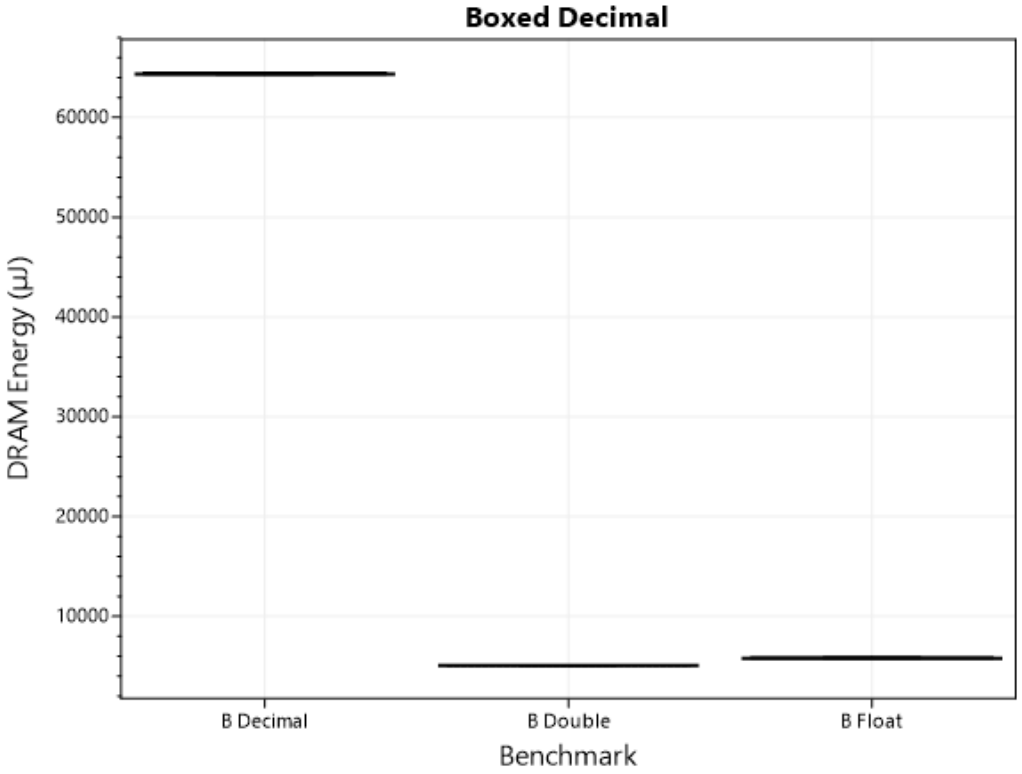

Figure A.113: Boxplot showing how efficient different Boxed Decimal types are with regards to Dram Energy.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
B Decimal	115,834380	1.397.003,555	64.403,364
B Double	9,104699	129.586,090	5.065,184
B Float	10,431647	148.731,747	5.805,426

Table A.141: Table showing the elapsed time and energy measurement for each Boxed Decimal.

Elapsed Time *p*-Values	B Float	B Double	B Decimal
B Float	-	<0,05	<0,05
B Double	<0,05	-	<0,05
B Decimal	<0,05	<0,05	-

Table A.142: Table showing the *p*-values for the group Boxed Decimal with regards to Elapsed Time.

Package Energy *p*-Values	B Float	B Double	B Decimal
B Float	-	<0,05	<0,05
B Double	<0,05	-	<0,05
B Decimal	<0,05	<0,05	-

Table A.143: Table showing the *p*-values for the group Boxed Decimal with regards to Package Energy.

DRAM Energy *p*-Values	B Float	B Double	B Decimal
B Float	-	<0,05	<0,05
B Double	<0,05	-	<0,05
B Decimal	<0,05	<0,05	-

Table A.144: Table showing the *p*-values for the group Boxed Decimal with regards to DRAM Energy.

Boxed Boolean Types

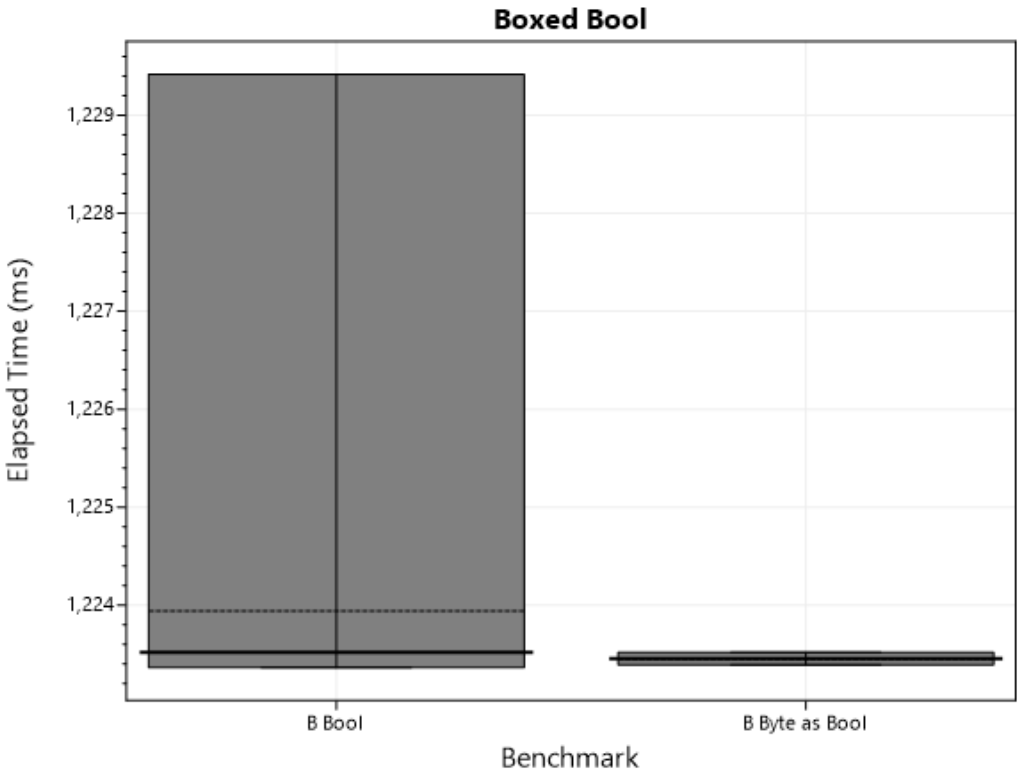

Figure A.114: Boxplot showing how efficient different Boxed Bool types are with regards to Elapsed Time.

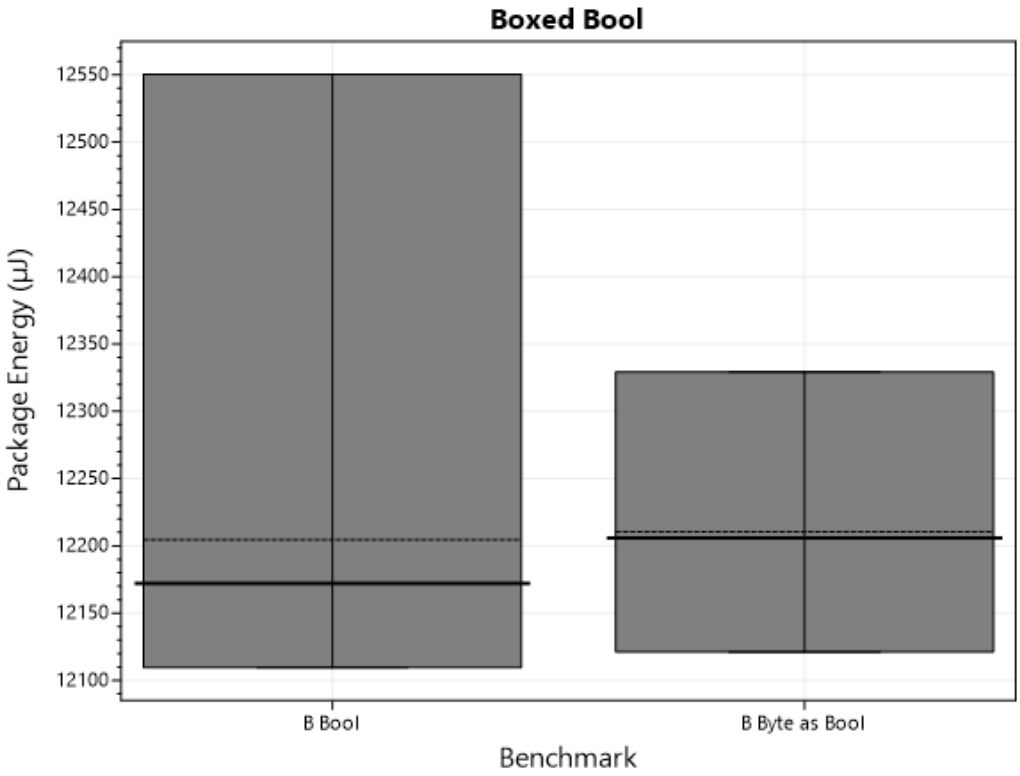

Figure A.115: Boxplot showing how efficient different Boxed Bool types are with regards to Package Energy.

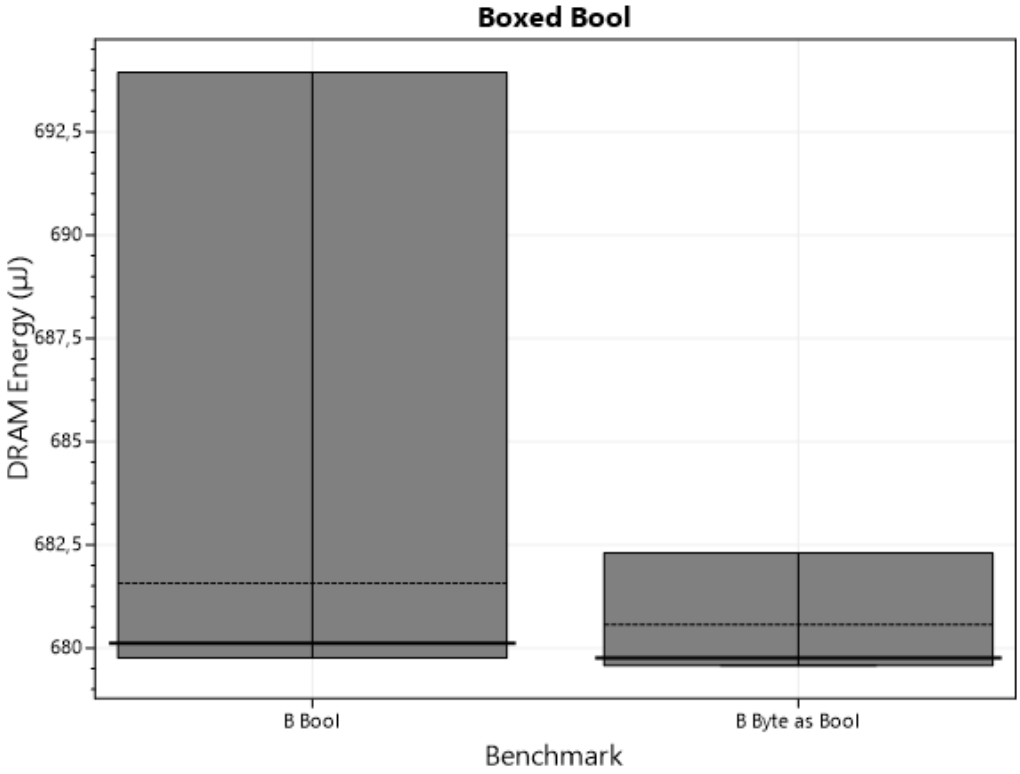

Figure A.116: Boxplot showing how efficient different Boxed Bool types are with regards to Dram Energy.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
B Bool	1,223939	12.204,422	681,574
B Byte as Bool	1,223451	12.210,319	680,563

Table A.145: Table showing the elapsed time and energy measurement for each Boxed Bool.

Elapsed Time p-Values	B Bool	B Byte as Bool
B Bool	-	0,389
B Byte as Bool	0,389	-

Table A.146: Table showing the p-values for the group Boxed Bool with regards to Elapsed Time.

Package Energy p-Values	B Bool	B Byte as Bool
B Bool	-	0,896
B Byte as Bool	0,896	-

Table A.147: Table showing the p-values for the group Boxed Bool with regards to Package Energy.

DRAM Energy p-Values	B Bool	B Byte as Bool
B Bool	-	0,449
B Byte as Bool	0,449	-

Table A.148: Table showing the p-values for the group Boxed Bool with regards to DRAM Energy.

A.3.9 Invocation

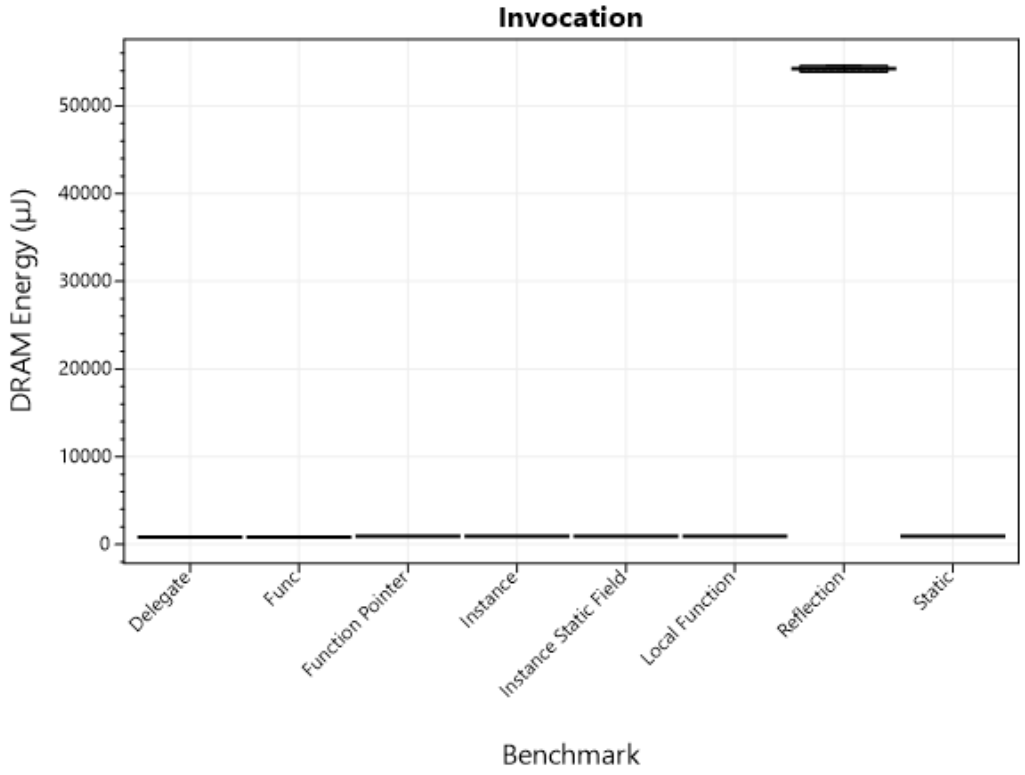

Figure A.117: Boxplot showing how efficient different Invocation types are with regards to DRAM Energy.

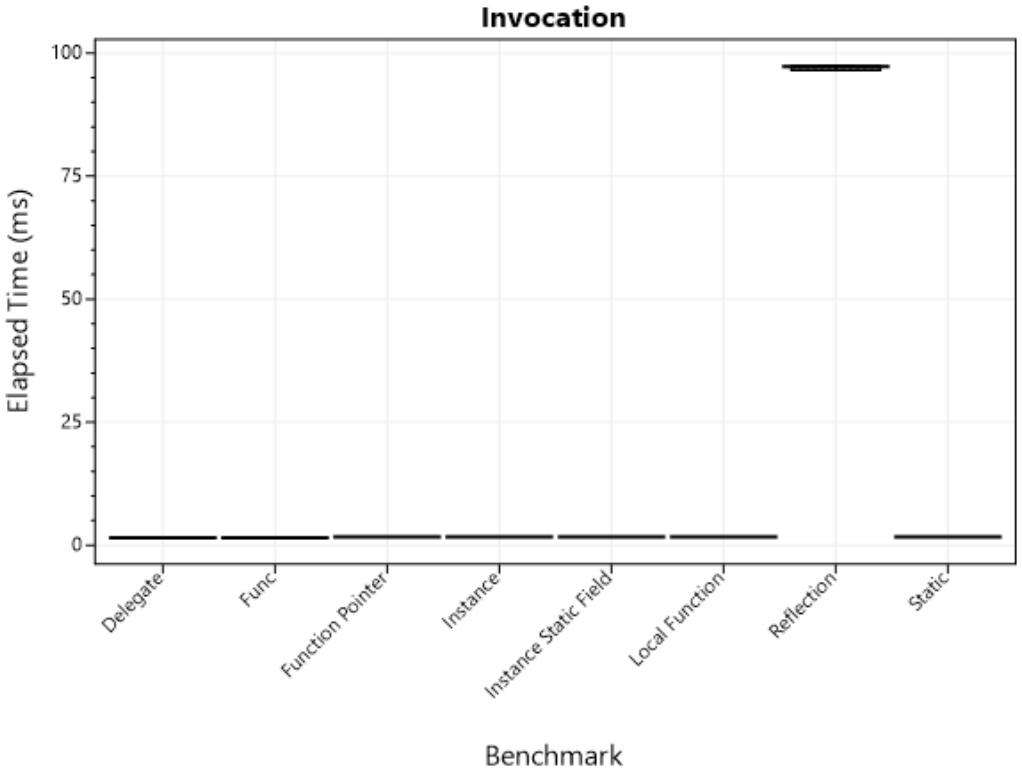

Figure A.118: Boxplot showing how efficient different Invocation types are with regards to Elapsed Time.

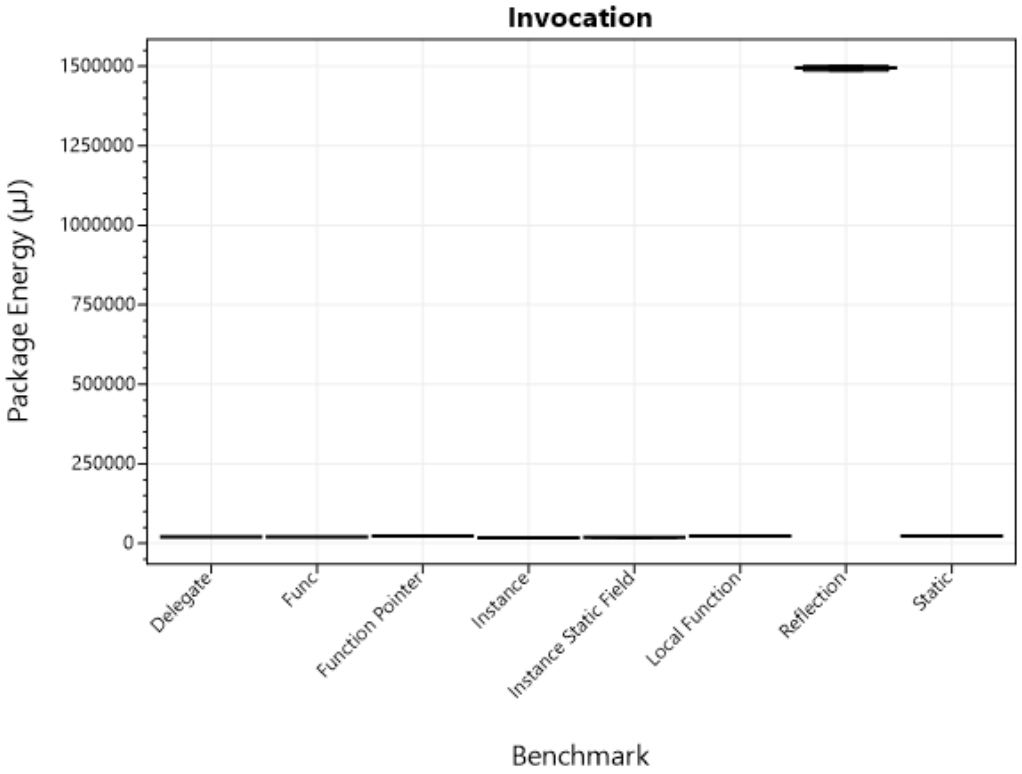

Figure A.119: Boxplot showing how efficient different Invocation types are with regards to Package Energy.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Delegate	1,467887	20.793,926	816,441
Func	1,467897	20.754,333	816,825
Function Pointer	1,713080	22.556,866	953,107
Instance	1,705568	17.844,522	948,513
Instance Static Field	1,703423	18.290,845	947,662
Local Function	1,713078	22.495,021	952,824
Reflection	97,006671	1.493.560,791	54.132,970
Static	1,713041	22.541,142	952,703

Table A.149: Table showing the elapsed time and energy measurement for each Invocation.

Elapsed Time *p*-Values	Instance	Instance Static Field	Static	Local Function	Func	Reflection	Function Pointer	Delegate
Instance	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Instance Static Field	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Static	<0,05	<0,05	-	0,538	<0,05	<0,05	0,508	<0,05
Local Function	<0,05	<0,05	0,538	-	<0,05	<0,05	0,956	<0,05
Func	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	0,765
Reflection	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Function Pointer	<0,05	<0,05	0,508	0,956	<0,05	<0,05	-	<0,05
Delegate	<0,05	<0,05	<0,05	<0,05	0,765	<0,05	<0,05	-

Table A.150: Table showing the *p*-values for the group Invocation with regards to Elapsed Time.

Package Energy *p*-Values	Instance	Instance Static Field	Static	Local Function	Func	Reflection	Function Pointer	Delegate
Instance	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Instance Static Field	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Static	<0,05	<0,05	-	0,210	<0,05	<0,05	0,703	<0,05
Local Function	<0,05	<0,05	0,210	-	<0,05	<0,05	0,173	<0,05
Func	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	0,112
Reflection	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Function Pointer	<0,05	<0,05	0,703	0,173	<0,05	<0,05	-	<0,05
Delegate	<0,05	<0,05	<0,05	<0,05	0,112	<0,05	<0,05	-

Table A.151: Table showing the *p*-values for the group Invocation with regards to Package Energy.

DRAM Energy *p*-Values	Instance	Instance Static Field	Static	Local Function	Func	Reflection	Function Pointer	Delegate
Instance	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Instance Static Field	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Static	<0,05	<0,05	-	0,874	<0,05	<0,05	0,601	<0,05
Local Function	<0,05	<0,05	0,874	-	<0,05	<0,05	0,723	<0,05
Func	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	0,447
Reflection	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Function Pointer	<0,05	<0,05	0,601	0,723	<0,05	<0,05	-	<0,05
Delegate	<0,05	<0,05	<0,05	<0,05	0,447	<0,05	<0,05	-

Table A.152: Table showing the *p*-values for the group Invocation with regards to DRAM Energy.

Delegate

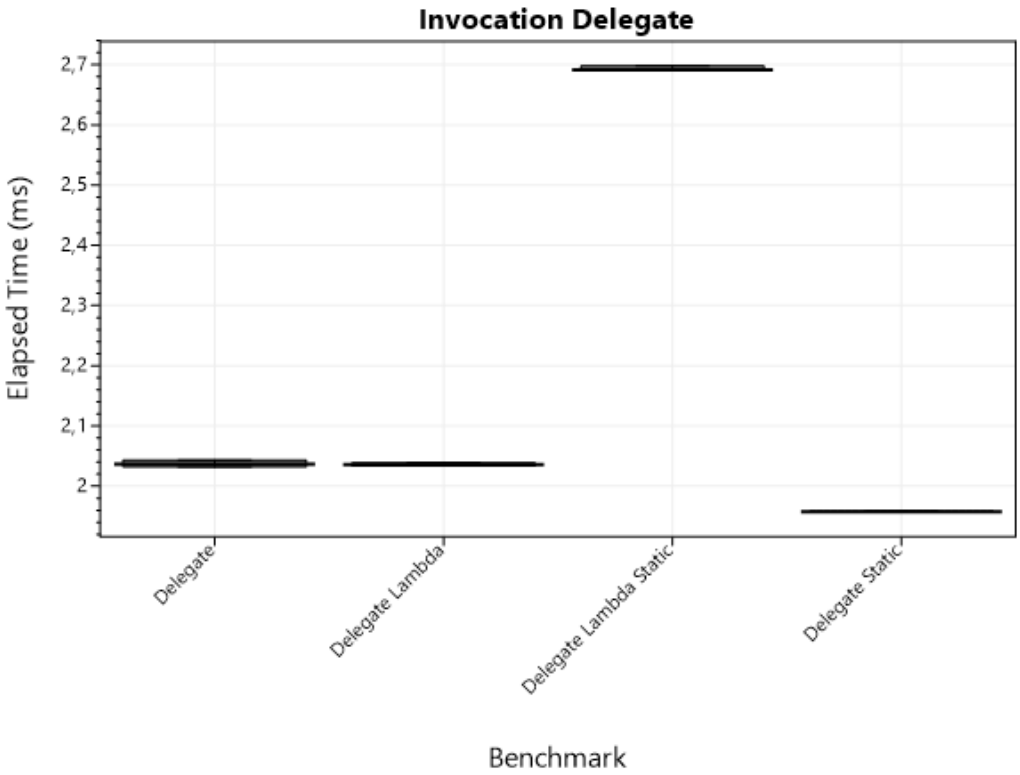

Figure A.120: Boxplot showing how efficient different Invocation types are with regards to Elapsed Time.

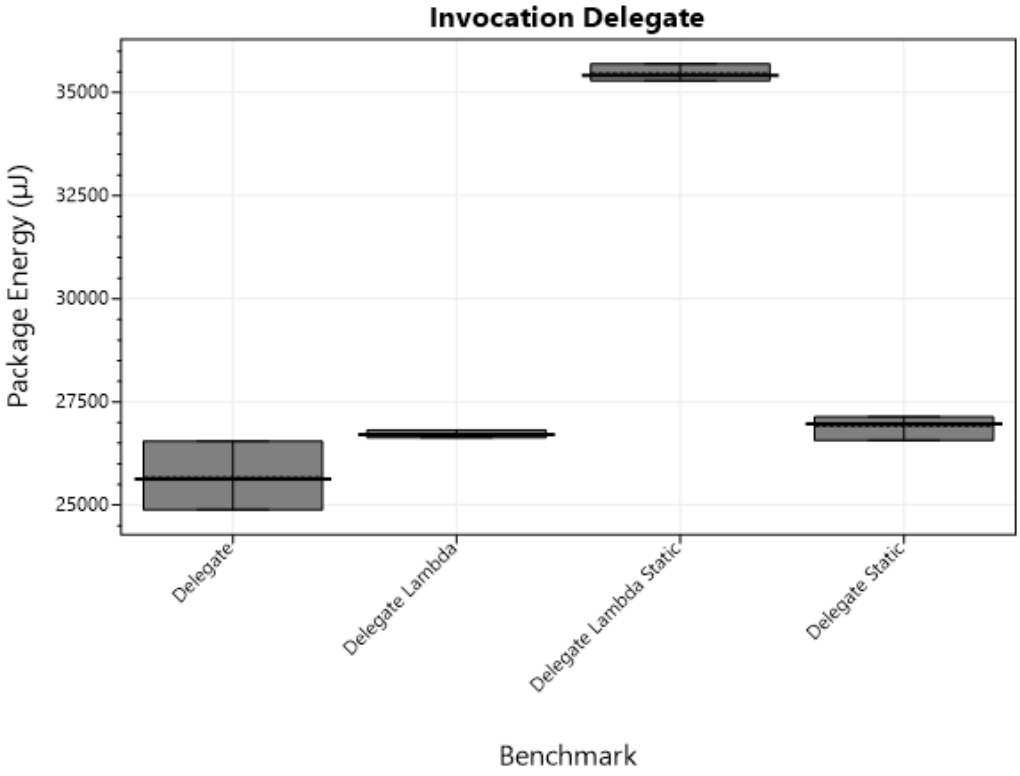

Figure A.121: Boxplot showing how efficient different Invocation types are with regards to Package Energy.

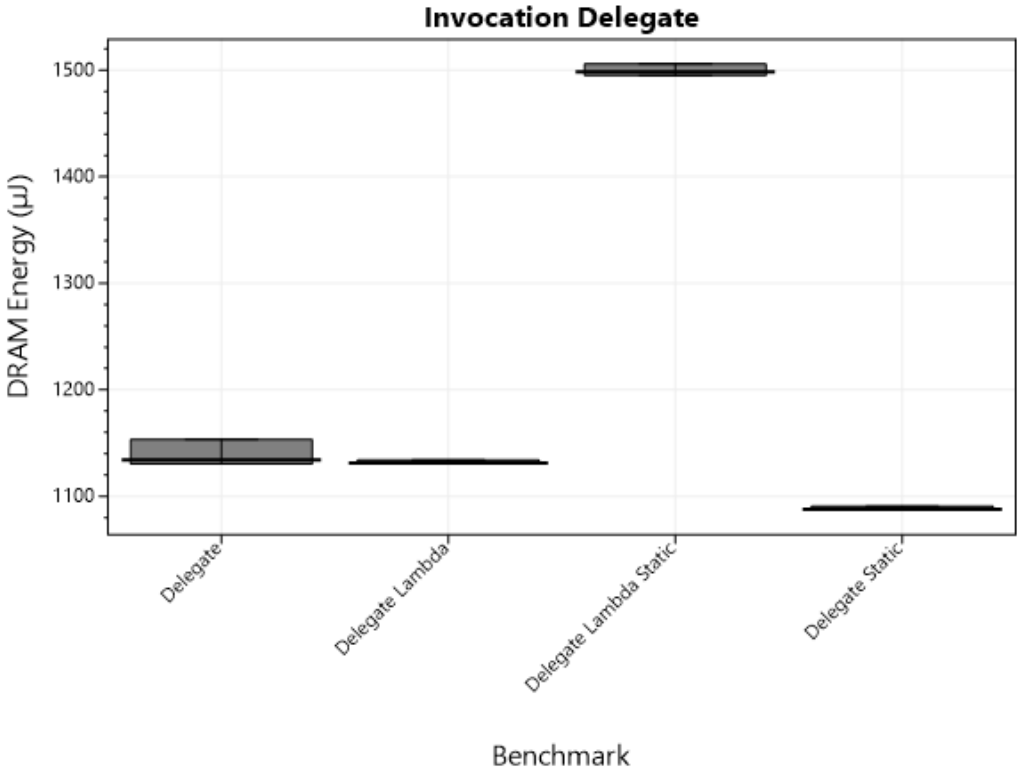

Figure A.122: Boxplot showing how efficient different Invocation types are with regards to Dram Energy.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Delegate	2,036381	25.669,072	1.133,515
Delegate Lambda	2,035998	26.709,608	1.132,096
Delegate Lambda Static	2,692120	35.465,190	1.498,075
Delegate Static	1,957641	26.916,932	1.088,319

Table A.153: Table showing the elapsed time and energy measurement for each Invocation Delegate.

Elapsed Time p-Values	Delegate	Delegate Static	Delegate Lambda	Delegate Lambda Static
Delegate	-	<0,05	0,611	<0,05
Delegate Static	<0,05	-	<0,05	<0,05
Delegate Lambda	0,611	<0,05	-	<0,05
Delegate Lambda Static	<0,05	<0,05	<0,05	-

Table A.154: Table showing the p-values for the group Invocation Delegate with regards to Elapsed Time.

Package Energy *p*-Values	Delegate	Delegate Static	Delegate Lambda	Delegate Lambda Static
Delegate	-	<0,05	<0,05	<0,05
Delegate Static	<0,05	-	<0,05	<0,05
Delegate Lambda	<0,05	<0,05	-	<0,05
Delegate Lambda Static	<0,05	<0,05	<0,05	-

Table A.155: Table showing the *p*-values for the group Invocation Delegate with regards to Package Energy.

DRAM Energy *p*-Values	Delegate	Delegate Static	Delegate Lambda	Delegate Lambda Static
Delegate	-	<0,05	0,287	<0,05
Delegate Static	<0,05	-	<0,05	<0,05
Delegate Lambda	0,287	<0,05	-	<0,05
Delegate Lambda Static	<0,05	<0,05	<0,05	-

Table A.156: Table showing the *p*-values for the group Invocation Delegate with regards to DRAM Energy.

Function

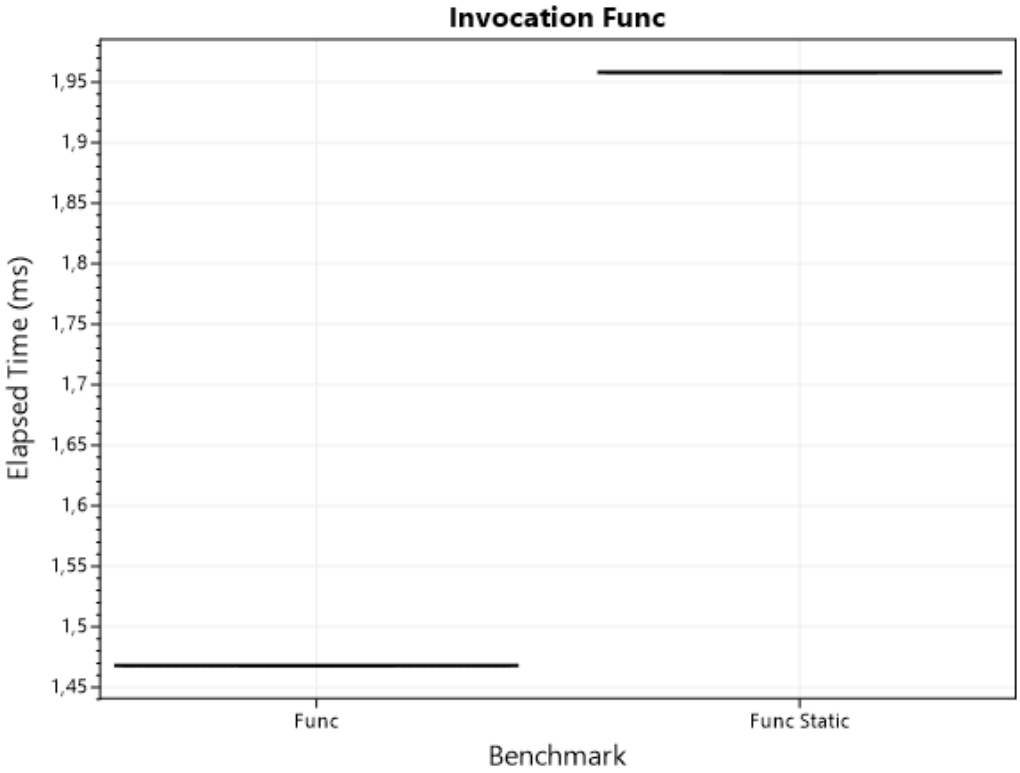

Figure A.123: Boxplot showing how efficient different Invocation types are with regards to Elapsed Time.

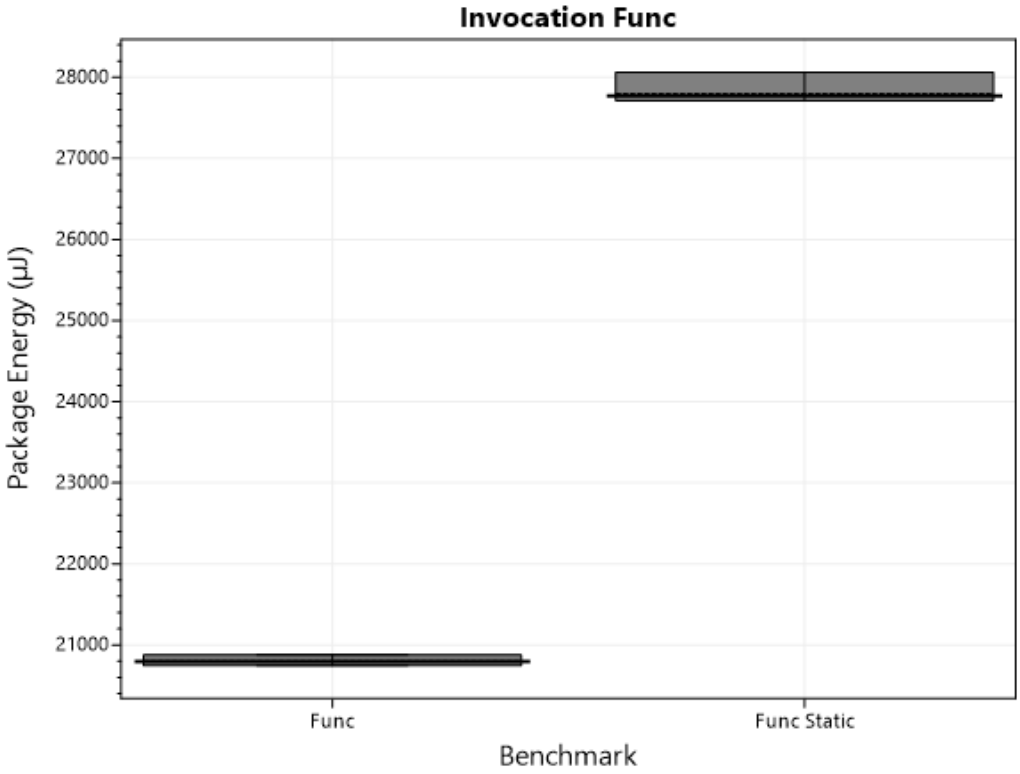

Figure A.124: Boxplot showing how efficient different Invocation types are with regards to Package Energy.

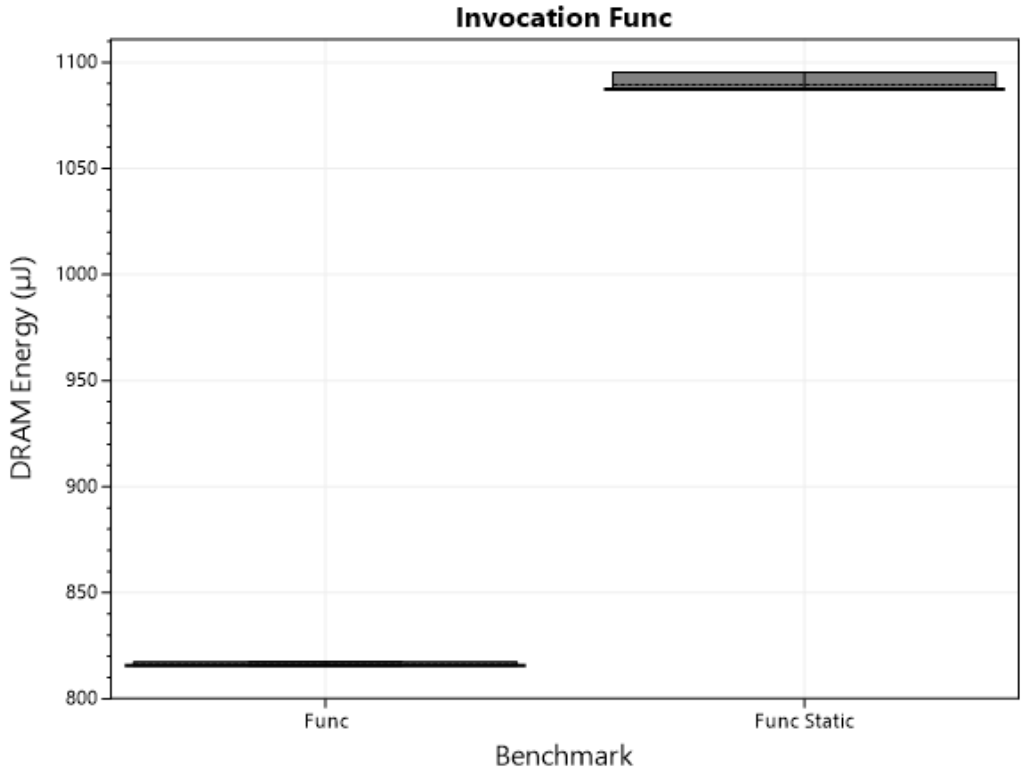

Figure A.125: Boxplot showing how efficient different Invocation types are with regards to Dram Energy.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Func	1,467980	20.801,849	816,360
Func Static	1,957950	27.798,332	1.089,409

Table A.157: Table showing the elapsed time and energy measurement for each Invocation Func.

Elapsed Time *p*-Values	Func	Func Static
Func	-	<0,05
Func Static	<0,05	-

Table A.158: Table showing the *p*-values for the group Invocation Func with regards to Elapsed Time.

Package Energy *p*-Values	Func	Func Static
Func	-	<0,05
Func Static	<0,05	-

Table A.159: Table showing the *p*-values for the group Invocation Func with regards to Package Energy.

DRAM Energy *p*-Values	Func	Func Static
Func	-	<0,05
Func Static	<0,05	-

Table A.160: Table showing the *p*-values for the group Invocation Func with regards to DRAM Energy.

Local Function

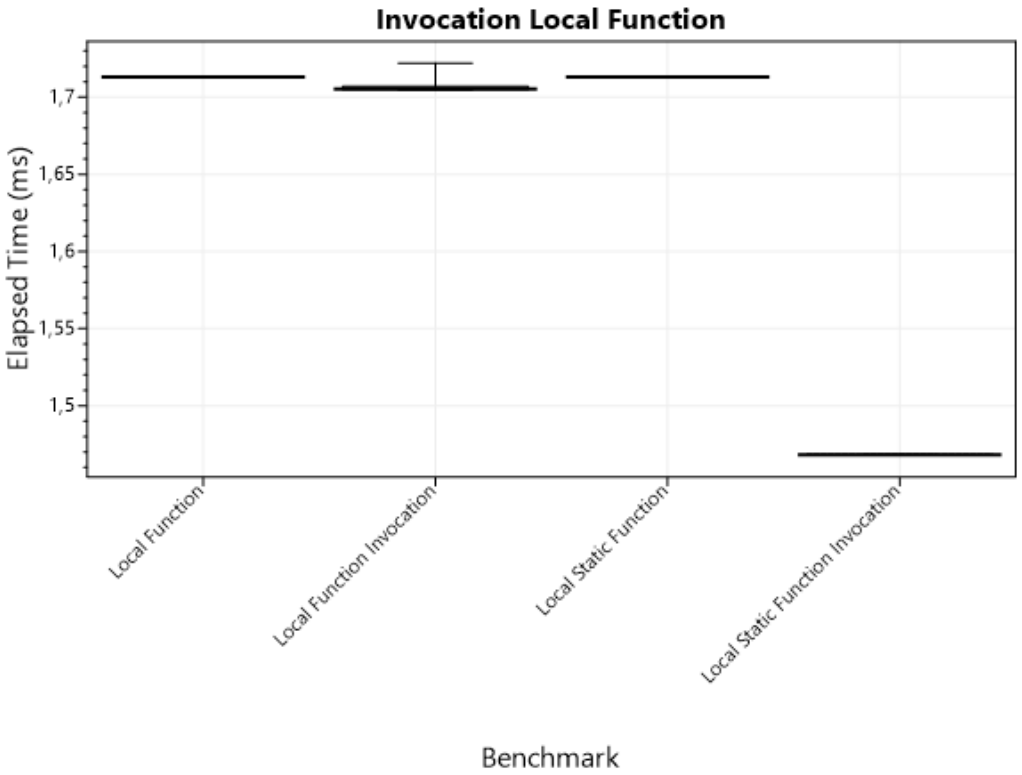

Figure A.126: Boxplot showing how efficient different Invocation types are with regards to Elapsed Time.

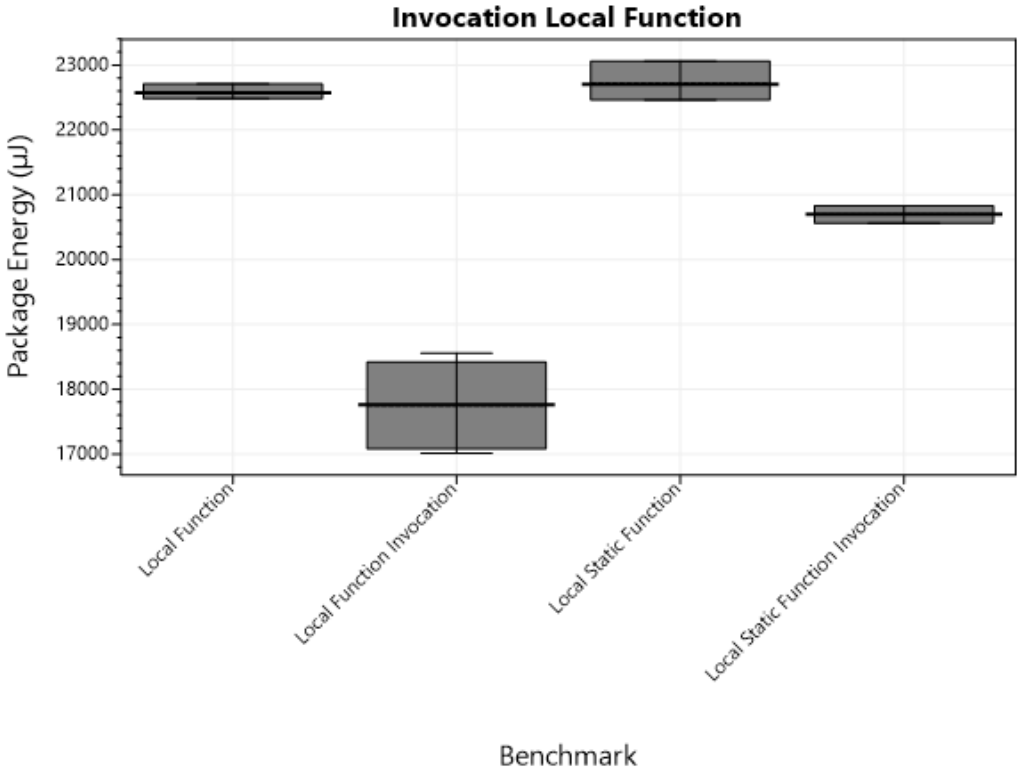

Figure A.127: Boxplot showing how efficient different Invocation types are with regards to Package Energy.

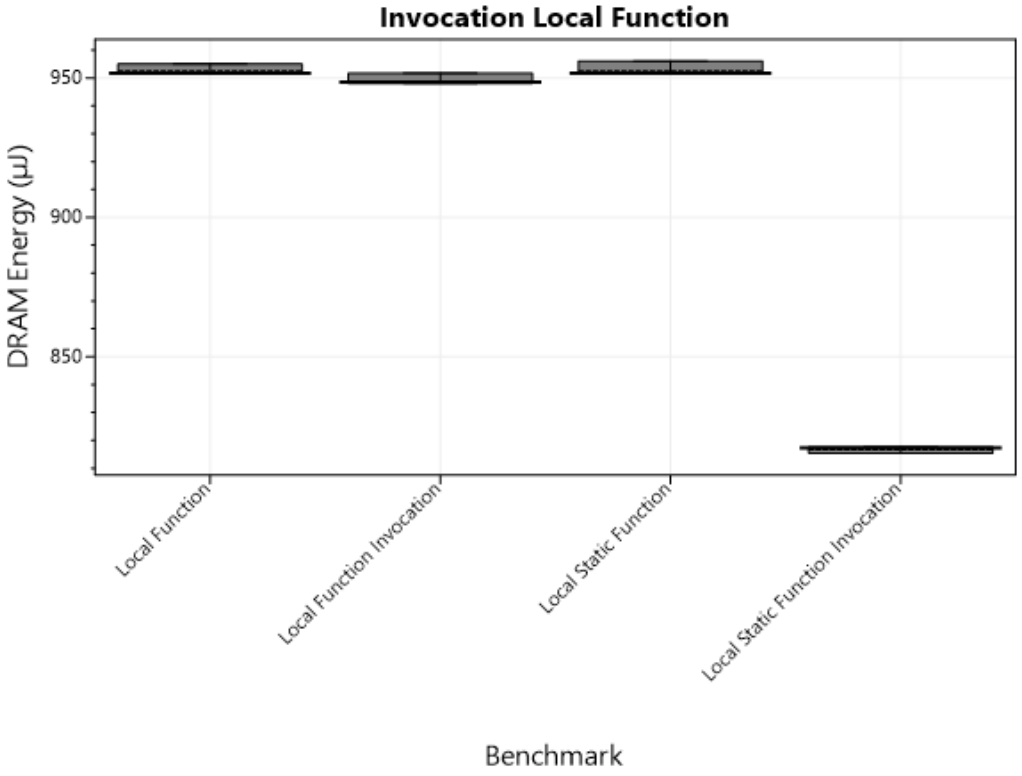

Figure A.128: Boxplot showing how efficient different Invocation types are with regards to Dram Energy.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Local Function	1,713111	22.572,590	952,421
Local Function Invocation	1,705528	17.748,136	948,553
Local Static Function	1,713017	22.726,087	952,533
Local Static Function Invocation	1,468121	20.700,552	816,623

Table A.161: Table showing the elapsed time and energy measurement for each Invocation Local Function.

Elapsed Time *p*-Values	Local Function	Local Static Function	Local Function Invocation	Local Static Function Invocation
Local Function	-	0,179	<0,05	<0,05
Local Static Function	0,179	-	<0,05	<0,05
Local Function Invocation	<0,05	<0,05	-	<0,05
Local Static Function Invocation	<0,05	<0,05	<0,05	-

Table A.162: Table showing the *p*-values for the group Invocation Local Function with regards to Elapsed Time.

Package Energy *p*-Values	Local Function	Local Static Function	Local Function Invocation	Local Static Function Invocation
Local Function	-	0,053	<0,05	<0,05
Local Static Function	0,053	-	<0,05	<0,05
Local Function Invocation	<0,05	<0,05	-	<0,05
Local Static Function Invocation	<0,05	<0,05	<0,05	-

Table A.163: Table showing the *p*-values for the group Invocation Local Function with regards to Package Energy.

DRAM Energy *p*-Values	Local Function	Local Static Function	Local Function Invocation	Local Static Function Invocation
Local Function	-	0,871	<0,05	<0,05
Local Static Function	0,871	-	<0,05	<0,05
Local Function Invocation	<0,05	<0,05	-	<0,05
Local Static Function Invocation	<0,05	<0,05	<0,05	-

Table A.164: Table showing the *p*-values for the group Invocation Local Function with regards to DRAM Energy.

Reflection

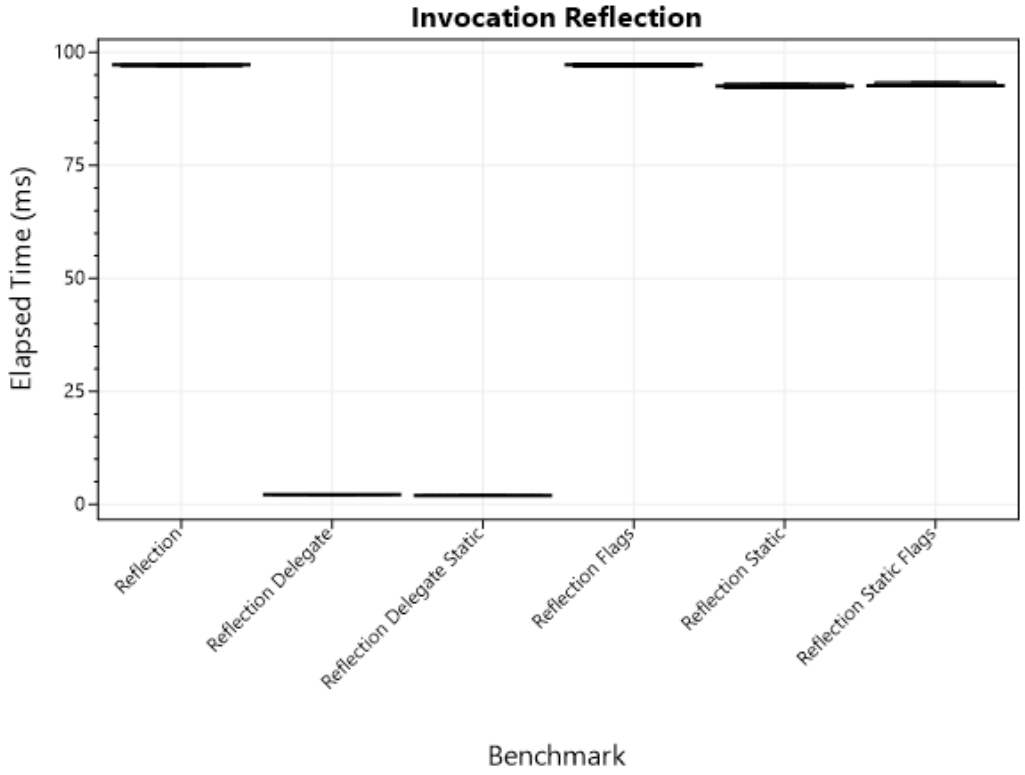

Figure A.129: Boxplot showing how efficient different Invocation types are with regards to Elapsed Time.

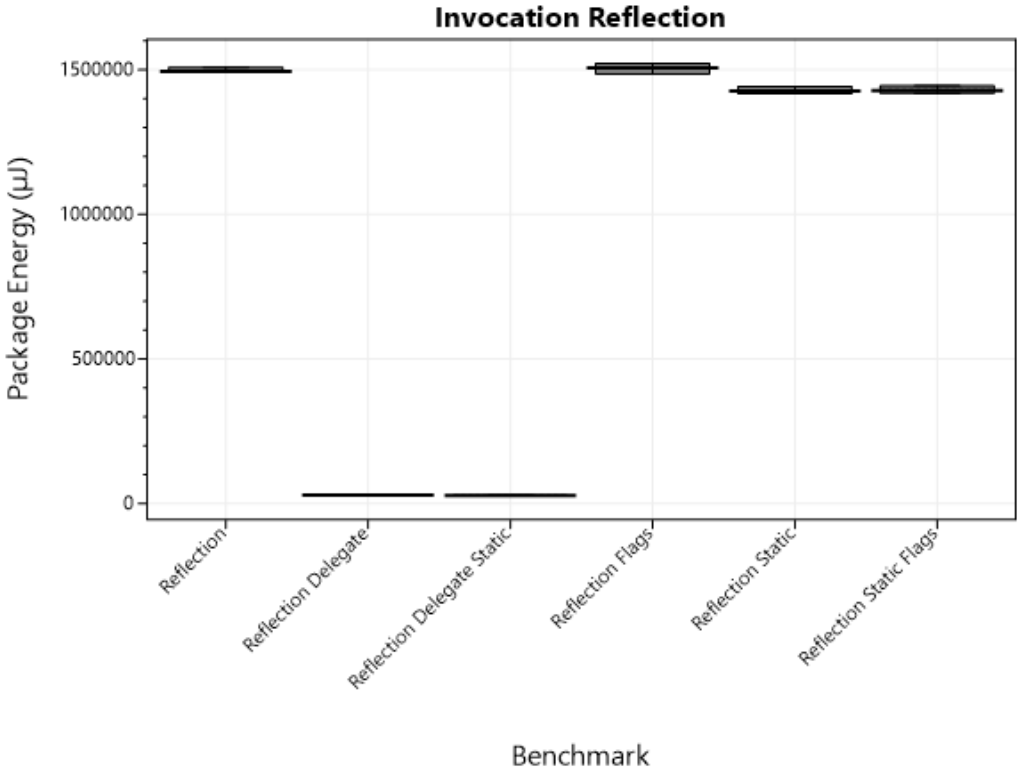

Figure A.130: Boxplot showing how efficient different Invocation types are with regards to Package Energy.

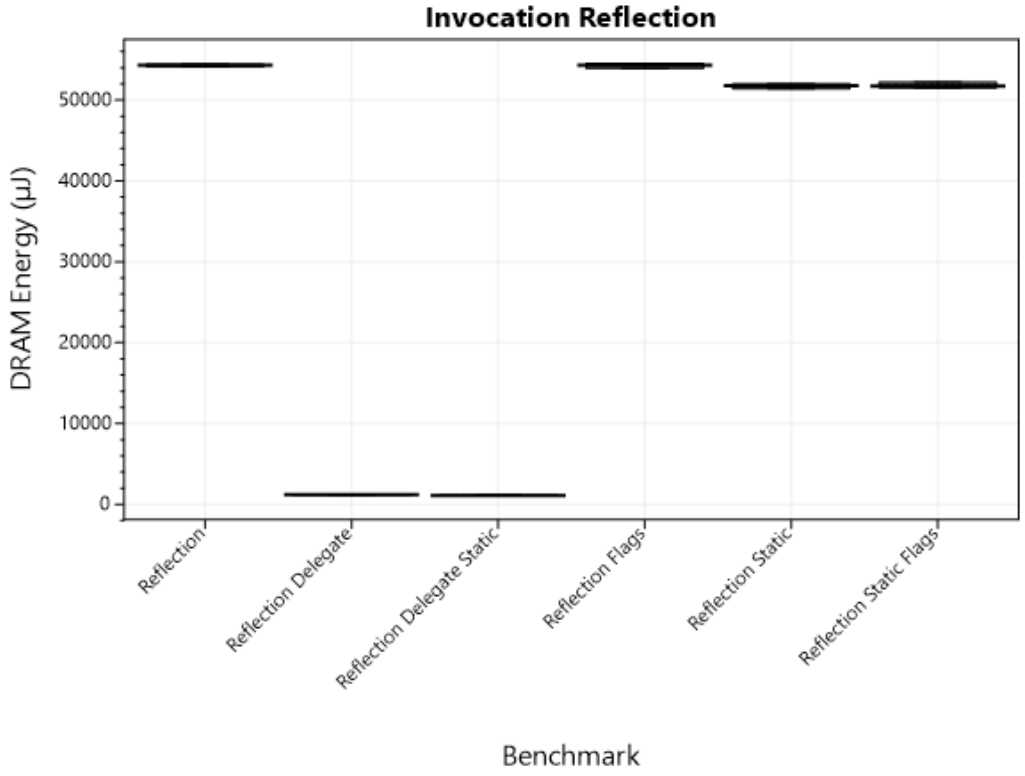

Figure A.131: Boxplot showing how efficient different Invocation types are with regards to Dram Energy.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Reflection	97,219471	1.497.731,018	54.298,528
Reflection Delegate	2,202594	29.323,287	1.224,904
Reflection Delegate Static	1,958067	27.730,138	1.087,511
Reflection Flags	97,242559	1.506.979,540	54.267,544
Reflection Static	92,630929	1.429.110,845	51.727,083
Reflection Static Flags	92,815287	1.431.378,343	51.815,033

Table A.165: Table showing the elapsed time and energy measurement for each Invocation Reflection.

Elapsed Time *p*-Values	Reflection	Reflection Static	Reflection Flags	Reflection Static Flags	Reflection Delegate	Reflection Delegate Static
Reflection	-	<0,05	0,859	<0,05	<0,05	<0,05
Reflection Static	<0,05	-	<0,05	0,340	<0,05	<0,05
Reflection Flags	0,859	<0,05	-	<0,05	<0,05	<0,05
Reflection Static Flags	<0,05	0,340	<0,05	-	<0,05	<0,05
Reflection Delegate	<0,05	<0,05	<0,05	<0,05	-	<0,05
Reflection Delegate Static	<0,05	<0,05	<0,05	<0,05	<0,05	-

Table A.166: Table showing the *p*-values for the group Invocation Reflection with regards to Elapsed Time.

Package Energy *p*-Values	Reflection	Reflection Static	Reflection Flags	Reflection Static Flags	Reflection Delegate	Reflection Delegate Static
Reflection	-	<0,05	0,053	<0,05	<0,05	<0,05
Reflection Static	<0,05	-	<0,05	0,607	<0,05	<0,05
Reflection Flags	0,053	<0,05	-	<0,05	<0,05	<0,05
Reflection Static Flags	<0,05	0,607	<0,05	-	<0,05	<0,05
Reflection Delegate	<0,05	<0,05	<0,05	<0,05	-	<0,05
Reflection Delegate Static	<0,05	<0,05	<0,05	<0,05	<0,05	-

Table A.167: Table showing the *p*-values for the group Invocation Reflection with regards to Package Energy.

DRAM Energy *p*-Values	Reflection	Reflection Static	Reflection Flags	Reflection Static Flags	Reflection Delegate	Reflection Delegate Static
Reflection	-	<0,05	0,726	<0,05	<0,05	<0,05
Reflection Static	<0,05	-	<0,05	0,431	<0,05	<0,05
Reflection Flags	0,726	<0,05	-	<0,05	<0,05	<0,05
Reflection Static Flags	<0,05	0,431	<0,05	-	<0,05	<0,05
Reflection Delegate	<0,05	<0,05	<0,05	<0,05	-	<0,05
Reflection Delegate Static	<0,05	<0,05	<0,05	<0,05	<0,05	-

Table A.168: Table showing the *p*-values for the group Invocation Reflection with regards to DRAM Energy.

A.3.10 Field Mutability

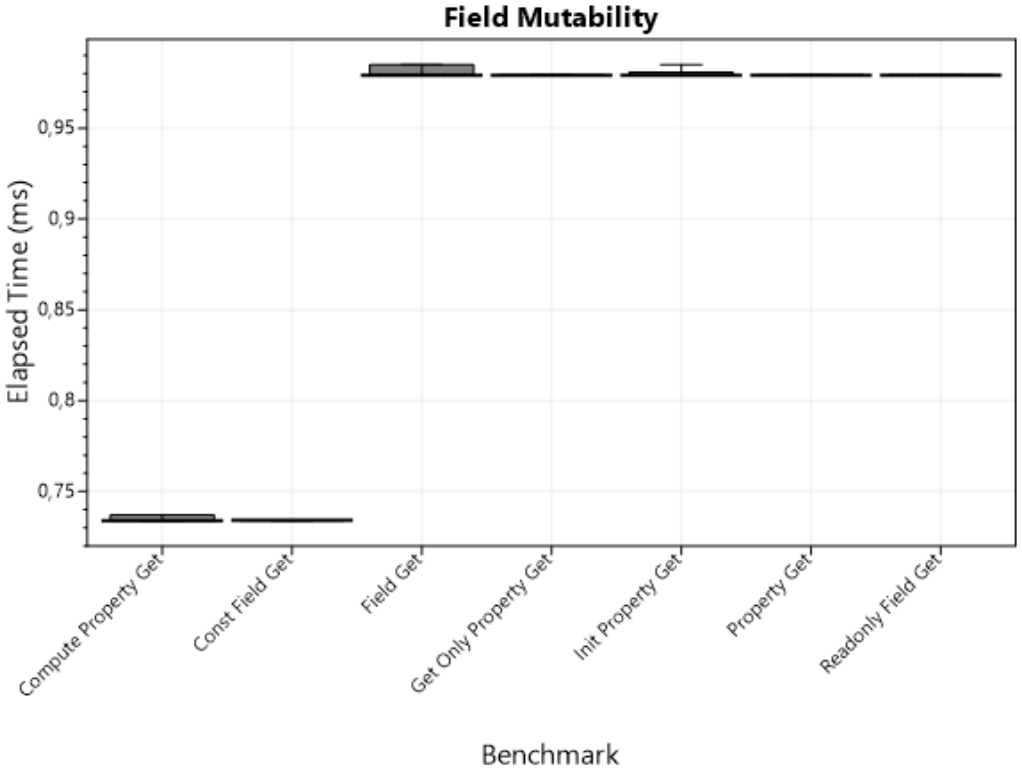

Figure A.132: Boxplot showing how efficient different Field Mutability types are with regards to Elapsed Time.

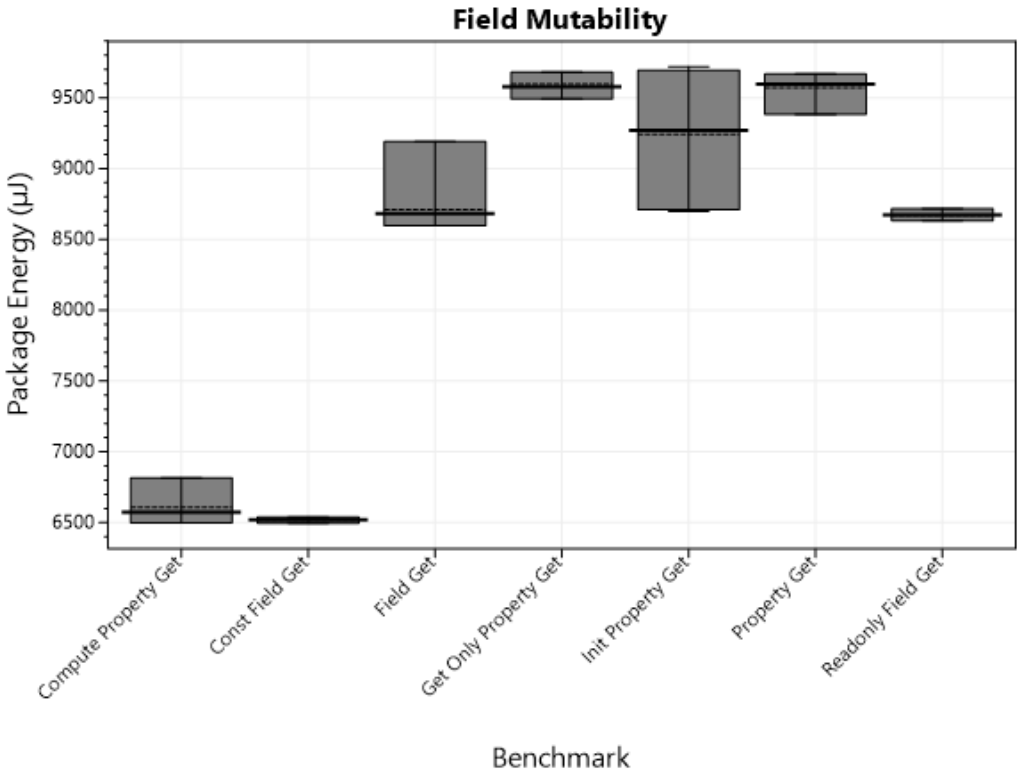

Figure A.133: Boxplot showing how efficient different Field Mutability types are with regards to Package Energy.

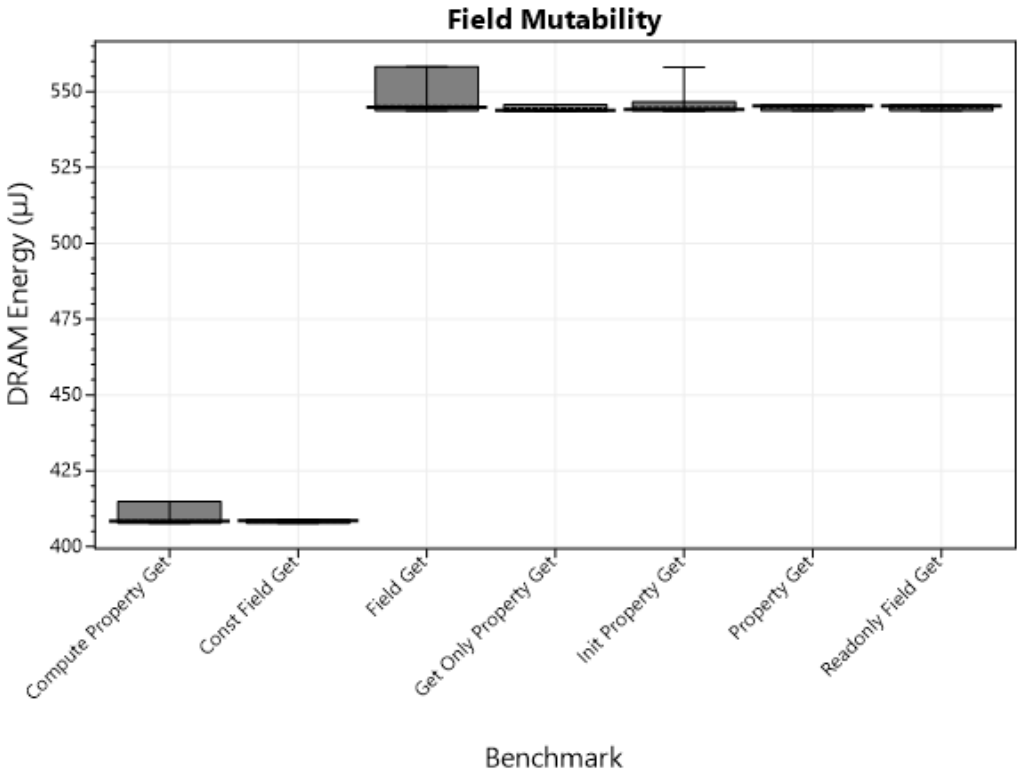

Figure A.134: Boxplot showing how efficient different Field Mutability types are with regards to Dram Energy.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Compute Property Get	0,734152	6.608,547	408,518
Const Field Get	0,734051	6.519,069	408,322
Field Get	0,979240	8.708,617	545,210
Get Only Property Get	0,979024	9.595,407	544,523
Init Property Get	0,979093	9.241,962	544,770
Property Get	0,979021	9.570,589	544,720
Readonly Field Get	0,979032	8.669,423	544,685

Table A.169: Table showing the elapsed time and energy measurement for each Field Mutability.

Elapsed Time *p*-Values	Field Get	Const Field Get	Readonly Field Get	Property Get	Init Property Get	Get Only Property Get	Compute Property Get
Field Get	-	<0,05	0,613	0,556	0,458	0,600	<0,05
Const Field Get	<0,05	-	<0,05	<0,05	<0,05	<0,05	0,598
Readonly Field Get	0,613	<0,05	-	0,723	0,797	0,807	<0,05
Property Get	0,556	<0,05	0,723	-	0,737	0,916	<0,05
Init Property Get	0,458	<0,05	0,797	0,737	-	0,770	<0,05
Get Only Property Get	0,600	<0,05	0,807	0,916	0,770	-	<0,05
Compute Property Get	<0,05	0,598	<0,05	<0,05	<0,05	<0,05	-

Table A.170: Table showing the *p*-values for the group Field Mutability with regards to Elapsed Time.

Package Energy *p*-Values	Field Get	Const Field Get	Readonly Field Get	Property Get	Init Property Get	Get Only Property Get	Compute Property Get
Field Get	-	<0,05	0,313	<0,05	<0,05	<0,05	<0,05
Const Field Get	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Readonly Field Get	0,313	<0,05	-	<0,05	<0,05	<0,05	<0,05
Property Get	<0,05	<0,05	<0,05	-	<0,05	0,477	<0,05
Init Property Get	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Get Only Property Get	<0,05	<0,05	<0,05	0,477	<0,05	-	<0,05
Compute Property Get	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-

Table A.171: Table showing the *p*-values for the group Field Mutability with regards to Package Energy.

DRAM Energy *p*-Values	Field Get	Const Field Get	Readonly Field Get	Property Get	Init Property Get	Get Only Property Get	Compute Property Get
Field Get	-	<0,05	0,597	0,587	0,361	0,492	<0,05
Const Field Get	<0,05	-	<0,05	<0,05	<0,05	<0,05	0,675
Readonly Field Get	0,597	<0,05	-	0,928	0,886	0,703	<0,05
Property Get	0,587	<0,05	0,928	-	0,926	0,638	<0,05
Init Property Get	0,361	<0,05	0,886	0,926	-	0,679	<0,05
Get Only Property Get	0,492	<0,05	0,703	0,638	0,679	-	<0,05
Compute Property Get	<0,05	0,675	<0,05	<0,05	<0,05	<0,05	-

Table A.172: Table showing the *p*-values for the group Field Mutability with regards to DRAM Energy.

A.3.11 Instance Variables

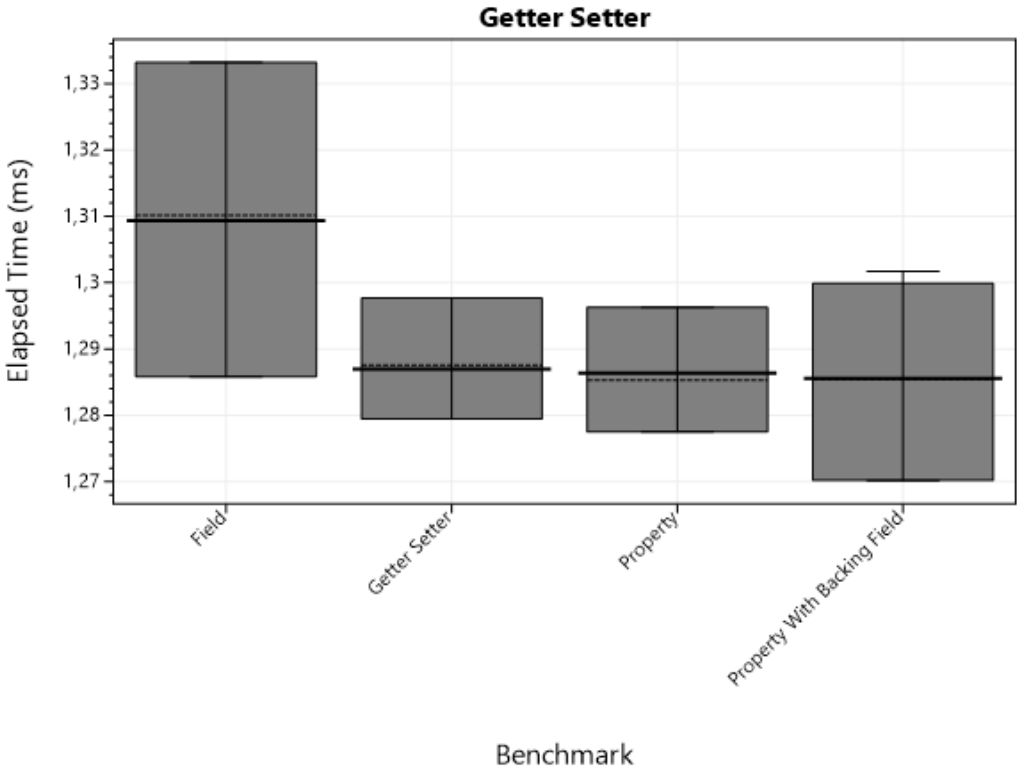

Figure A.135: Boxplot showing how efficient different Getter Setter types are with regards to Elapsed Time.

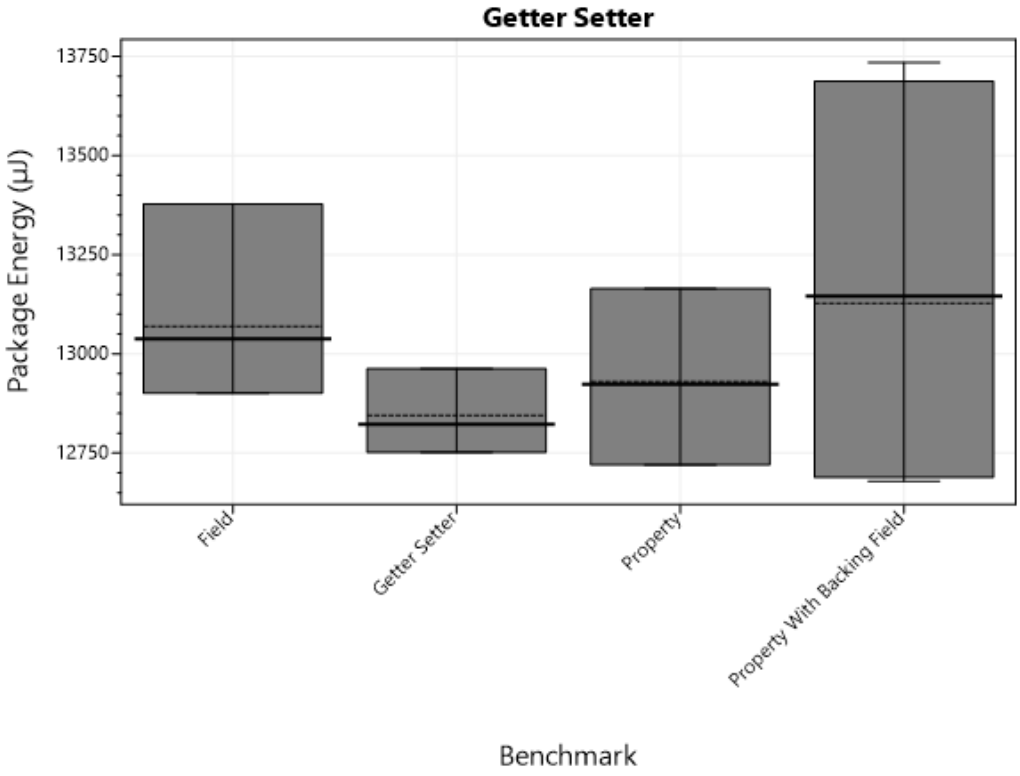

Figure A.136: Boxplot showing how efficient different Getter Setter types are with regards to Package Energy.

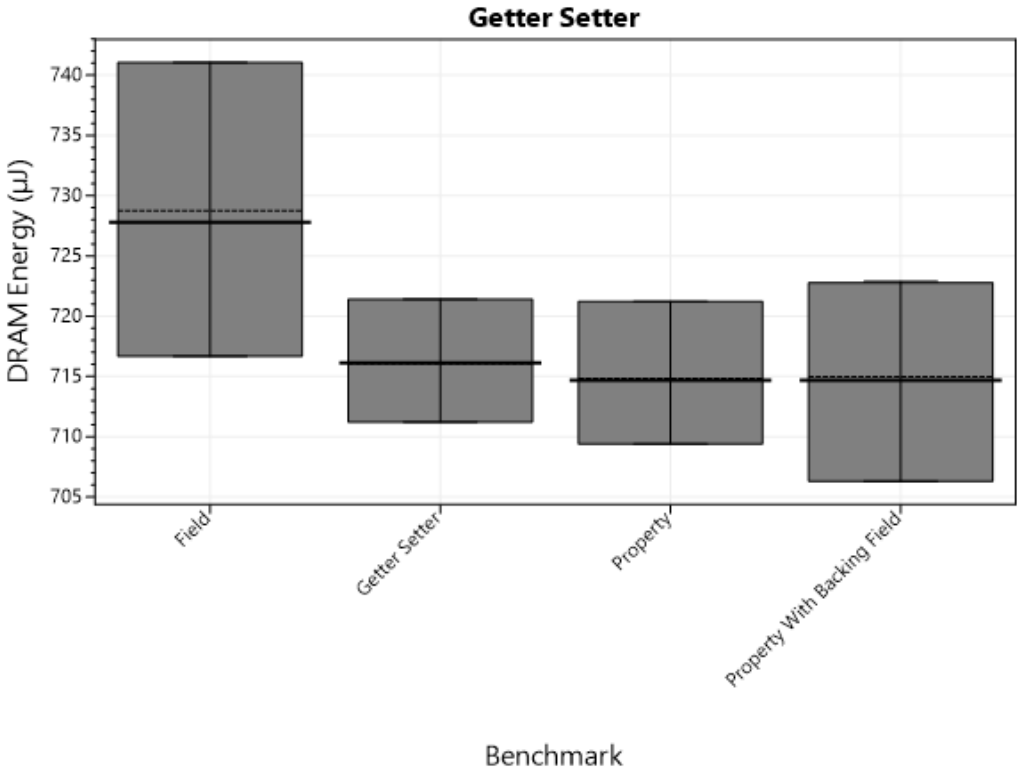

Figure A.137: Boxplot showing how efficient different Getter Setter types are with regards to DRAM Energy.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Field	1,310125	13.069,548	728,750
Getter Setter	1,287545	12.844,276	716,033
Property	1,285305	12.930,155	714,847
Property With Backing Field	1,285421	13.127,632	714,976

Table A.173: Table showing the elapsed time and energy measurement for each Getter Setter.

Elapsed Time *p*-Values	Property	Property With Backing Field	Getter Setter	Field
Property	-	0,951	0,401	<0,05
Property With Backing Field	0,951	-	0,346	<0,05
Getter Setter	0,401	0,346	-	<0,05
Field	<0,05	<0,05	<0,05	-

Table A.174: Table showing the *p*-values for the group Getter Setter with regards to Elapsed Time.

Package Energy *p*-Values	Property	Property With Backing Field	Getter Setter	Field
Property	-	<0,05	0,091	<0,05
Property With Backing Field	<0,05	-	<0,05	0,412
Getter Setter	0,091	<0,05	-	<0,05
Field	<0,05	0,412	<0,05	-

Table A.175: Table showing the *p*-values for the group Getter Setter with regards to Package Energy.

DRAM Energy *p*-Values	Property	Property With Backing Field	Getter Setter	Field
Property	-	0,911	0,464	<0,05
Property With Backing Field	0,911	-	0,426	<0,05
Getter Setter	0,464	0,426	-	<0,05
Field	<0,05	<0,05	<0,05	-

Table A.176: Table showing the *p*-values for the group Getter Setter with regards to DRAM Energy.

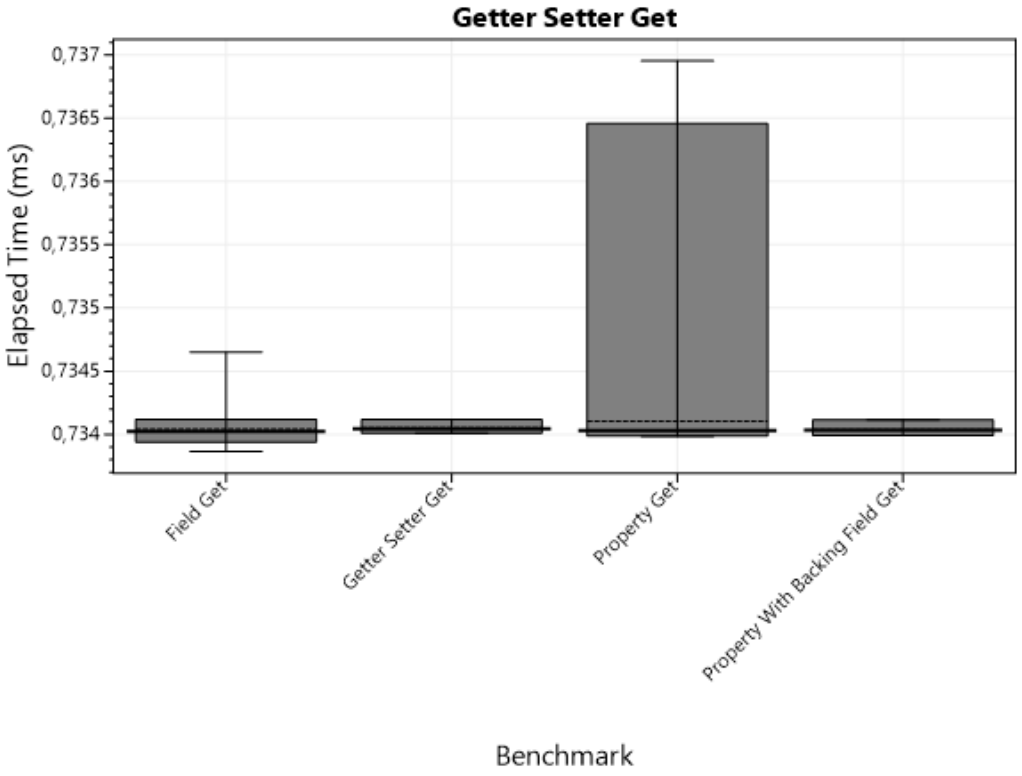

Figure A.138: Boxplot showing how efficient different Getter Setter Get types are with regards to Elapsed Time.

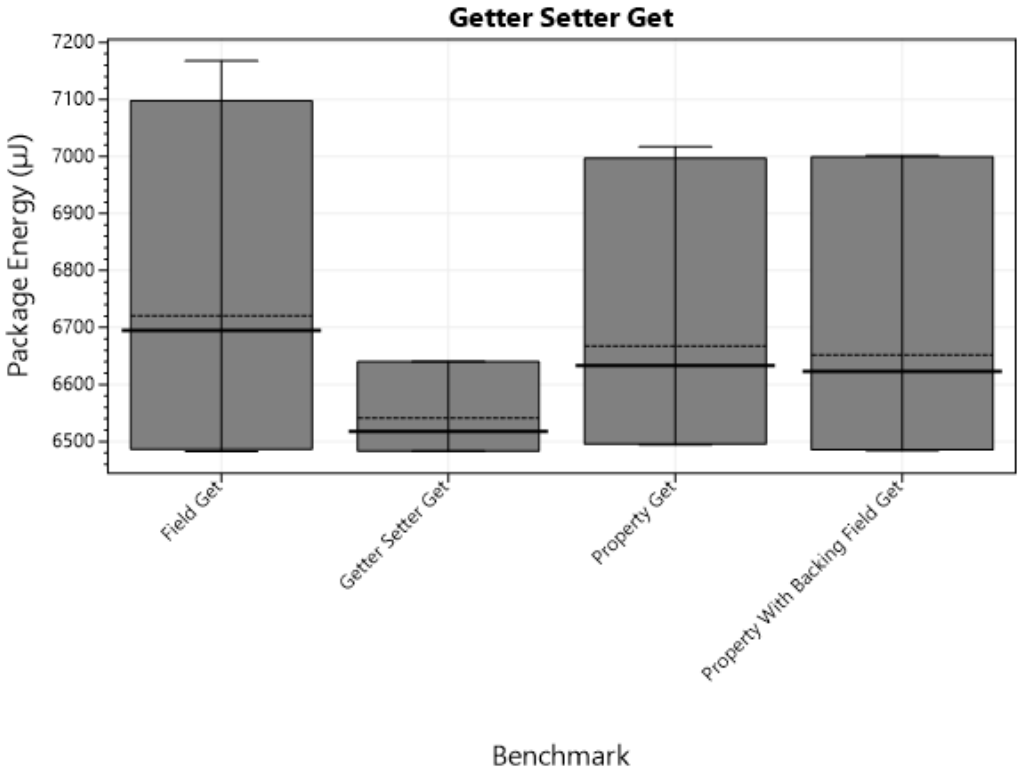

Figure A.139: Boxplot showing how efficient different Getter Setter Get types are with regards to Package Energy.

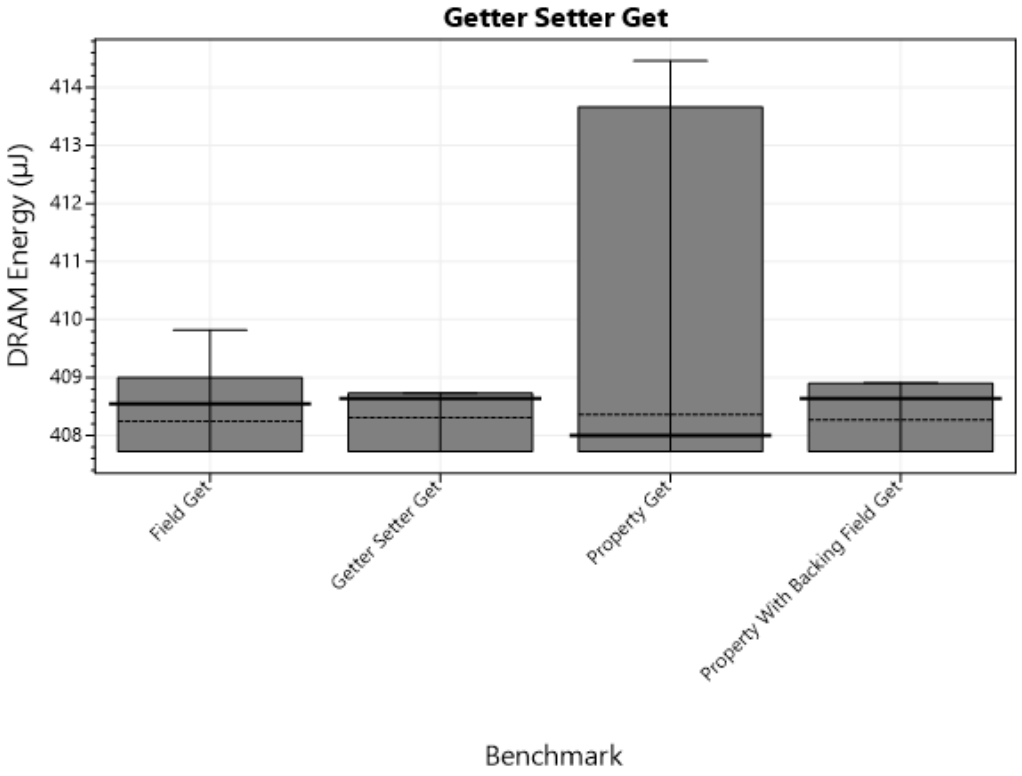

Figure A.140: Boxplot showing how efficient different Getter Setter Get types are with regards to DRAM Energy.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Field Get	0,734040	6.720,844	408,244
Getter Setter Get	0,734051	6.541,315	408,308
Property Get	0,734104	6.667,267	408,364
Property With Backing Field Get	0,734039	6.651,992	408,273

Table A.177: Table showing the elapsed time and energy measurement for each Getter Setter Get.

Elapsed Time *p*-Values	Property Get	Property With Backing Field Get	Getter Setter Get	Field Get
Property Get	-	0,336	0,703	0,150
Property With Backing Field Get	0,336	-	0,303	0,942
Getter Setter Get	0,703	0,303	-	0,627
Field Get	0,150	0,942	0,627	-

Table A.178: Table showing the *p*-values for the group Getter Setter Get with regards to Elapsed Time.

Package Energy *p*-Values	Property Get	Property With Backing Field Get	Getter Setter Get	Field Get
Property Get	-	0,534	<0,05	0,058
Property With Backing Field Get	0,534	-	<0,05	<0,05
Getter Setter Get	<0,05	<0,05	-	<0,05
Field Get	0,058	<0,05	<0,05	-

Table A.179: Table showing the *p*-values for the group Getter Setter Get with regards to Package Energy.

DRAM Energy *p*-Values	Property Get	Property With Backing Field Get	Getter Setter Get	Field Get
Property Get	-	0,606	0,868	0,338
Property With Backing Field Get	0,606	-	0,827	0,733
Getter Setter Get	0,868	0,827	-	0,680
Field Get	0,338	0,733	0,680	-

Table A.180: Table showing the *p*-values for the group Getter Setter Get with regards to DRAM Energy.

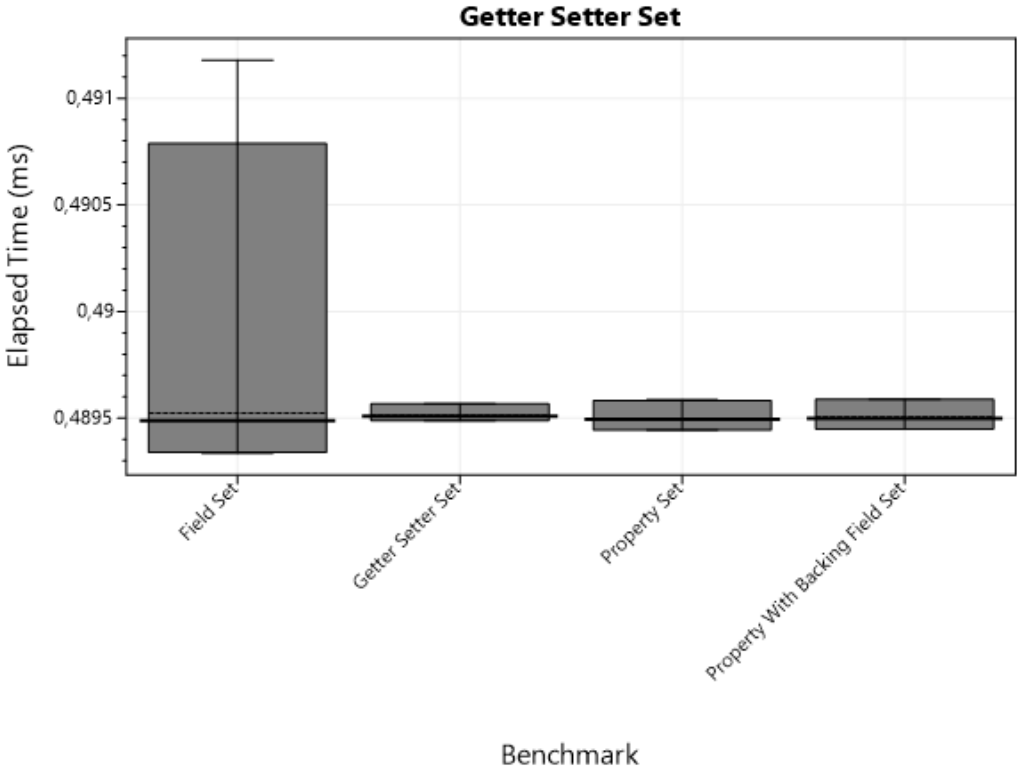

Figure A.141: Boxplot showing how efficient different Getter Setter Set types are with regards to Elapsed Time.

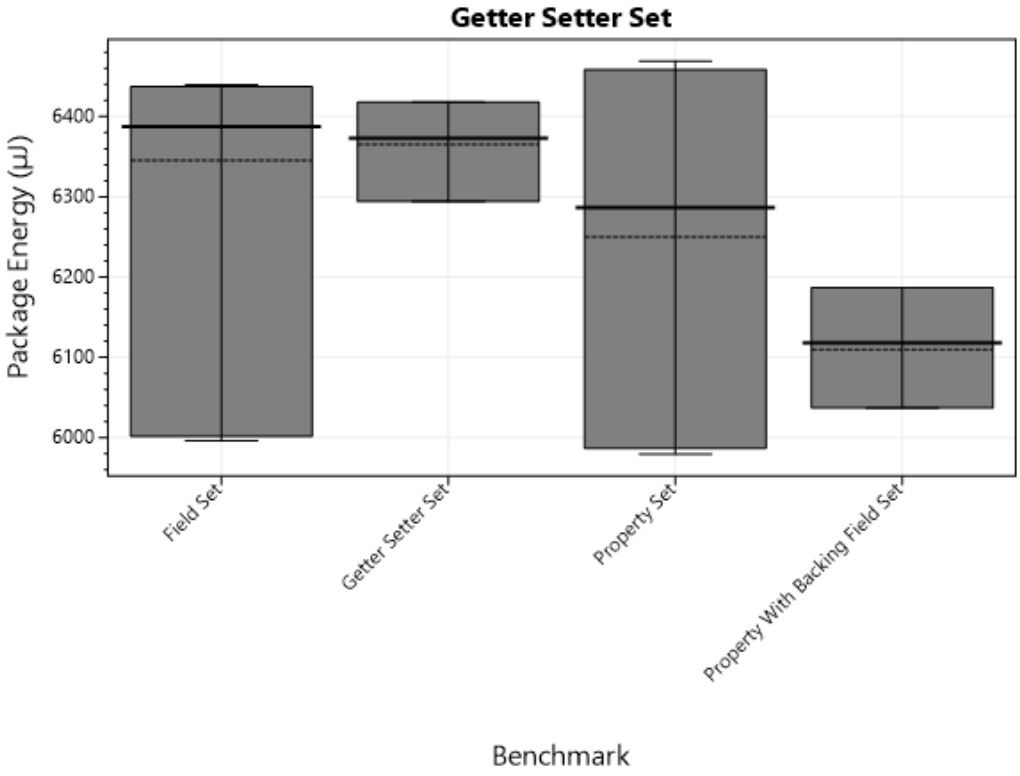

Figure A.142: Boxplot showing how efficient different Getter Setter Set types are with regards to Package Energy.

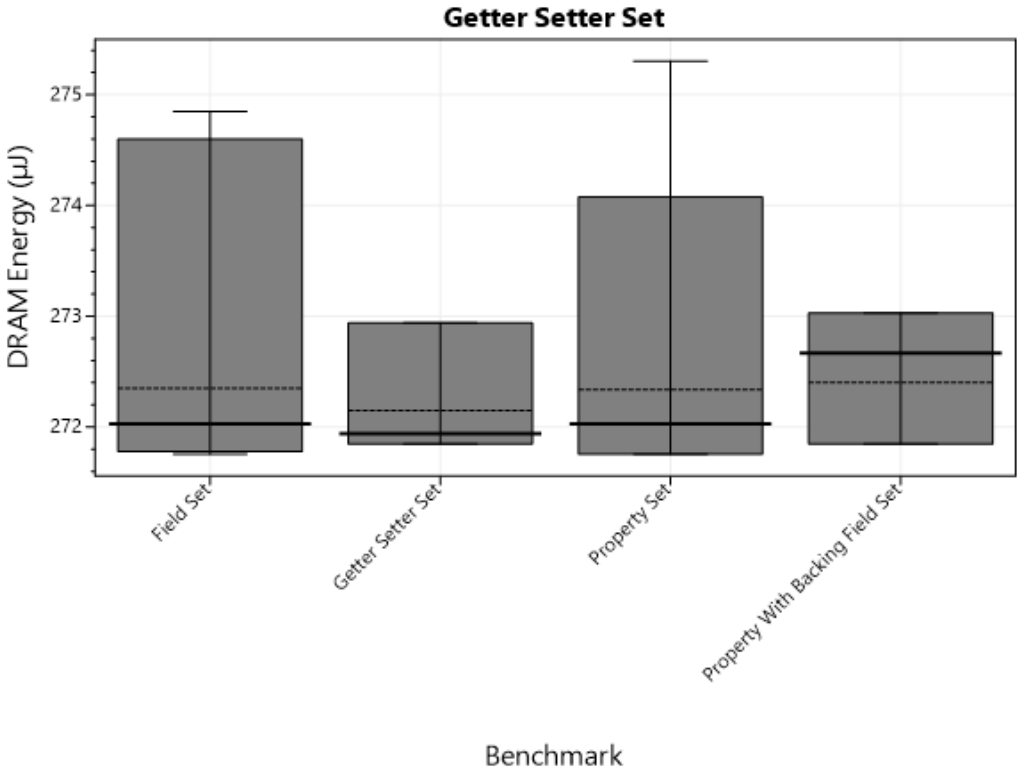

Figure A.143: Boxplot showing how efficient different Getter Setter Set types are with regards to DRAM Energy.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Field Set	0,489524	6.345,259	272,348
Getter Setter Set	0,489514	6.365,194	272,151
Property Set	0,489499	6.250,078	272,339
Property With Backing Field Set	0,489505	6.109,629	272,401

Table A.181: Table showing the elapsed time and energy measurement for each Getter Setter Set.

Elapsed Time *p*-Values	Property Set	Property With Backing Field Set	Getter Setter Set	Field Set
Property Set	-	0,618	0,205	0,427
Property With Backing Field Set	0,618	-	0,495	0,791
Getter Setter Set	0,205	0,495	-	0,910
Field Set	0,427	0,791	0,910	-

Table A.182: Table showing the *p*-values for the group Getter Setter Set with regards to Elapsed Time.

Package Energy *p*-Values	Property Set	Property With Backing Field Set	Getter Setter Set	Field Set
Property Set	-	<0,05	<0,05	<0,05
Property With Backing Field Set	<0,05	-	<0,05	<0,05
Getter Setter Set	<0,05	<0,05	-	0,586
Field Set	<0,05	<0,05	0,586	-

Table A.183: Table showing the *p*-values for the group Getter Setter Set with regards to Package Energy.

DRAM Energy *p*-Values	Property Set	Property With Backing Field Set	Getter Setter Set	Field Set
Property Set	-	0,730	0,351	0,934
Property With Backing Field Set	0,730	-	0,208	0,775
Getter Setter Set	0,351	0,208	-	0,338
Field Set	0,934	0,775	0,338	-

Table A.184: Table showing the *p*-values for the group Getter Setter Set with regards to DRAM Energy.

A.3.12 Objects

Creation

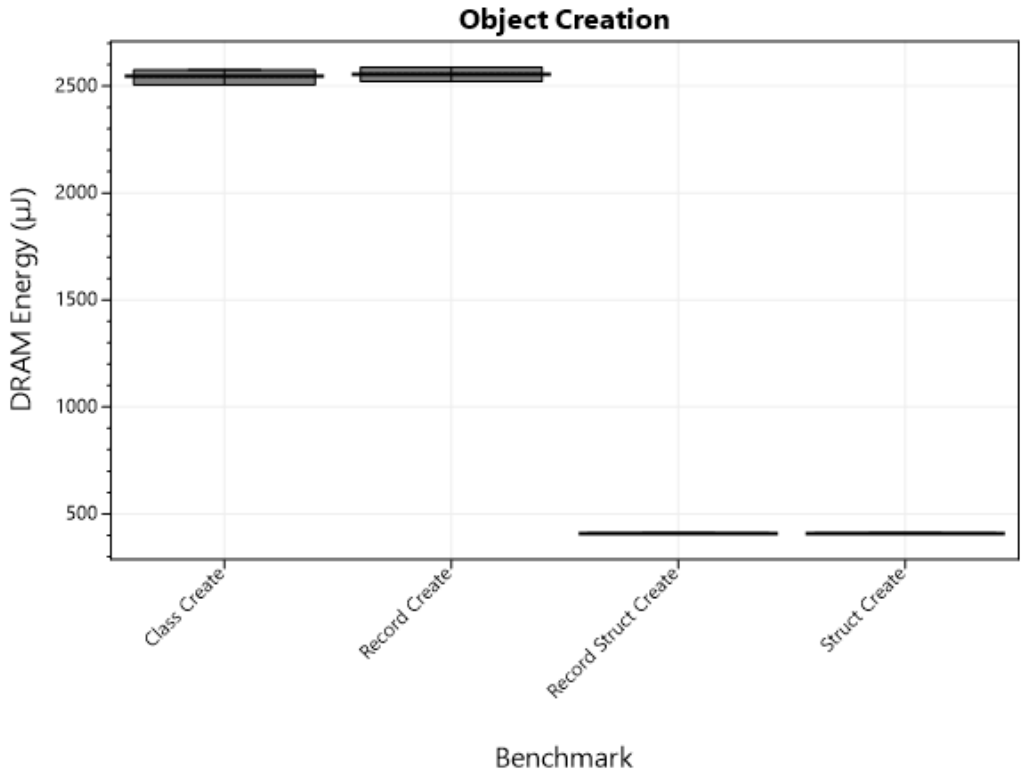

Figure A.144: Boxplot showing how efficient different Object Creation types are with regards to DRAM Energy.

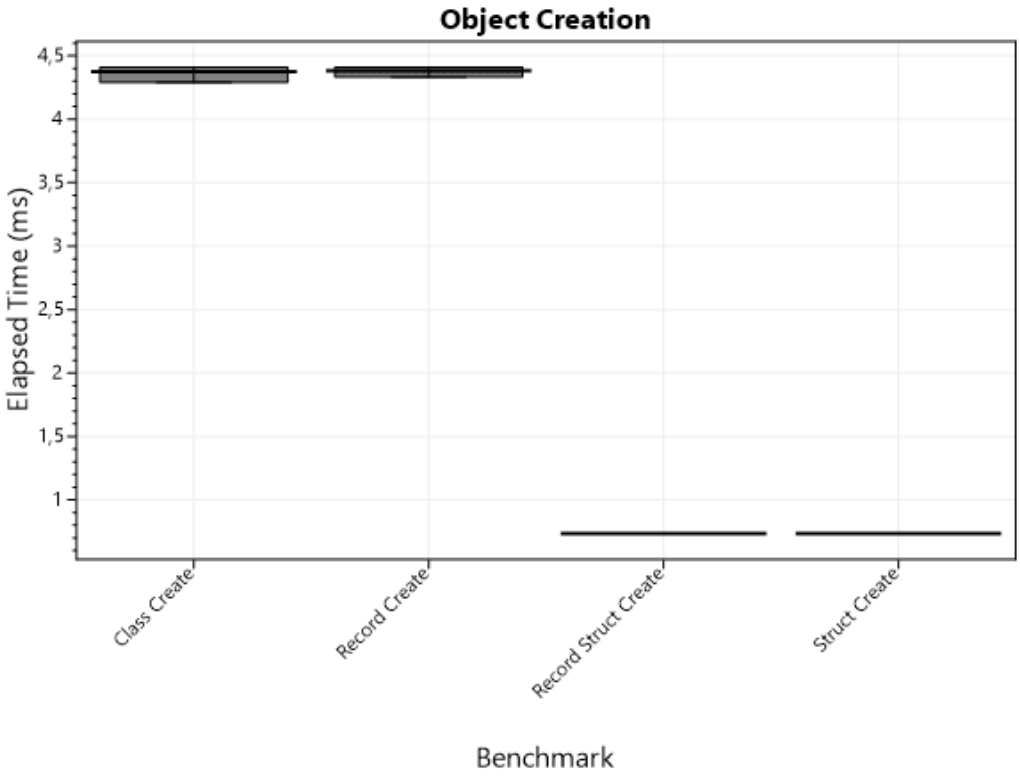

Figure A.145: Boxplot showing how efficient different Object Creation types are with regards to Elapsed Time.

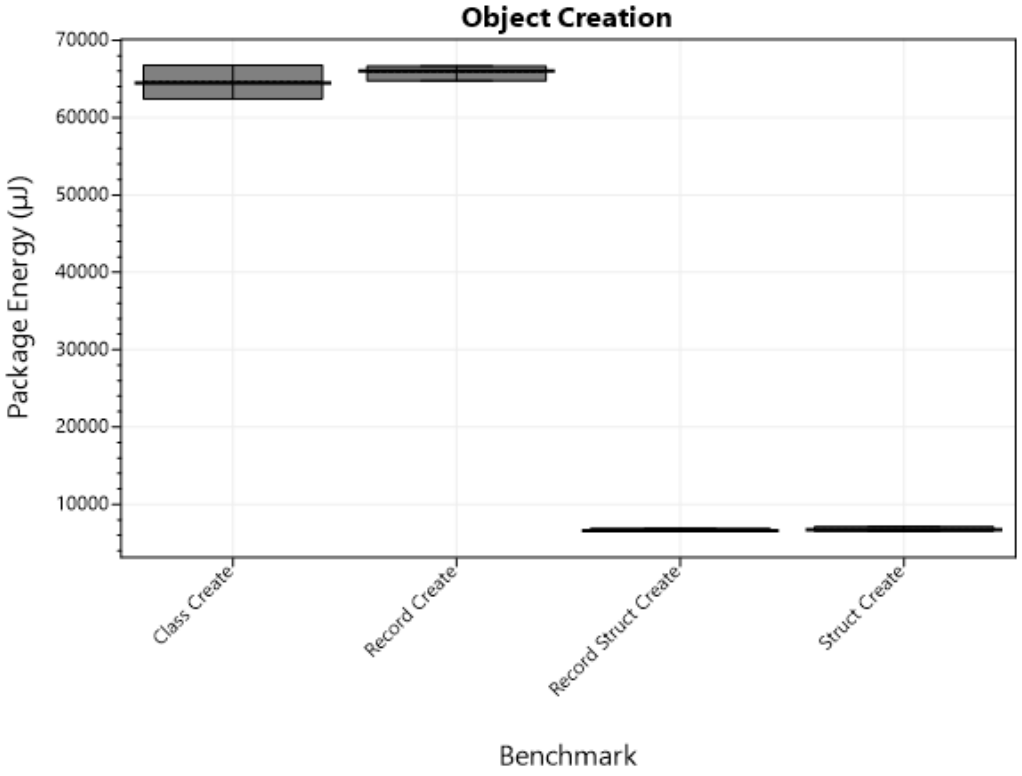

Figure A.146: Boxplot showing how efficient different Object Creation types are with regards to Package Energy.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Class Create	4,370712	64.624,107	2.544,209
Record Create	4,378273	65.926,919	2.556,278
Record Struct Create	0,734039	6.605,921	408,283
Struct Create	0,734024	6.748,895	408,236

Table A.185: Table showing the elapsed time and energy measurement for each Object Creation.

Elapsed Time *p*-Values	Class Create	Struct Create	Record Create	Record Struct Create
Class Create	-	<0,05	0,441	<0,05
Struct Create	<0,05	-	<0,05	0,519
Record Create	0,441	<0,05	-	<0,05
Record Struct Create	<0,05	0,519	<0,05	-

Table A.186: Table showing the *p*-values for the group Object Creation with regards to Elapsed Time.

Package Energy *p*-Values	Class Create	Struct Create	Record Create	Record Struct Create
Class Create	-	<0,05	<0,05	<0,05
Struct Create	<0,05	-	<0,05	<0,05
Record Create	<0,05	<0,05	-	<0,05
Record Struct Create	<0,05	<0,05	<0,05	-

Table A.187: Table showing the *p*-values for the group Object Creation with regards to Package Energy.

DRAM Energy *p*-Values	Class Create	Struct Create	Record Create	Record Struct Create
Class Create	-	<0,05	0,062	<0,05
Struct Create	<0,05	-	<0,05	0,624
Record Create	0,062	<0,05	-	<0,05
Record Struct Create	<0,05	0,624	<0,05	-

Table A.188: Table showing the *p*-values for the group Object Creation with regards to DRAM Energy.

Invocation

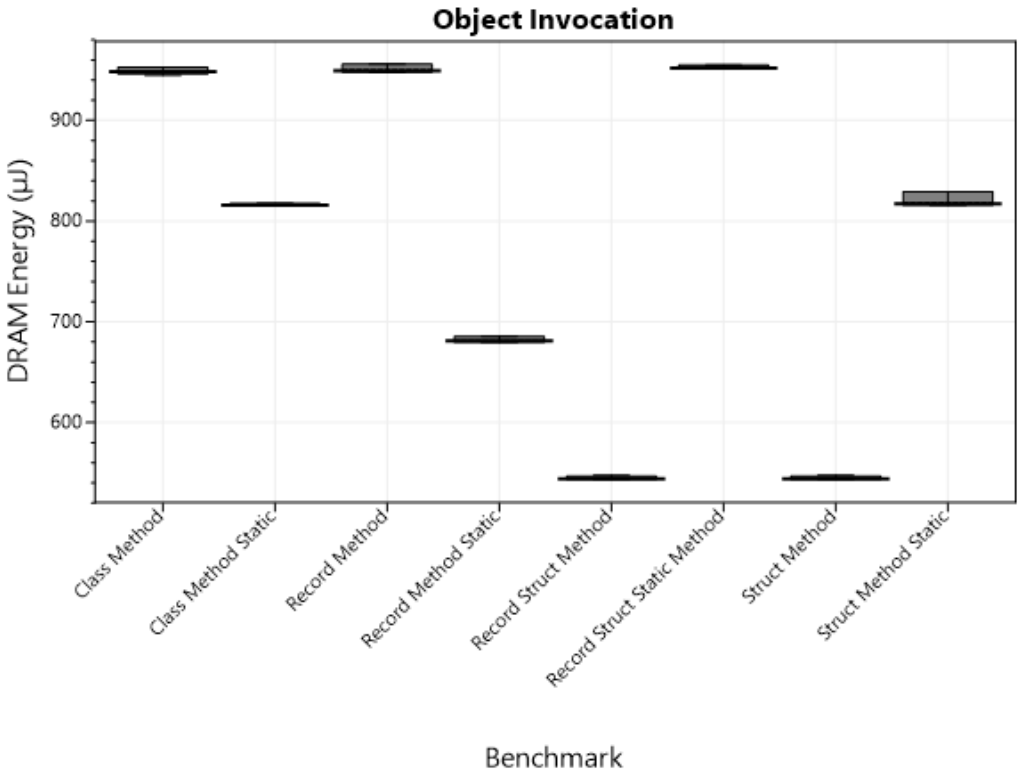

Figure A.147: Boxplot showing how efficient different Object Invocation types are with regards to DRAM Energy.

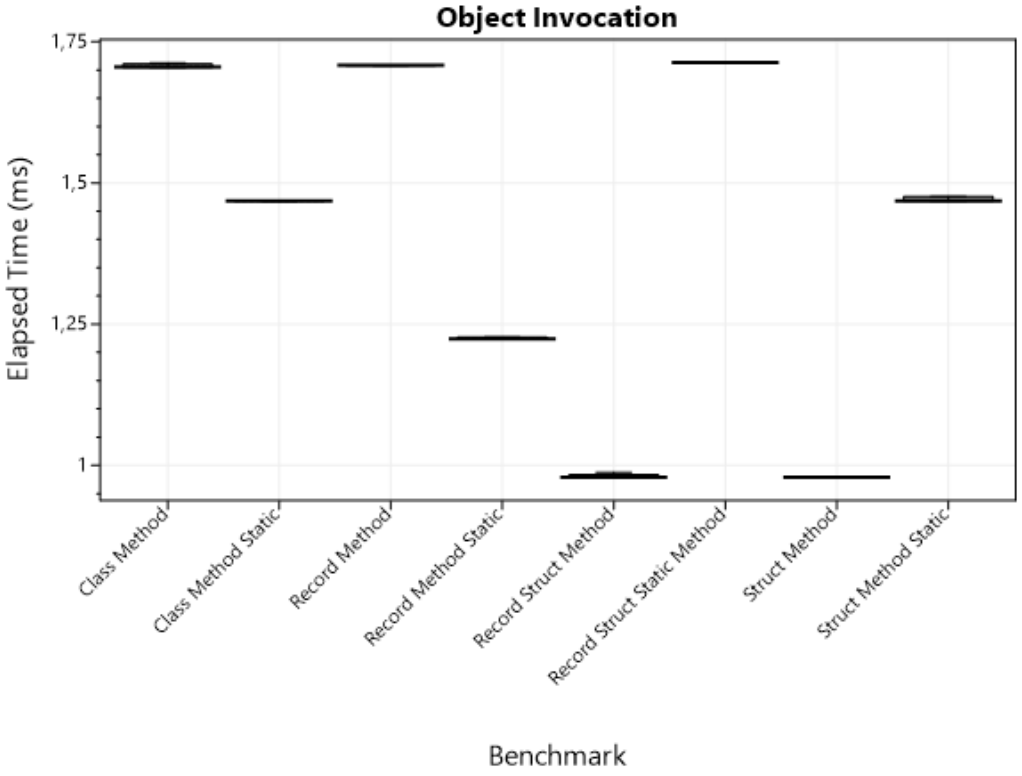

Figure A.148: Boxplot showing how efficient different Object Invocation types are with regards to Elapsed Time.

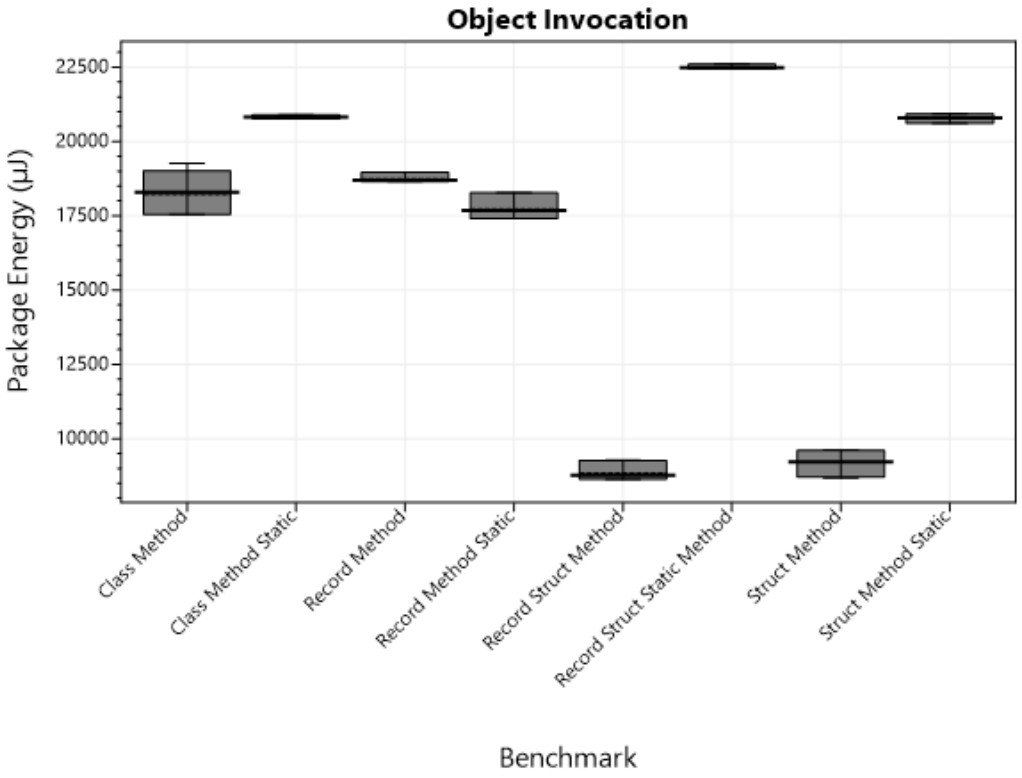

Figure A.149: Boxplot showing how efficient different Object Invocation types are with regards to Package Energy.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Class Method	1,705729	18.222,192	948,875
Class Method Static	1,468121	20.817,694	816,381
Record Method	1,708151	18.740,392	950,319
Record Method Static	1,223811	17.721,990	681,006
Record Struct Static Method	1,713339	22.498,618	952,704
Record Struct Method	0,979123	8.842,624	544,515
Struct Method	0,978953	9.201,011	544,441
Struct Method Static	1,468980	20.776,040	818,018

Table A.189: Table showing the elapsed time and energy measurement for each Object Invocation.

Elapsed Time *p*-Values	Class Method	Class Method Static	Struct Method	Struct Method Static
Class Method	-	<0,05	<0,05	<0,05
Class Method Static	<0,05	-	<0,05	0,3475
Struct Method	<0,05	<0,05	-	<0,05
Struct Method Static	<0,05	0,347	<0,05	-
Record Method	<0,05	<0,05	<0,05	<0,05
Record Method Static	<0,05	<0,05	<0,05	<0,055
Record Struct Method	<0,05	<0,05	0,154	<0,0
Record Struct Static Method	<0,05	<0,05	<0,05	<0,05

Elapsed Time *p*-Values	Record Method	Record Method Static	Record Struct Method	Record Struct Static Method
Class Method	<0,05	<0,05	<0,05	<0,05
Class Method Static	<0,05	<0,05	<0,05	<0,05
Struct Method	<0,05	<0,05	0,154	<0,05
Struct Method Static	<0,05	<0,05	<0,05	<0,05
Record Method	-	<0,05	<0,05	<0,05
Record Method Static	<0,05	-	<0,05	<0,05
Record Struct Method	<0,05	<0,05	-	<0,05
Record Struct Static Method	<0,05	<0,05	<0,05	-

Table A.190: Table showing the *p*-values for the group Object Invocation with regards to Elapsed Time.

Package Energy *p*-Values	Class Method	Class Method Static	Struct Method	Struct Method Static
Class Method	-	<0,05	<0,05	<0,05
Class Method Static	<0,05	-	<0,05	0,287
Struct Method	<0,05	<0,05	-	<0,05
Struct Method Static	<0,05	0,287	<0,05	-
Record Method	<0,05	<0,05	<0,05	<0,05
Record Method Static	<0,05	<0,05	<0,05	<0,05
Record Struct Method	<0,05	<0,05	<0,05	<0,05
Record Struct Static Method	<0,05	<0,05	<0,05	<0,05

Package Energy *p*-Values	Record Method	Record Method Static	Record Struct Method	Record Struct Static Method
Class Method	<0,05	<0,05	<0,05	<0,05
Class Method Static	<0,05	<0,05	<0,05	<0,05
Struct Method	<0,05	<0,05	<0,05	<0,05
Struct Method Static	<0,05	<0,05	<0,05	<0,05
Record Method	-	<0,05	<0,05	<0,05
Record Method Static	<0,05	-	<0,05	<0,05
Record Struct Method	<0,05	<0,05	-	<0,05
Record Struct Static Method	<0,05	<0,05	<0,05	-

Table A.191: Table showing the *p*-values for the group Object Invocation with regards to Package Energy part 2.

DRAM Energy *p*-Values	Class Method	Class Method Static	Struct Method	Struct Method Static	Record Method	Record Method Static	Record Struct Method	Record Struct Static Method
Class Method	-	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05
Class Method Static	<0,05	-	<0,05	0,280	<0,05	<0,05	<0,05	<0,05
Struct Method	<0,05	<0,05	-	<0,05	<0,05	<0,05	0,634	<0,05
Struct Method Static	<0,05	0,280	<0,05	-	<0,05	<0,05	<0,05	<0,05
Record Method	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05	<0,05
Record Method Static	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Record Struct Method	<0,05	<0,05	0,634	<0,05	<0,05	<0,05	-	<0,05
Record Struct Static Method	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-

Table A.192: Table showing the *p*-values for the group Object Invocation with regards to DRAM Energy.

Field Access

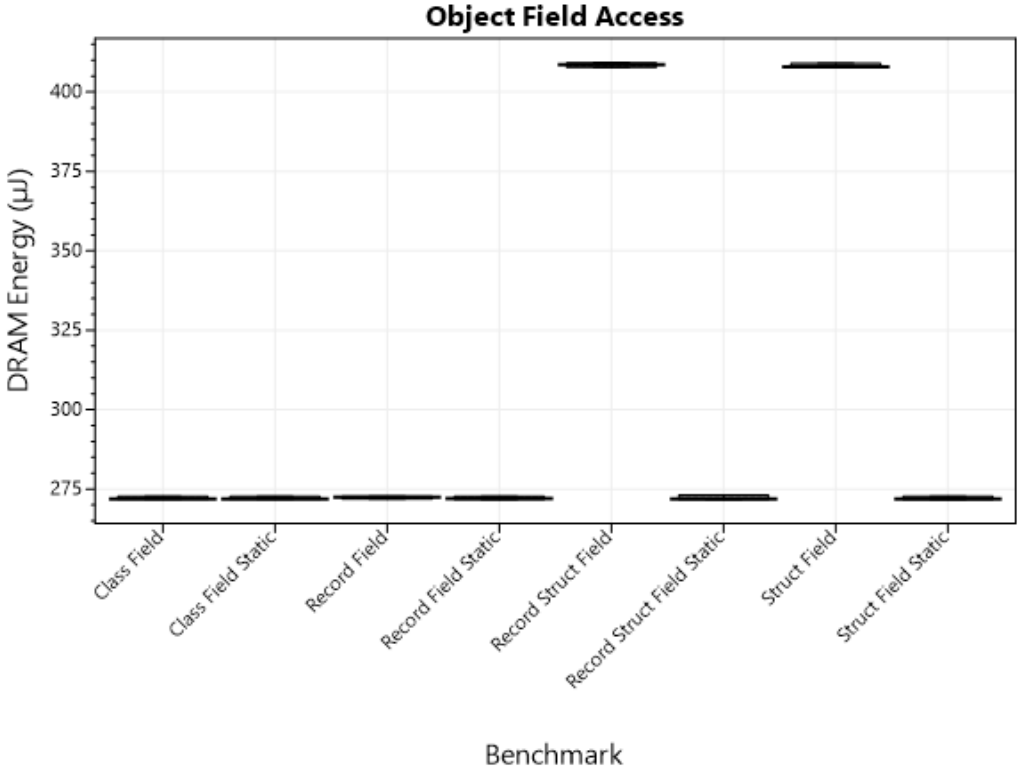

Figure A.150: Boxplot showing how efficient different Object Field Access types are with regards to DRAM Energy.

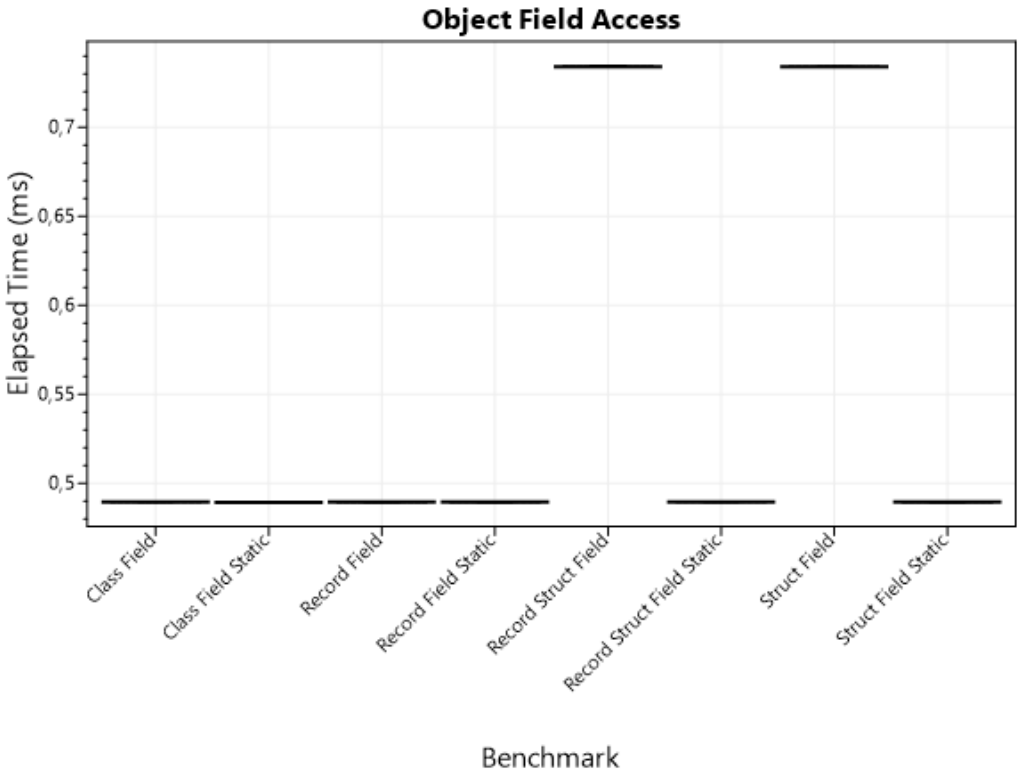

Figure A.151: Boxplot showing how efficient different Object Field Access types are with regards to Elapsed Time.

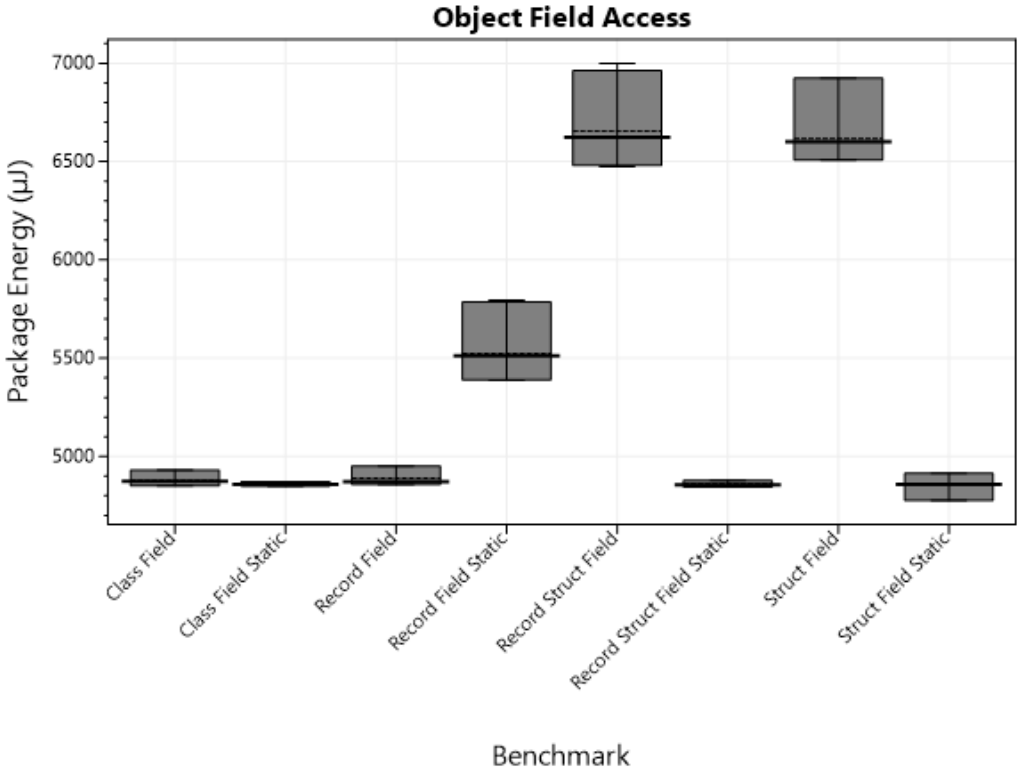

Figure A.152: Boxplot showing how efficient different Object Field Access types are with regards to Package Energy.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Class Field	0,489496	4.878,254	272,181
Class Field Static	0,489445	4.858,751	271,979
Record Field	0,489500	4.889,176	272,329
Record Field Static	0,489473	5.522,367	272,244
Record Struct Field	0,734047	6.653,983	408,301
Record Struct Field Static	0,489498	4.859,418	272,151
Struct Field	0,734010	6.616,529	408,233
Struct Field Static	0,489489	4.856,244	272,047

Table A.193: Table showing the elapsed time and energy measurement for each Object Field Access.

Elapsed Time *p*-Values	Class Field	Class Field Static	Struct Field	Struct Field Static	Record Field	Record Field Static	Record Struct Field	Record Struct Field Static
Class Field	-	<0,05	<0,05	0,766	0,862	0,083	<0,05	0,941
Class Field Static	<0,05	-	<0,05	0,103	<0,05	<0,05	<0,05	<0,05
Struct Field	<0,05	<0,05	-	<0,05	<0,05	<0,05	<0,05	<0,05
Struct Field Static	0,766	0,103	<0,05	-	0,653	0,301	<0,05	0,714
Record Field	0,862	<0,05	<0,05	0,653	-	<0,05	<0,05	0,909
Record Field Static	0,083	<0,05	<0,05	0,301	<0,05	-	<0,05	0,056
Record Struct Field	<0,05	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05
Record Struct Field Static	0,941	<0,05	<0,05	0,714	0,909	0,056	<0,05	-

Table A.194: Table showing the *p*-values for the group Object Field Access with regards to Elapsed Time.

Package Energy *p*-Values	Class Field	Class Field Static	Struct Field	Struct Field Static	Record Field	Record Field Static	Record Struct Field	Record Struct Field Static
Class Field	-	<0,05	<0,05	0,175	0,426	<0,05	<0,05	<0,05
Class Field Static	<0,05	-	<0,05	0,859	<0,05	<0,05	<0,05	0,887
Struct Field	<0,05	<0,05	-	<0,05	<0,05	<0,05	0,194	<0,05
Struct Field Static	0,175	0,859	<0,05	-	0,064	<0,05	<0,05	0,825
Record Field	0,426	<0,05	<0,05	0,064	-	<0,05	<0,05	<0,05
Record Field Static	<0,05	<0,05	<0,05	<0,05	<0,05	-	<0,05	<0,05
Record Struct Field	<0,05	<0,05	0,194	<0,05	<0,05	<0,05	-	<0,05
Record Struct Field Static	<0,05	0,887	<0,05	0,825	<0,05	<0,05	<0,05	-

Table A.195: Table showing the *p*-values for the group Object Field Access with regards to Package Energy.

DRAM Energy *p*-Values	Class Field	Class Field Static	Struct Field	Struct Field Static	Record Field	Record Field Static	Record Struct Field	Record Struct Field Static
Class Field	-	0,240	<0,05	0,464	0,451	0,678	<0,05	0,893
Class Field Static	0,240	-	<0,05	0,643	<0,05	0,072	<0,05	0,383
Struct Field	<0,05	<0,05	-	<0,05	<0,05	<0,05	0,501	<0,05
Struct Field Static	0,464	0,643	<0,05	-	0,123	0,172	<0,05	0,612
Record Field	0,451	<0,05	<0,05	0,123	-	0,562	<0,05	0,415
Record Field Static	0,678	0,072	<0,05	0,172	0,562	-	<0,05	0,556
Record Struct Field	<0,05	<0,05	0,501	<0,05	<0,05	<0,05	-	<0,05
Record Struct Field Static	0,893	0,383	<0,05	0,612	0,415	0,556	<0,05	-

Table A.196: Table showing the *p*-values for the group Object Field Access with regards to DRAM Energy.

A.3.13 Inheritance

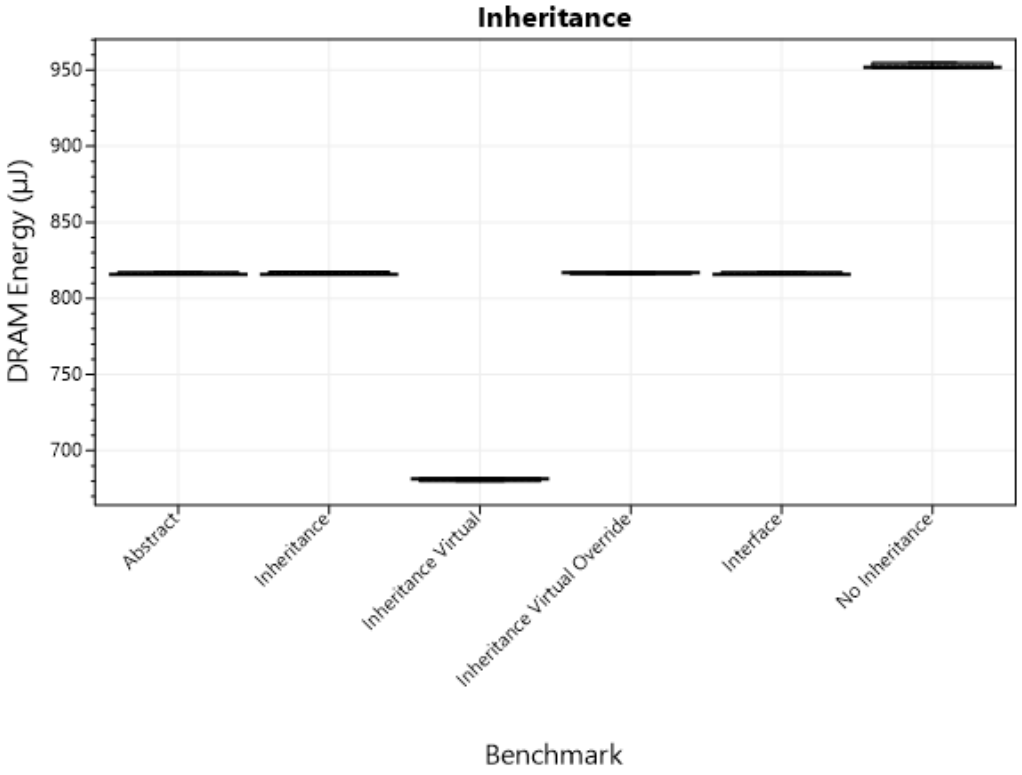

Figure A.153: Boxplot showing how efficient different Inheritance types are with regards to DRAM Energy.

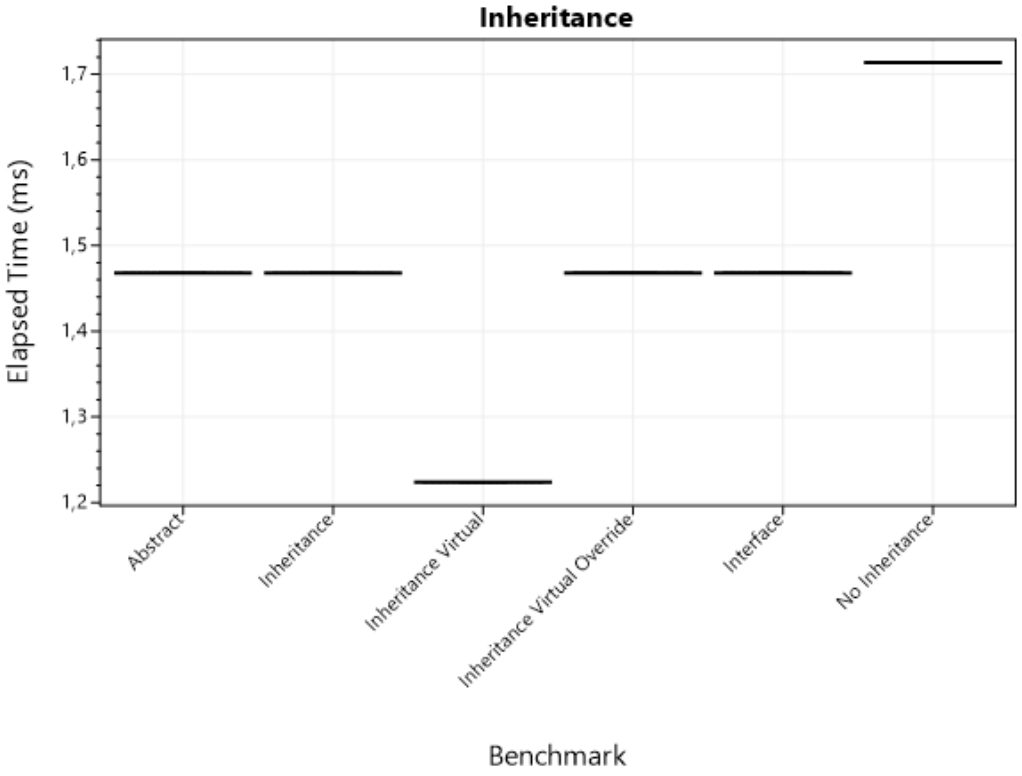

Figure A.154: Boxplot showing how efficient different Inheritance types are with regards to Elapsed Time.

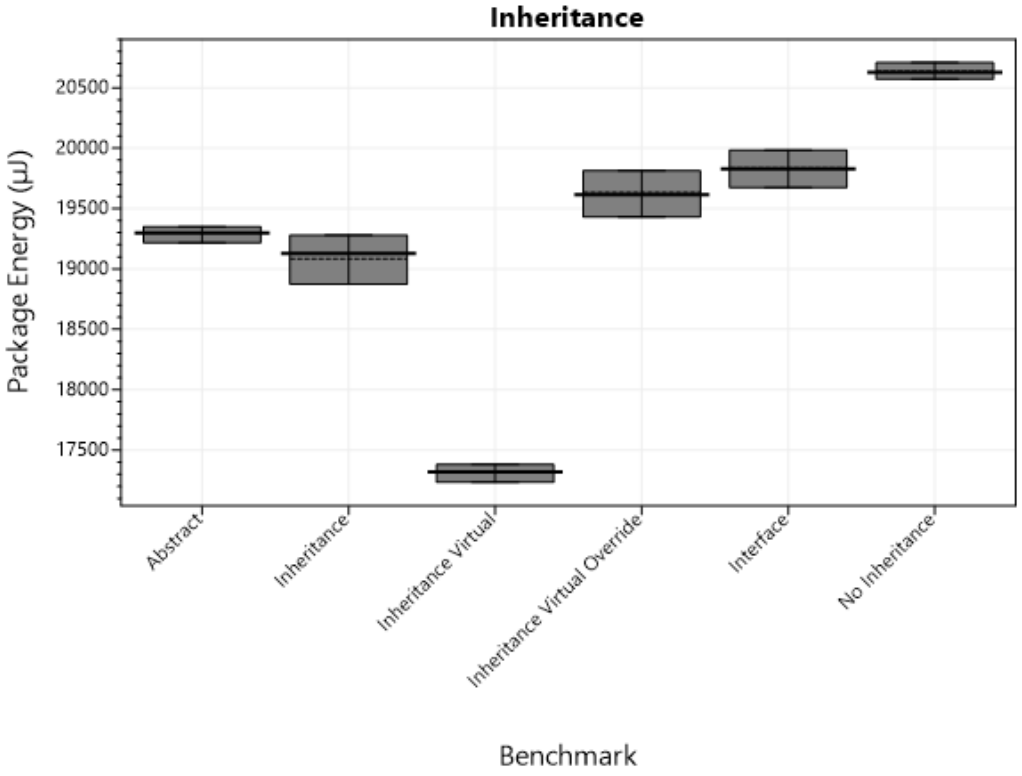

Figure A.155: Boxplot showing how efficient different Inheritance types are with regards to Package Energy.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Abstract	1,467985	19.300,014	816,340
Inheritance	1,468008	19.082,808	816,461
Inheritance Virtual	1,223562	17.313,036	680,684
Inheritance Virtual Override	1,468085	19.632,412	816,560
Interface	1,468055	19.834,493	816,239
No Inheritance	1,713479	20.634,361	952,703

Table A.197: Table showing the elapsed time and energy measurement for each Inheritance.

Elapsed Time *p*-Values	No Inheritance	Inheritance	Abstract	Interface	Inheritance Virtual	Inheritance Virtual Override
No Inheritance	-	<0,05	<0,05	<0,05	<0,05	<0,05
Inheritance	<0,05	-	0,611	0,310	<0,05	<0,05
Abstract	<0,05	0,611	-	0,140	<0,05	<0,05
Interface	<0,05	0,310	0,140	-	<0,05	0,425
Inheritance Virtual	<0,05	<0,05	<0,05	<0,05	-	<0,05
Inheritance Virtual Override	<0,05	<0,05	<0,05	0,425	<0,05	-

Table A.198: Table showing the *p*-values for the group Inheritance with regards to Elapsed Time.

Package Energy *p*-Values	No Inheritance	Inheritance	Abstract	Interface	Inheritance Virtual	Inheritance Virtual Override
No Inheritance	-	<0,05	<0,05	<0,05	<0,05	<0,05
Inheritance	<0,05	-	<0,05	<0,05	<0,05	<0,05
Abstract	<0,05	<0,05	-	<0,05	<0,05	<0,05
Interface	<0,05	<0,05	<0,05	-	<0,05	<0,05
Inheritance Virtual	<0,05	<0,05	<0,05	<0,05	-	<0,05
Inheritance Virtual Override	<0,05	<0,05	<0,05	<0,05	<0,05	-

Table A.199: Table showing the *p*-values for the group Inheritance with regards to Package Energy.

DRAM Energy *p*-Values	No Inheritance	Inheritance	Abstract	Interface	Inheritance Virtual	Inheritance Virtual Override
No Inheritance	-	<0,05	<0,05	<0,05	<0,05	<0,05
Inheritance	<0,05	-	0,793	0,614	<0,05	0,806
Abstract	<0,05	0,793	-	0,820	<0,05	0,591
Interface	<0,05	0,614	0,820	-	<0,05	0,409
Inheritance Virtual	<0,05	<0,05	<0,05	<0,05	-	<0,05
Inheritance Virtual Override	<0,05	0,806	0,591	0,409	<0,05	-

Table A.200: Table showing the *p*-values for the group Inheritance with regards to DRAM Energy.

A.3.14 Generics

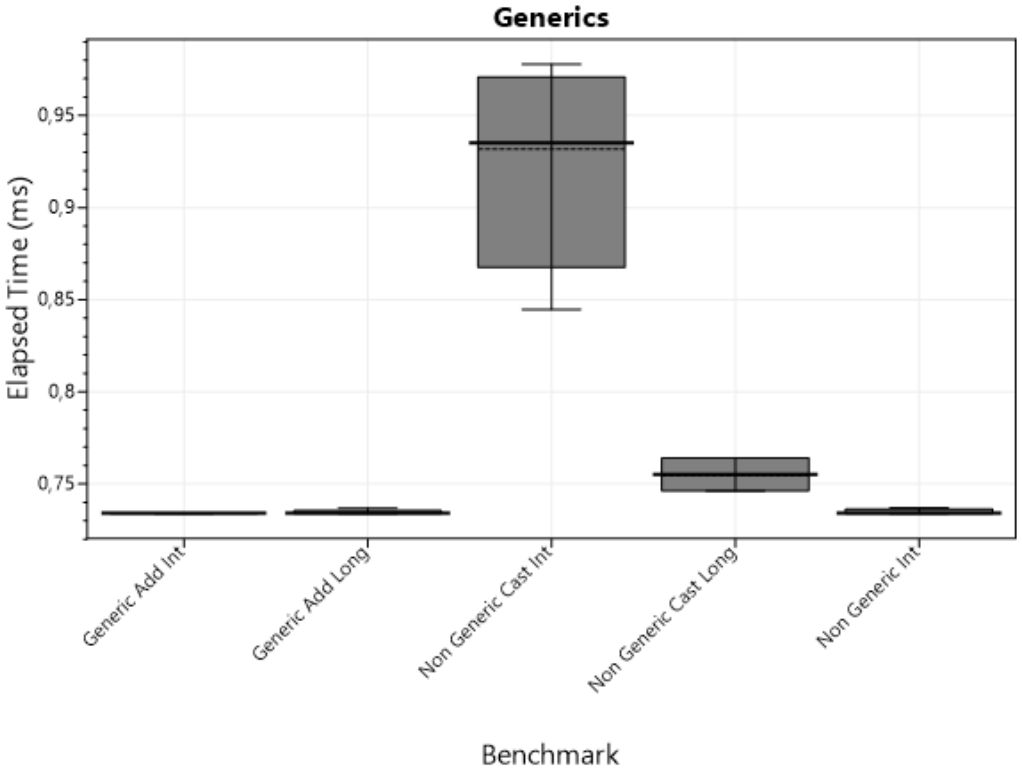

Figure A.156: Boxplot showing how efficient different Generics types are with regards to Elapsed Time.

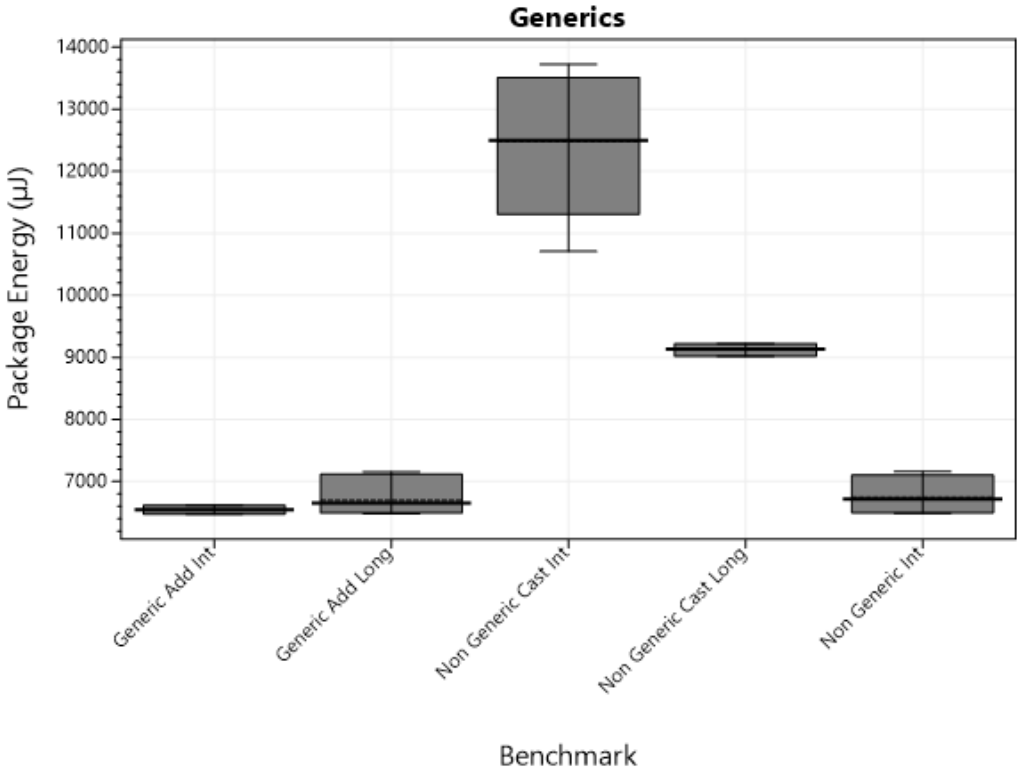

Figure A.157: Boxplot showing how efficient different Generics types are with regards to Package Energy.

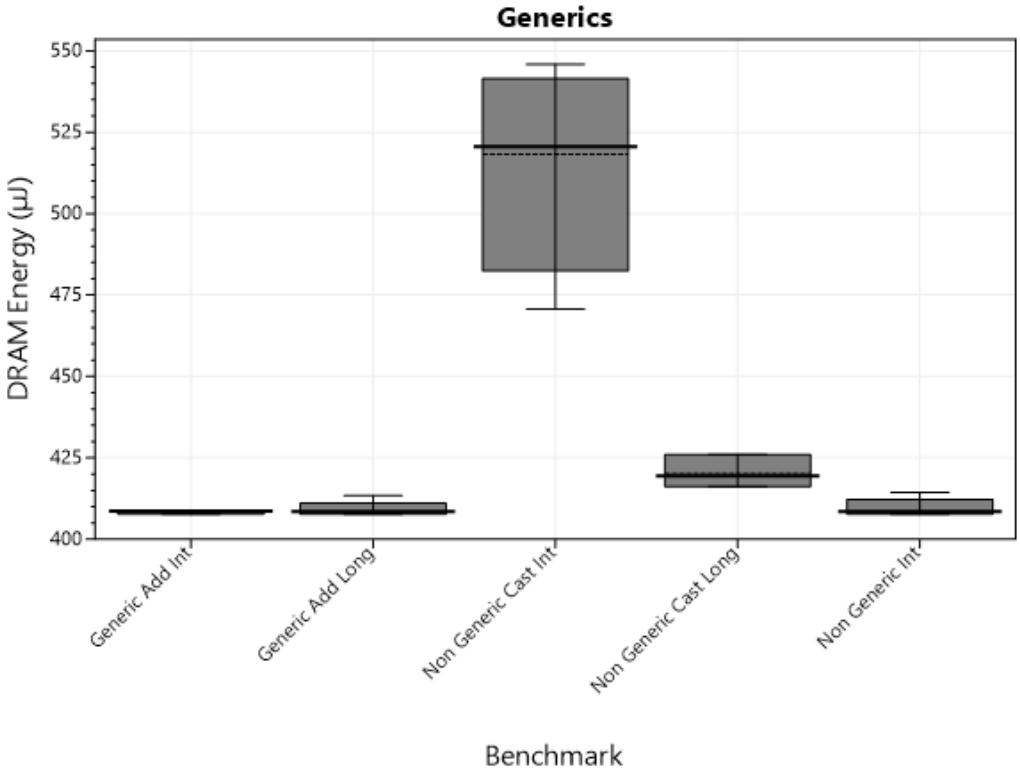

Figure A.158: Boxplot showing how efficient different Generics types are with regards to DRAM Energy.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Generic Add Int	0,734047	6.535,603	408,326
Generic Add Long	0,734099	6.695,250	408,373
Non Generic Cast Int	0,931632	12.482,432	518,253
Non Generic Cast Long	0,754723	9.124,408	420,119
Non Generic Int	0,734115	6.747,542	408,418

Table A.201: Table showing the elapsed time and energy measurement for each Generics.

Elapsed Time *p*-Values	Generic Add Int	Non Generic Cast Int	Non Generic Int	Generic Add Long	Non Generic Cast Long
Generic Add Int	-	<0,05	0,626	0,676	<0,05
Non Generic Cast Int	<0,05	-	<0,05	<0,05	<0,05
Non Generic Int	0,626	<0,05	-	0,791	<0,05
Generic Add Long	0,676	<0,05	0,791	-	<0,05
Non Generic Cast Long	<0,05	<0,05	<0,05	<0,05	-

Table A.202: Table showing the *p*-values for the group Generics with regards to Elapsed Time.

Package Energy *p*-Values	Generic Add Int	Non Generic Cast Int	Non Generic Int	Generic Add Long	Non Generic Cast Long
Generic Add Int	-	<0,05	<0,05	<0,05	<0,05
Non Generic Cast Int	<0,05	-	<0,05	<0,05	<0,05
Non Generic Int	<0,05	<0,05	-	<0,05	<0,05
Generic Add Long	<0,05	<0,05	<0,05	-	<0,05
Non Generic Cast Long	<0,05	<0,05	<0,05	<0,05	-

Table A.203: Table showing the *p*-values for the group Generics with regards to Package Energy.

DRAM Energy *p*-Values	Generic Add Int	Non Generic Cast Int	Non Generic Int	Generic Add Long	Non Generic Cast Long
Generic Add Int	-	<0,05	0,769	0,862	<0,05
Non Generic Cast Int	<0,05	-	<0,05	<0,05	<0,05
Non Generic Int	0,769	<0,05	-	0,719	<0,05
Generic Add Long	0,862	<0,05	0,719	-	<0,05
Non Generic Cast Long	<0,05	<0,05	<0,05	<0,05	-

Table A.204: Table showing the *p*-values for the group Generics with regards to DRAM Energy.

A.3.15 Jumps

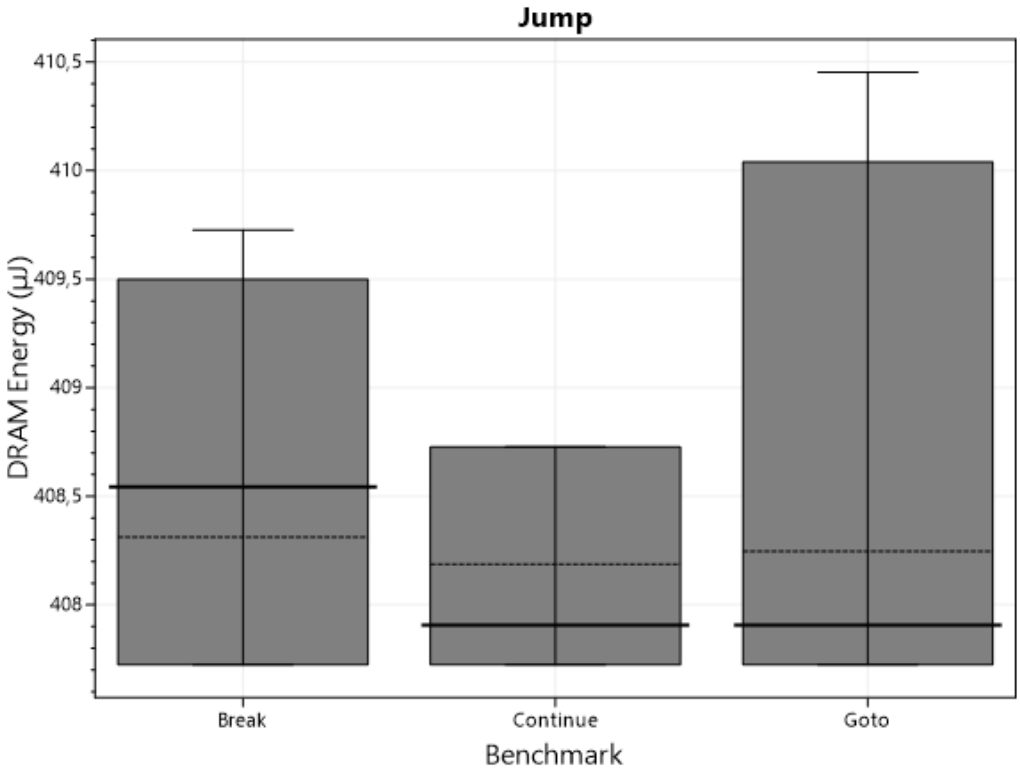

Figure A.159: Boxplot showing how efficient different Jump types are with regards to DRAM Energy.

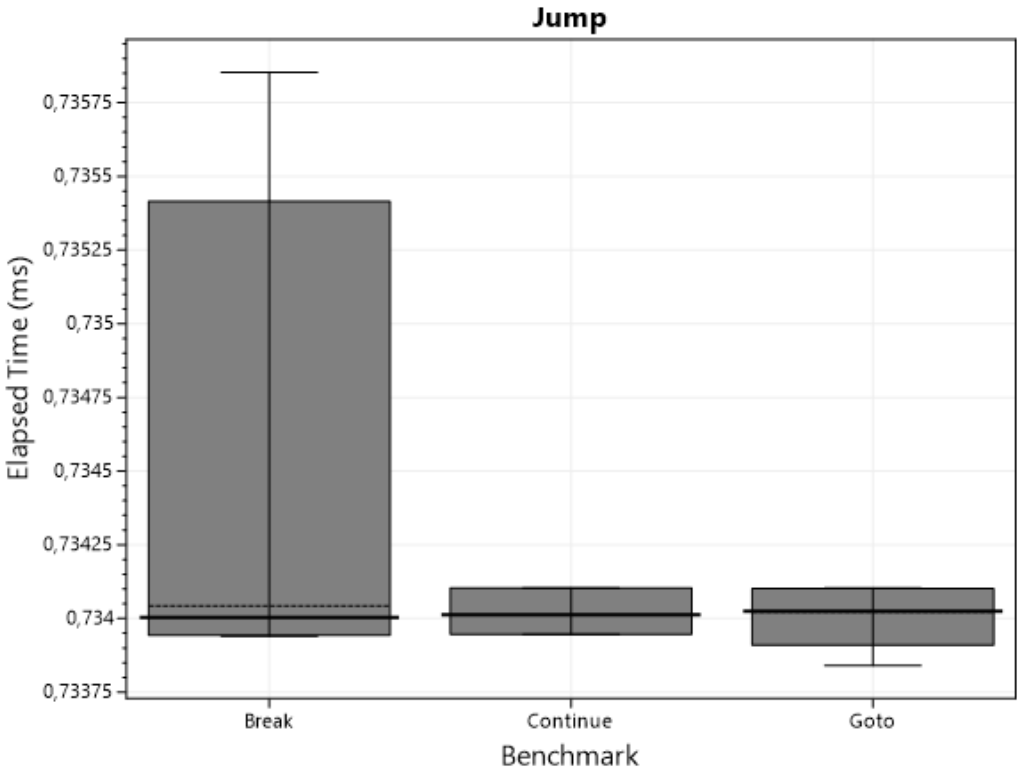

Figure A.160: Boxplot showing how efficient different Jump types are with regards to Elapsed Time.

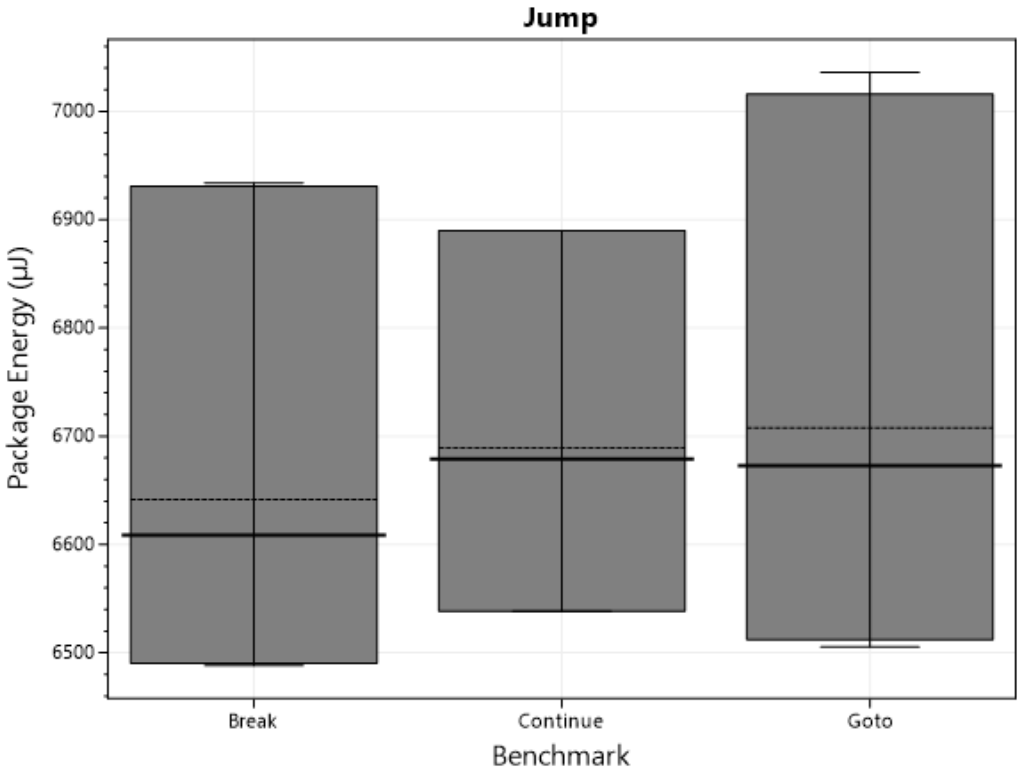

Figure A.161: Boxplot showing how efficient different Jump types are with regards to Package Energy.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Break	0,734043	6.641,496	408,312
Continue	0,734011	6.689,269	408,186
Goto	0,734021	6.707,739	408,247

Table A.205: Table showing the elapsed time and energy measurement for each Jump.

Elapsed Time *p*-Values	Break	Continue	Goto
Break	-	0,518	0,518
Continue	0,518	-	0,280
Goto	0,518	0,280	-

Table A.206: Table showing the *p*-values for the group Jump with regards to Elapsed Time.

Package Energy *p*-Values	Break	Continue	Goto
Break	-	0,064	<0,05
Continue	0,064	-	0,512
Goto	<0,05	0,512	-

Table A.207: Table showing the *p*-values for the group Jump with regards to Package Energy.

DRAM Energy *p*-Values	Break	Continue	Goto
Break	-	0,246	0,515
Continue	0,246	-	0,601
Goto	0,515	0,601	-

Table A.208: Table showing the *p*-values for the group Jump with regards to DRAM Energy.

A.3.16 Return Jump

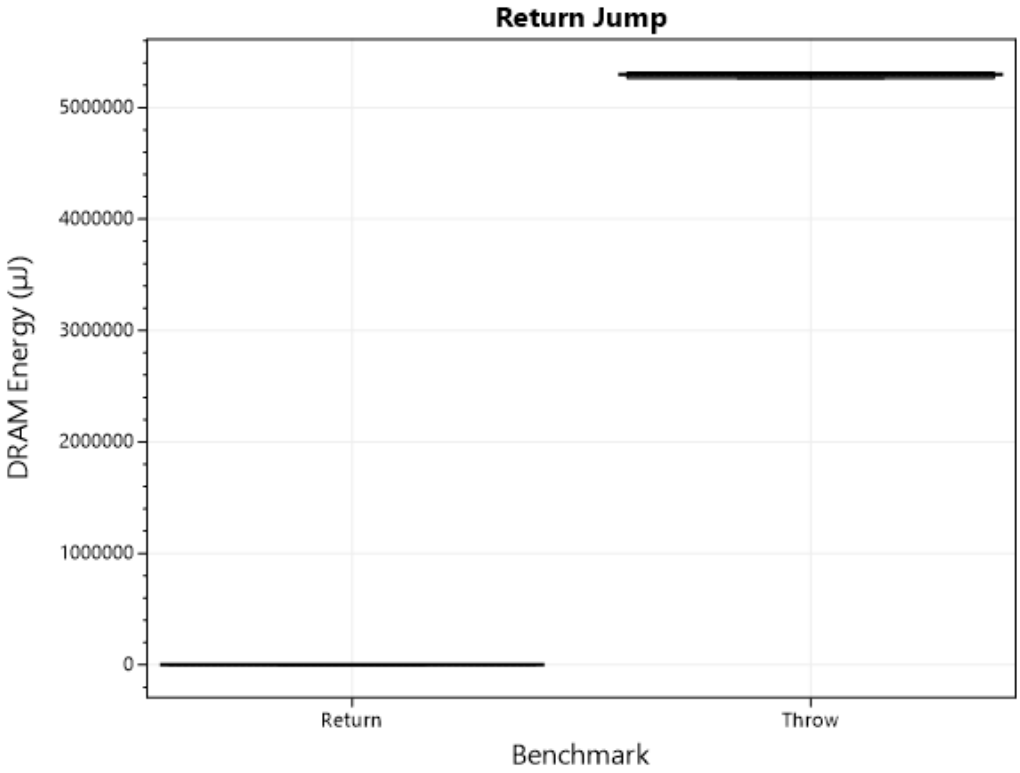

Figure A.162: Boxplot showing how efficient different Return Jump types are with regards to DRAM Energy.

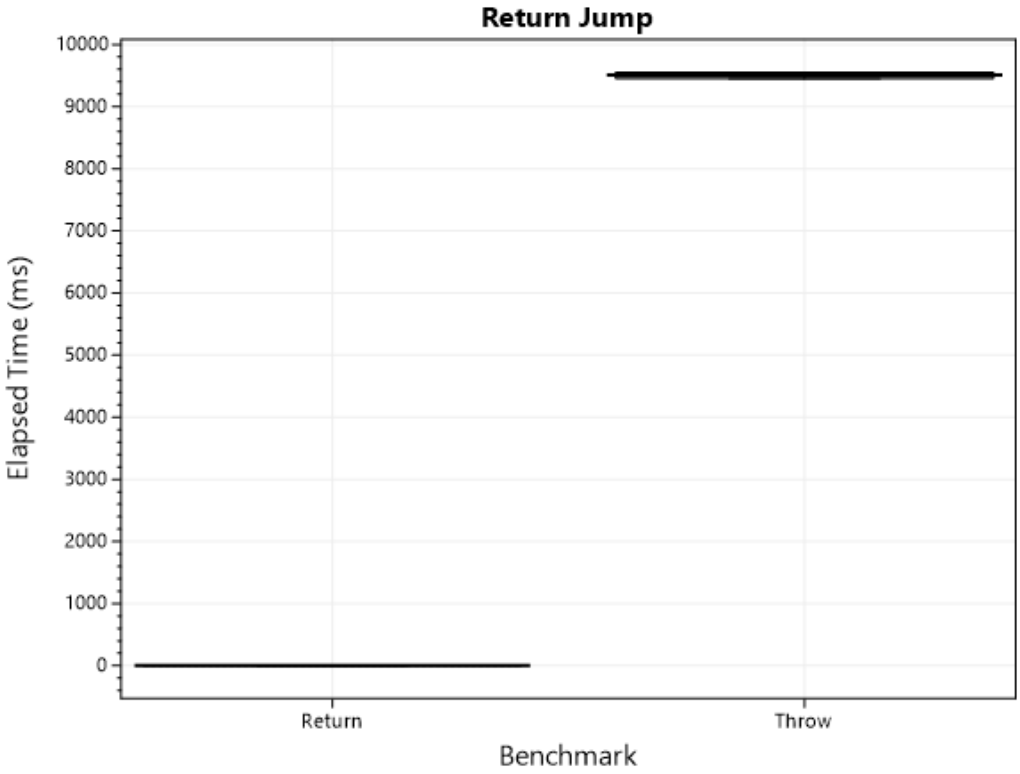

Figure A.163: Boxplot showing how efficient different Return Jump types are with regards to Elapsed Time.

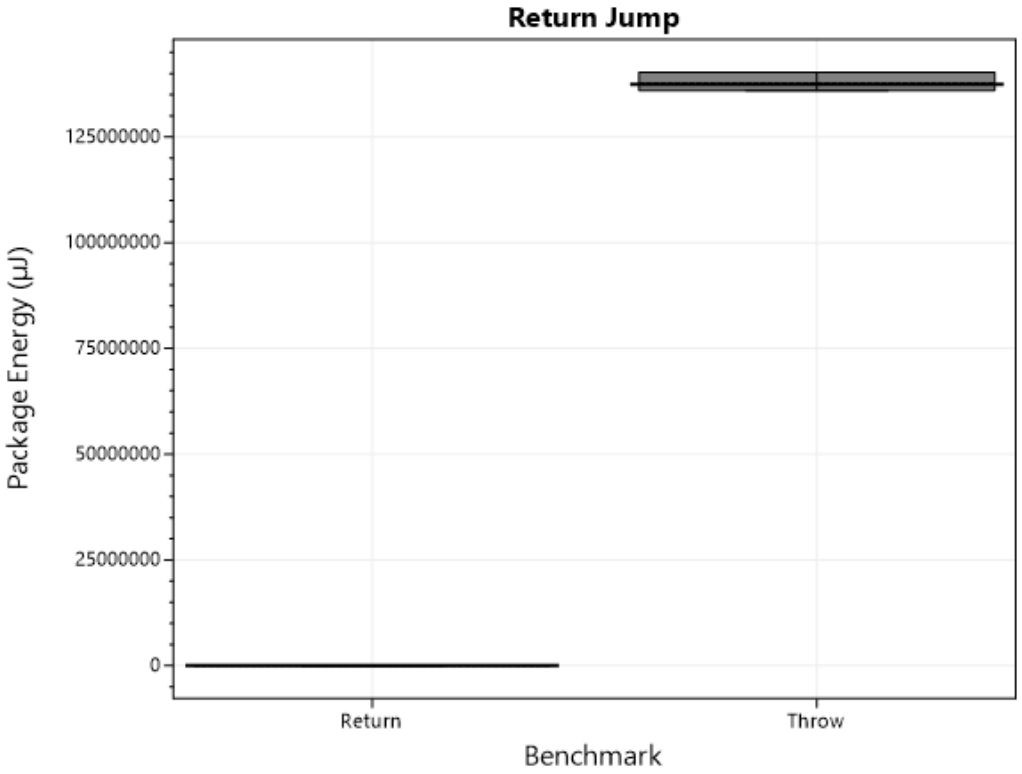

Figure A.164: Boxplot showing how efficient different Return Jump types are with regards to Package Energy.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Return	0,734034	6.547,751	408,232
Throw	9.507,311707	137.726.794,434	5.294.371,033

Table A.209: Table showing the elapsed time and energy measurement for each Return Jump.

Elapsed Time p-Values	Return	Throw
Return	-	<0,05
Throw	<0,05	-

Table A.210: Table showing the p-values for the group Return Jump with regards to Elapsed Time.

Package Energy p-Values	Return	Throw
Return	-	<0,05
Throw	<0,05	-

Table A.211: Table showing the p-values for the group Return Jump with regards to Package Energy.

DRAM Energy p-Values	Return	Throw
Return	-	<0,05
Throw	<0,05	-

Table A.212: Table showing the p-values for the group Return Jump with regards to DRAM Energy.

A.3.17 Exceptions

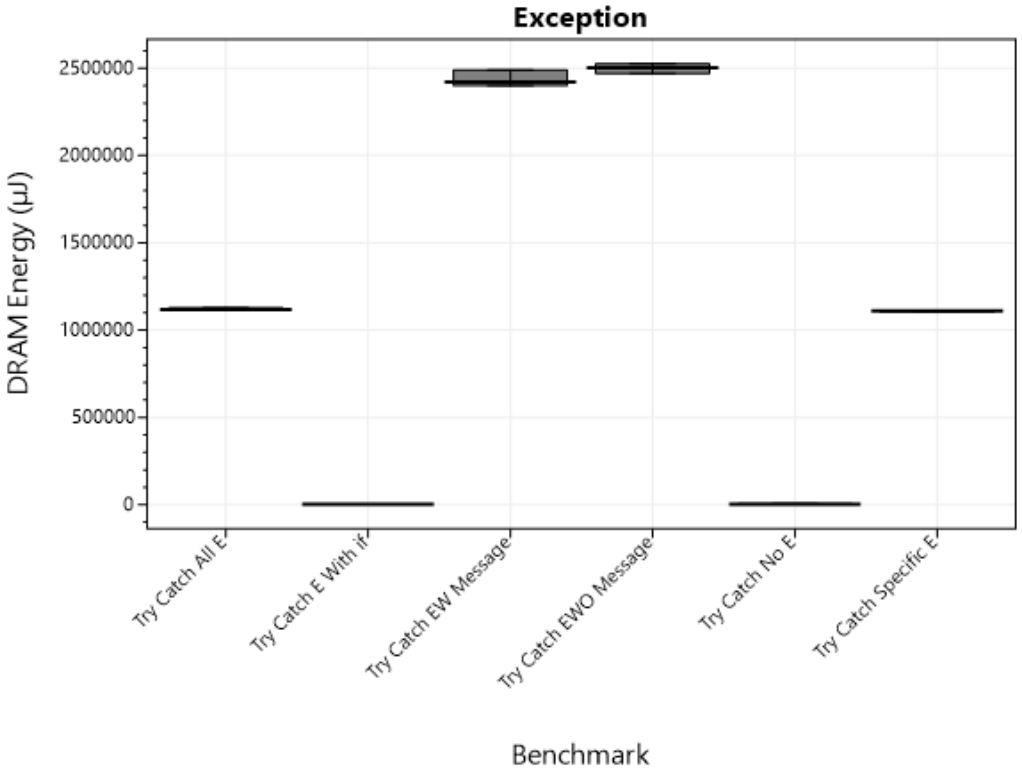

Figure A.165: Boxplot showing how efficient different Exception types are with regards to DRAM Energy.

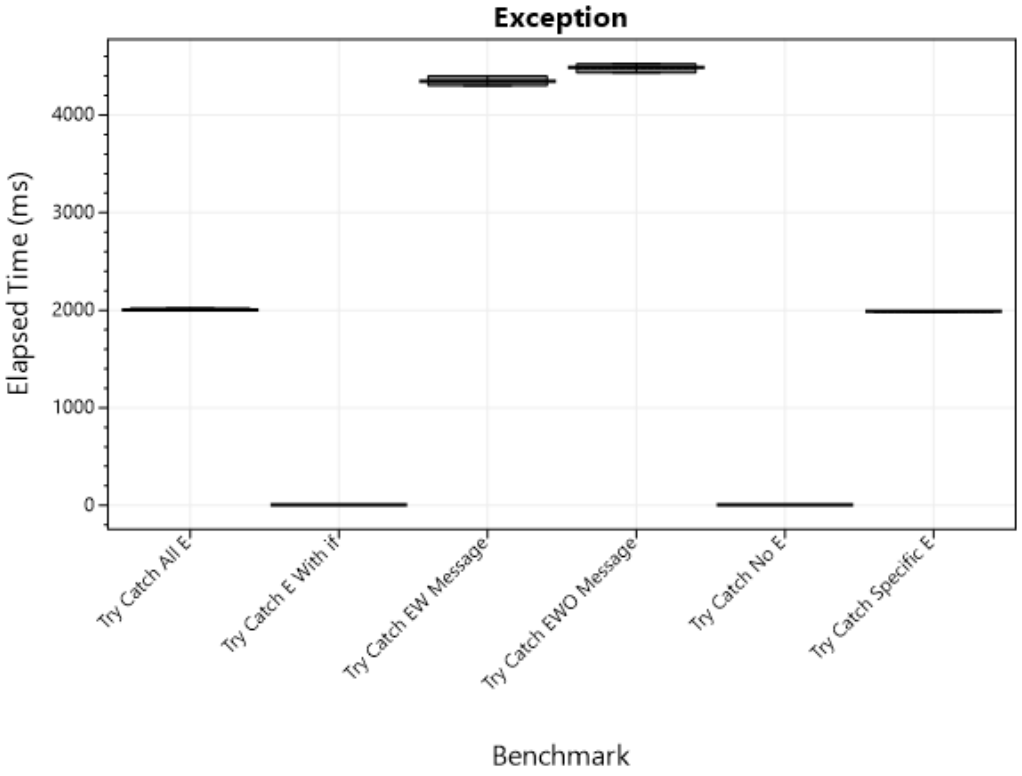

Figure A.166: Boxplot showing how efficient different Exception types are with regards to Elapsed Time.

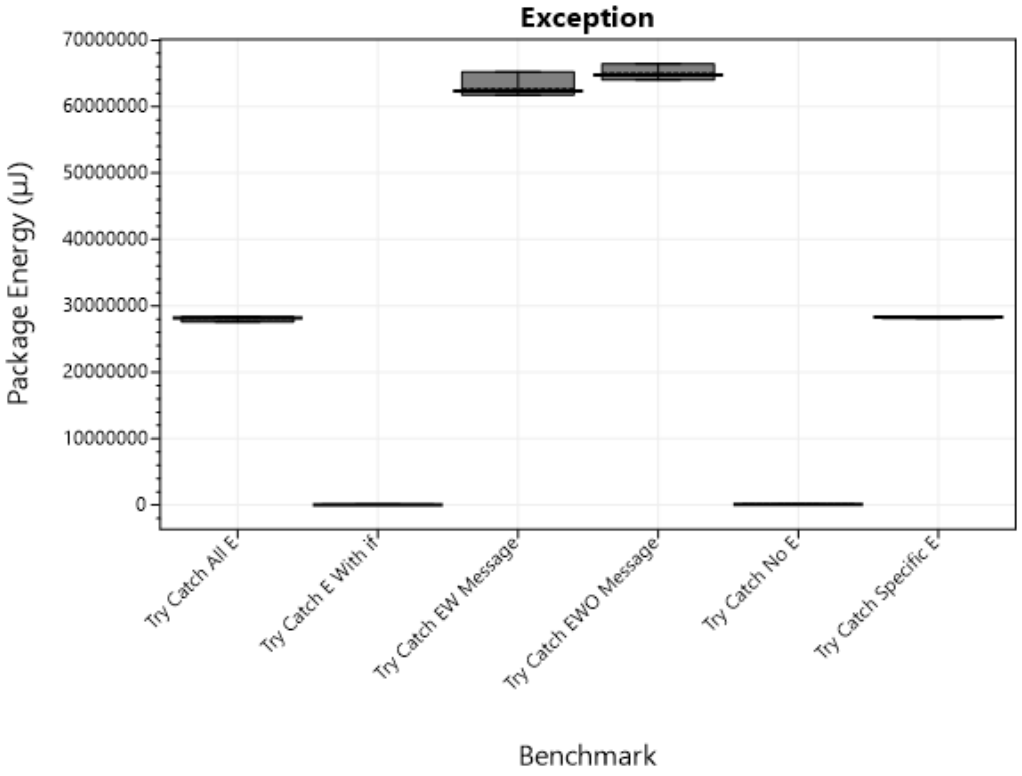

Figure A.167: Boxplot showing how efficient different Exception types are with regards to Package Energy.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Try Catch All E	2.005,725403	28.131.047,363	1.118.590,698
Try Catch E With if	7,161153	64.449,111	3.982,336
Try Catch EW Message	4.347,888499	62.661.289,141	2.424.532,983
Try Catch EWO Message	4.484,282973	65.027.300,347	2.499.001,058
Try Catch No E	7,305439	66.096,315	4.062,883
Try Catch Specific E	1.988,904690	28.280.468,750	1.109.098,985

Table A.213: Table showing the elapsed time and energy measurement for each Exception.

Elapsed Time *p*-Values	Try Catch No E	Try Catch All E	Try Catch Specific E	Try Catch EW Message	Try Catch EWO Message	Try Catch E With if
Try Catch No E	-	<0,05	<0,05	<0,05	<0,05	<0,05
Try Catch All E	<0,05	-	<0,05	<0,05	<0,05	<0,05
Try Catch Specific E	<0,05	<0,05	-	<0,05	<0,05	<0,05
Try Catch EW Message	<0,05	<0,05	<0,05	-	<0,05	<0,05
Try Catch EWO Message	<0,05	<0,05	<0,05	<0,05	-	<0,05
Try Catch E With if	<0,05	<0,05	<0,05	<0,05	<0,05	-

Table A.214: Table showing the *p*-values for the group Exception with regards to Elapsed Time.

Package Energy *p*-Values	Try Catch No E	Try Catch All E	Try Catch Specific E	Try Catch EW Message	Try Catch EWO Message	Try Catch E With if
Try Catch No E	-	<0,05	<0,05	<0,05	<0,05	<0,05
Try Catch All E	<0,05	-	0,114	<0,05	<0,05	<0,05
Try Catch Specific E	<0,05	0,114	-	<0,05	<0,05	<0,05
Try Catch EW Message	<0,05	<0,05	<0,05	-	<0,05	<0,05
Try Catch EWO Message	<0,05	<0,05	<0,05	<0,05	-	<0,05
Try Catch E With if	<0,05	<0,05	<0,05	<0,05	<0,05	-

Table A.215: Table showing the *p*-values for the group Exception with regards to Package Energy.

DRAM Energy *p*-Values	Try Catch No E	Try Catch All E	Try Catch Specific E	Try Catch EW Message	Try Catch EWO Message	Try Catch E With if
Try Catch No E	-	<0,05	<0,05	<0,05	<0,05	<0,05
Try Catch All E	<0,05	-	<0,05	<0,05	<0,05	<0,05
Try Catch Specific E	<0,05	<0,05	-	<0,05	<0,05	<0,05
Try Catch EW Message	<0,05	<0,05	<0,05	-	<0,05	<0,05
Try Catch EWO Message	<0,05	<0,05	<0,05	<0,05	-	<0,05
Try Catch E With if	<0,05	<0,05	<0,05	<0,05	<0,05	-

Table A.216: Table showing the *p*-values for the group Exception with regards to DRAM Energy.

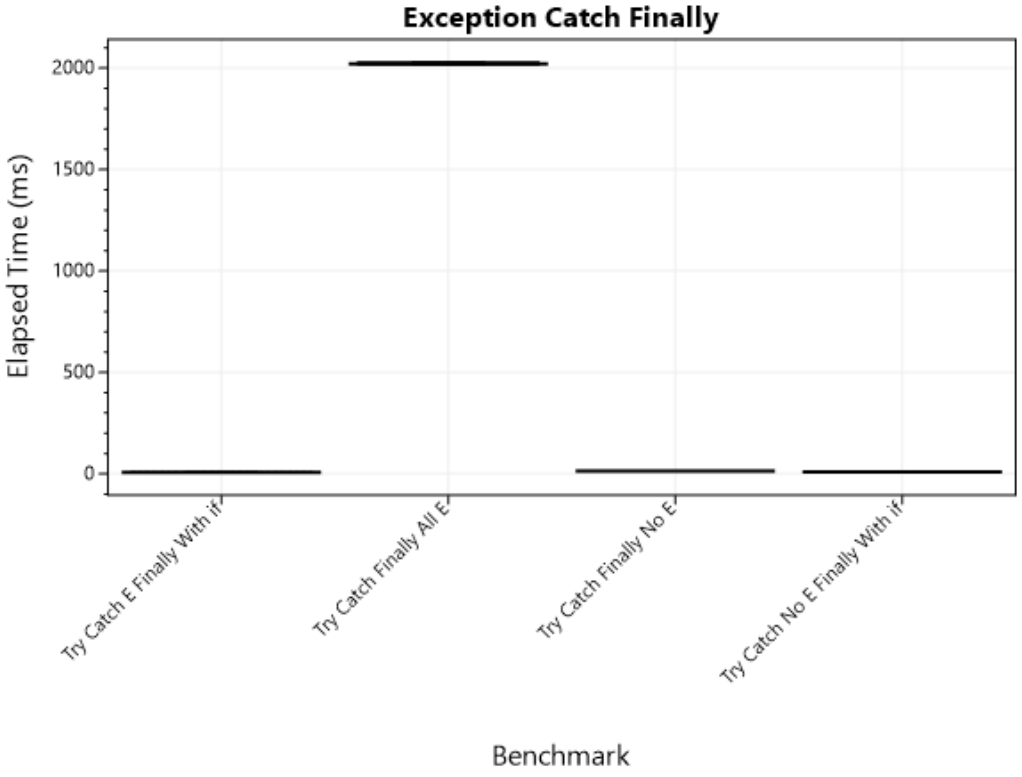

Figure A.168: Boxplot showing how efficient different Exception Catch Finally types are with regards to Elapsed Time.

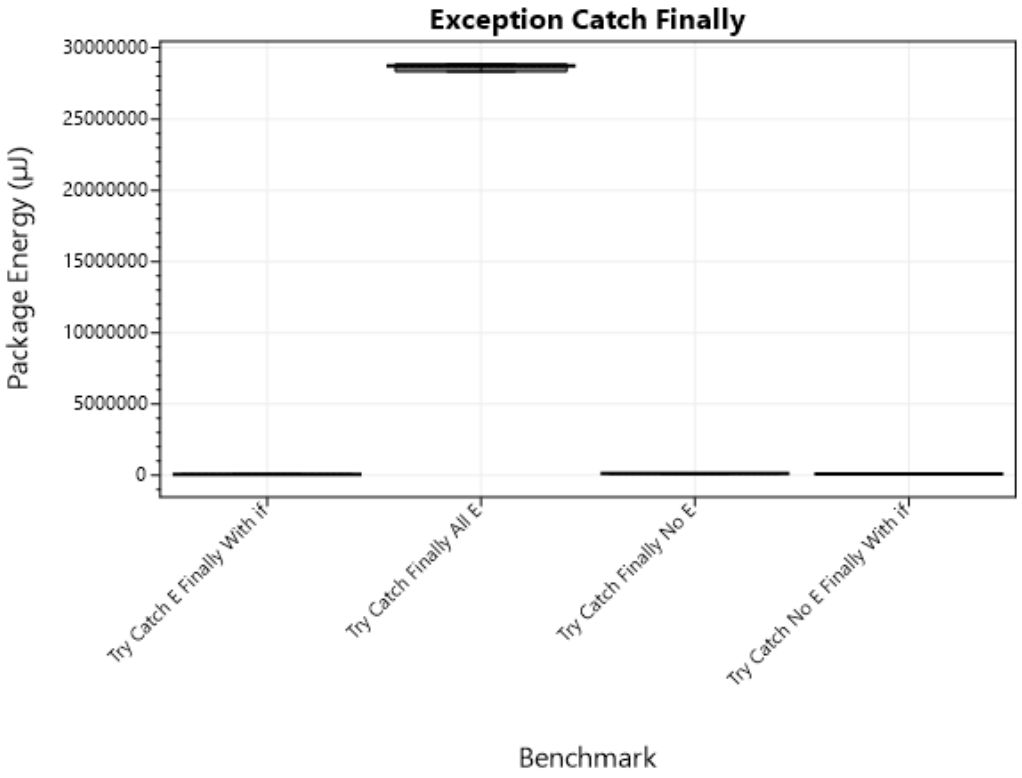

Figure A.169: Boxplot showing how efficient different Exception Catch Finally types are with regards to Package Energy.

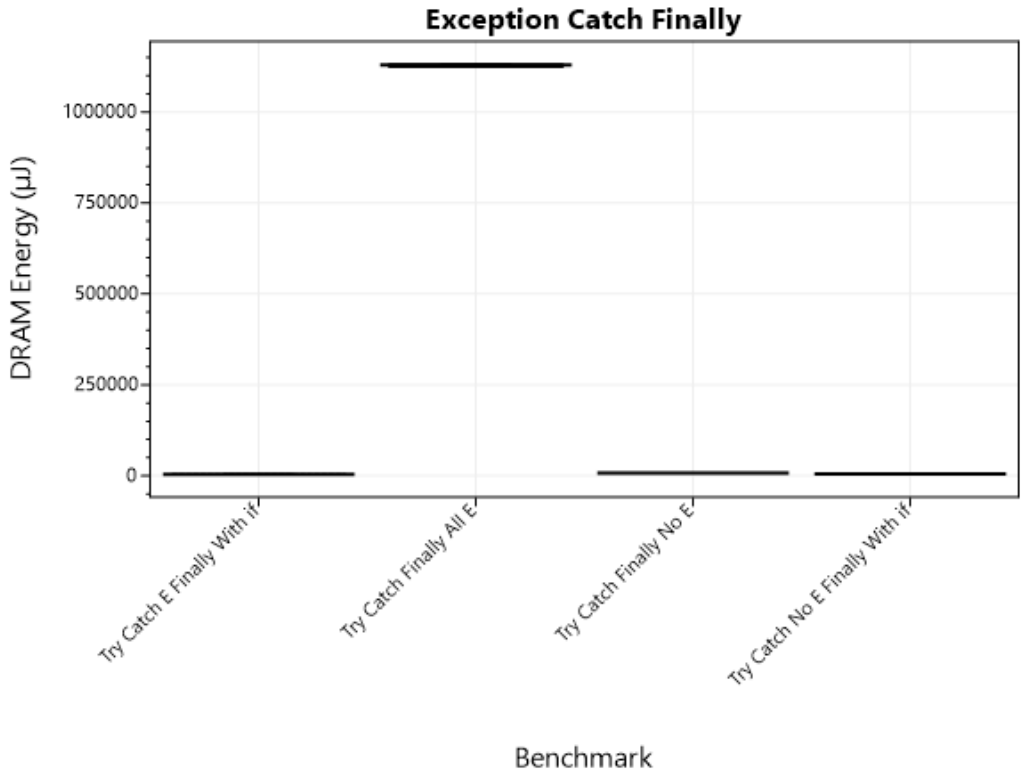

Figure A.170: Boxplot showing how efficient different Exception Catch Finally types are with regards to DRAM Energy.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Try Catch E Finally With if	7,337773	68.690,238	4.081,852
Try Catch Finally All E	2.021,963704	28.706.392,415	1.127.766,927
Try Catch Finally No E	13,656935	129.138,501	7.598,777
Try Catch No E Finally With if	9,392268	84.450,170	5.225,050

Table A.217: Table showing the elapsed time and energy measurement for each Exception Catch Finally.

Elapsed Time *p*-Values	Try Catch E Finally With if	Try Catch Finally All E	Try Catch Finally No E	Try Catch No E Finally With if
Try Catch E Finally With if	-	<0,05	<0,05	<0,05
Try Catch Finally All E	<0,05	-	<0,05	<0,05
Try Catch Finally No E	<0,05	<0,05	-	<0,05
Try Catch No E Finally With if	<0,05	<0,05	<0,05	-

Table A.218: Table showing the *p*-values for the group Exception Catch Finally with regards to Elapsed Time.

Package Energy *p*-Values	Try Catch E Finally With if	Try Catch Finally All E	Try Catch Finally No E	Try Catch No E Finally With if
Try Catch E Finally With if	-	<0,05	<0,05	<0,05
Try Catch Finally All E	<0,05	-	<0,05	<0,05
Try Catch Finally No E	<0,05	<0,05	-	<0,05
Try Catch No E Finally With if	<0,05	<0,05	<0,05	-

Table A.219: Table showing the *p*-values for the group Exception Catch Finally with regards to Package Energy.

DRAM Energy *p*-Values	Try Catch E Finally With if	Try Catch Finally All E	Try Catch Finally No E	Try Catch No E Finally With if
Try Catch E Finally With if	-	<0,05	<0,05	<0,05
Try Catch Finally All E	<0,05	-	<0,05	<0,05
Try Catch Finally No E	<0,05	<0,05	-	<0,05
Try Catch No E Finally With if	<0,05	<0,05	<0,05	-

Table A.220: Table showing the *p*-values for the group Exception Catch Finally with regards to DRAM Energy.

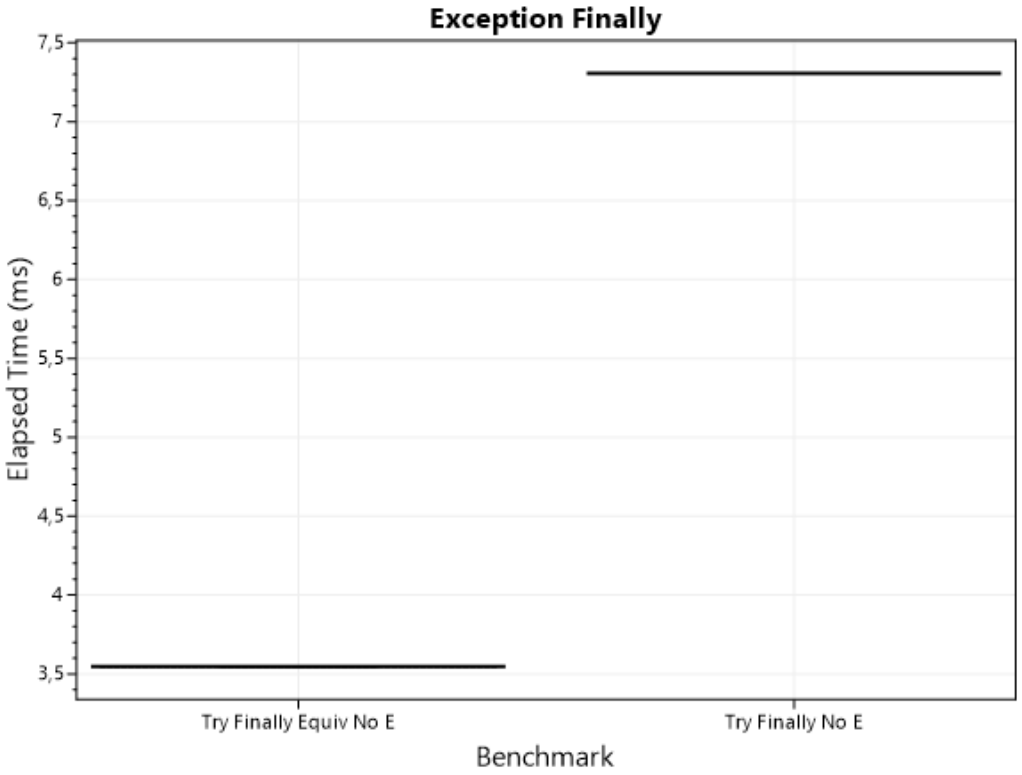

Figure A.171: Boxplot showing how efficient different Exception Finally types are with regards to Elapsed Time.

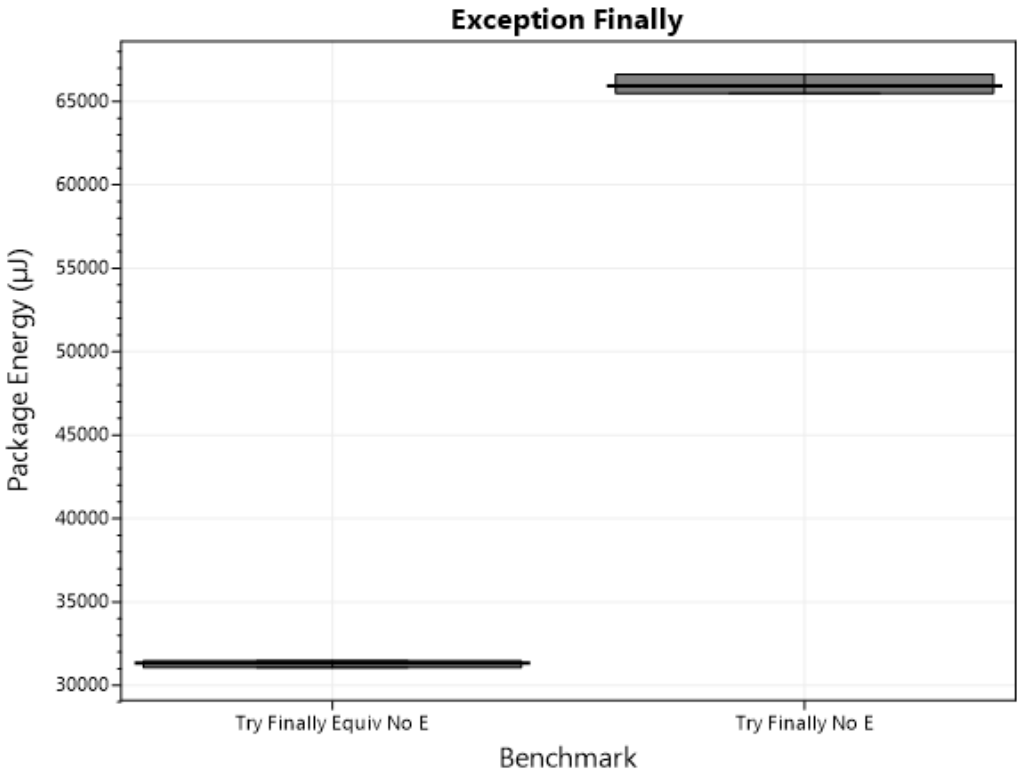

Figure A.172: Boxplot showing how efficient different Exception Finally types are with regards to Package Energy.

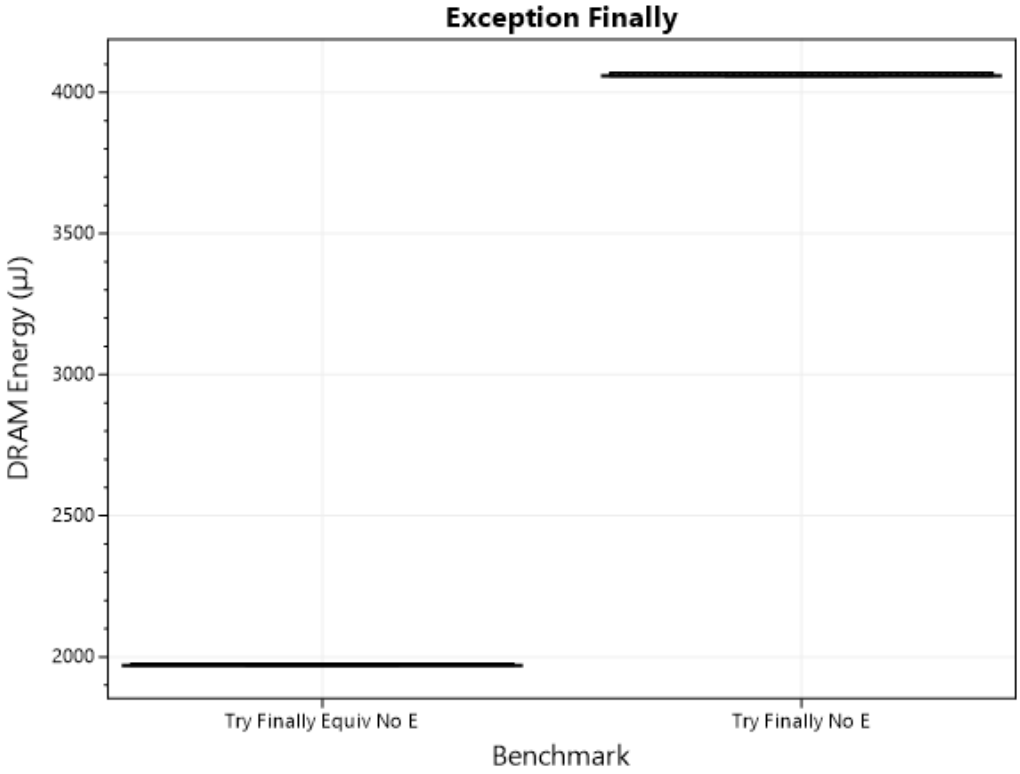

Figure A.173: Boxplot showing how efficient different Exception Finally types are with regards to DRAM Energy.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Try Finally Equiv No E	3,546218	31.294,703	1.970,243
Try Finally No E	7,306547	65.979,833	4.062,883

Table A.221: Table showing the elapsed time and energy measurement for each Exception Finally.

Elapsed Time p-Values	Try Finally No E	Try Finally Equiv No E
Try Finally No E	-	<0,05
Try Finally Equiv No E	<0,05	-

Table A.222: Table showing the p-values for the group Exception Finally with regards to Elapsed Time.

Package Energy *p*-Values	Try Finally No E	Try Finally Equiv No E
Try Finally No E	-	<0,05
Try Finally Equiv No E	<0,05	-

Table A.223: Table showing the *p*-values for the group Exception Finally with regards to Package Energy.

DRAM Energy *p*-Values	Try Finally No E	Try Finally Equiv No E
Try Finally No E	-	<0,05
Try Finally Equiv No E	<0,05	-

Table A.224: Table showing the *p*-values for the group Exception Finally with regards to DRAM Energy.

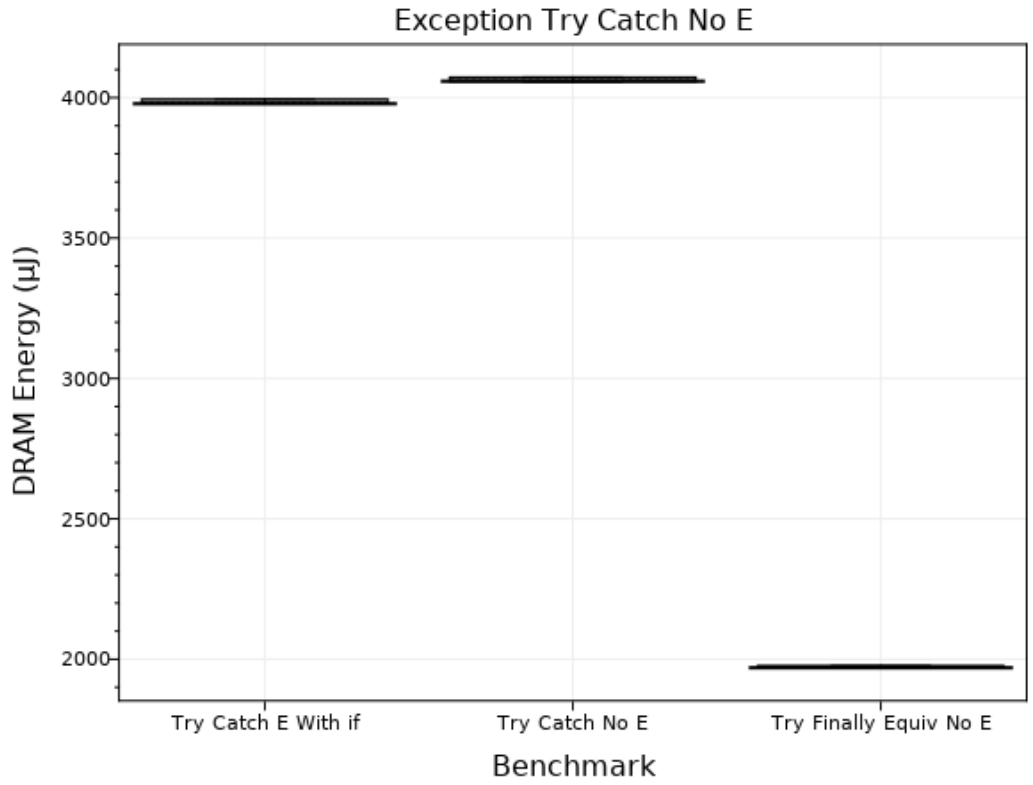

Figure A.174: Boxplot showing how efficient different Exception Try Catch No E types are with regards to DRAM Energy.

Figure A.175: Boxplot showing how efficient different Exception Try Catch No E types are with regards to Elapsed Time.

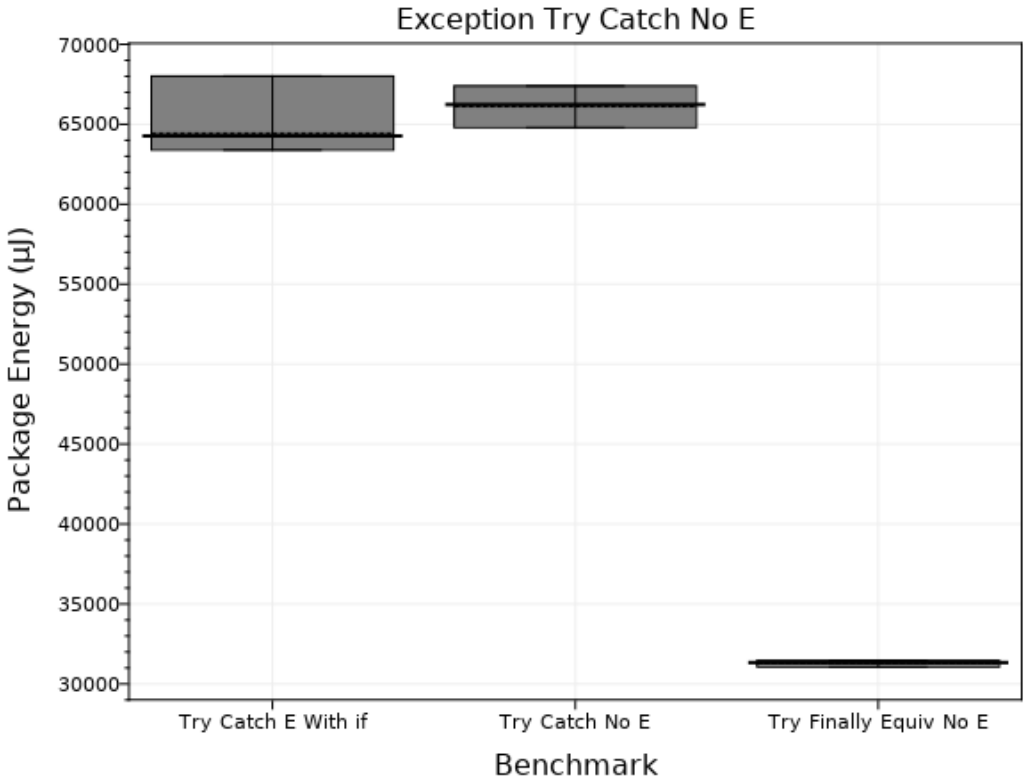

Figure A.176: Boxplot showing how efficient different Exception Try Catch No E types are with regards to Package Energy.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Try Catch E With if	7,161153	64.449,111	3.982,336
Try Catch No E	7,305439	66.096,315	4.062,883
Try Finally Equiv No E	3,546218	31.294,703	1.970,243

Table A.225: Table showing the elapsed time and energy measurement for each Exception Try Catch No E.

Elapsed Time p-Values	Try Catch E With if	Try Catch No E	Try Finally Equiv No E
Try Catch E With if	-	<0,05	<0,05
Try Catch No E	<0,05	-	<0,05
Try Finally Equiv No E	<0,05	<0,05	-

Table A.226: Table showing the p-values for the group Exception Try Catch No E with regards to Elapsed Time.

Package Energy *p*-Values	Try Catch E With if	Try Catch No E	Try Finally Equiv No E
Try Catch E With if	-	<0,05	<0,05
Try Catch No E	<0,05	-	<0,05
Try Finally Equiv No E	<0,05	<0,05	-

Table A.227: Table showing the *p*-values for the group Exception Try Catch No E with regards to Package Energy.

DRAM Energy *p*-Values	Try Catch E With if	Try Catch No E	Try Finally Equiv No E
Try Catch E With if	-	<0,05	<0,05
Try Catch No E	<0,05	-	<0,05
Try Finally Equiv No E	<0,05	<0,05	-

Table A.228: Table showing the *p*-values for the group Exception Try Catch No E with regards to DRAM Energy.

A.4 Full Analysis

A.4.1 Primitive Integer

Unsigned vs Signed Integer Datatypes

The first step analyzing the differences between unsigned and signed integer datatypes is looking at the benchmarks.

```
1  public class PrimitivesBenchmarks {
2      ...
3      [Benchmark("PrimitiveInteger", "Tests operation on primitive
   ↪   int")]
4      public static int Int() {
5          int primitive = 0;
6          for (int i = 0; i < LoopIterations; i++) {
7              primitive++;
8              primitive *= 3;
9              primitive /= 2;
10             primitive--;
11             primitive %= 20;
12         }
```

```
13          return primitive;
14      }
15      ...
16  }
```

Listing 109: The `Int` method which tests operations on `int` type in the `PrimitiveBenchmarks` class.

```
1   public class PrimitivesBenchmarks {
2       ...
3       [Benchmark("PrimitiveInteger", "Tests operation on primitive
    ↪   uint")]
4       public static uint Uint() {
5           uint primitive = 0;
6           for (int i = 0; i < LoopIterations; i++) {
7               primitive++;
8               primitive *= 3;
9               primitive /= 2;
10              primitive--;
11              primitive %= 20;
12          }
13          return primitive;
14      }
15      ...
16  }
```

Listing 110: The `Uint` method which tests operations on `uint` type in the `PrimitiveBenchmarks` class.

We look at the methods for benchmarking `int` and `uint` as these show the only differences between the signed and unsigned integer datatypes. We can see in Listing 109 and Listing 110 that there is only one difference between the benchmarks, specifically in Listing 109 we use `int` while in Listing 110 we use `uint`.

This difference does not tell us why unsigned integer datatypes are more efficient than signed integer datatypes, therefore we move onto the next step of the analysis process.

In the documentation [113] for integral numeric types, there is no information that could give insight into why unsigned integer datatypes are more efficient than signed integer datatypes.

Besides [113], there is not much documentation that is relevant for these results, therefore we move onto the next step of result analysis and look at the Intermediate Language (IL) code.

We look at the IL code for the five operations done inside the loop in the benchmarks for `int` and `uint` to see what the difference between the operations are.

```
IL_0008: ldloc.0
IL_0009: ldc.i4.1
IL_000a: add
IL_000b: stloc.0

IL_000c: ldloc.0
IL_000d: ldc.i4.3
IL_000e: mul
IL_000f: stloc.0

IL_0010: ldloc.0
IL_0011: ldc.i4.2
IL_0012: div
IL_0013: stloc.0

IL_0014: ldloc.0
IL_0015: ldc.i4.1
IL_0016: sub
IL_0017: stloc.0

IL_0018: ldloc.0
IL_0019: ldc.i4.s     20
IL_001b: rem
IL_001c: stloc.0
```

Listing 111: The IL code for the operations in the loop for the `int` benchmark.

In Listing 111 we can see the IL code for the `int` benchmark, it shows that `add` occurs first, then `mul`, then `div`, then `sub` and at the end `rem`.

```
IL_0008: ldloc.0
IL_0009: ldc.i4.1
IL_000a: add
IL_000b: stloc.0

IL_000c: ldloc.0
IL_000d: ldc.i4.3
IL_000e: mul
IL_000f: stloc.0

IL_0010: ldloc.0
IL_0011: ldc.i4.2
IL_0012: div.un
IL_0013: stloc.0

IL_0014: ldloc.0
IL_0015: ldc.i4.1
IL_0016: sub
IL_0017: stloc.0

IL_0018: ldloc.0
IL_0019: ldc.i4.s     20
IL_001b: rem.un
IL_001c: stloc.0
```

Listing 112: The IL code for the operations in the loop for the `uint` benchmark.

In Listing 112 we can see the IL code for the `uint` benchmark. It shows that there are only two differences between the `int` and `uint` benchmark, these being that `div.un` and `rem.un` is used instead of `div` and `rem` respectively.

In the Common Language Infrastructure standard [109, p.356-357, p.381-382] there is one interesting part for the instructions `div`, `div.un`, `rem` and `rem.un`. Namely that the instructions `div` and `rem` work on both integer and floating point datatypes, while `div.un` and `rem.un` only are present for unsigned integer datatypes. Because of this, we test if `uint` and `int` have a significant difference when removing these instructions, i.e. we remove division and modulo from our benchmarks, this results in the code seen in

Listing 113 and Listing 114.

```
public class PrimitivesBenchmarks {
    ...
    [Benchmark("PrimitiveInteger", "Tests operation on primitive
    ↪ int")]
    public static int Int() {
        int primitive = 0;
        for (int i = 0; i < LoopIterations; i++) {
            primitive++;
            primitive *= 3;
            primitive--;
        }
        return primitive;
    }
    ...
}
```

Listing 113: The `Int` method which tests operations on `int` type in the `PrimitiveBenchmarks` class where division and modulo are removed.

```
public class PrimitivesBenchmarks {
    ...
    [Benchmark("PrimitiveInteger", "Tests operation on primitive
    ↪ uint")]
    public static uint Uint() {
        uint primitive = 0;
        for (int i = 0; i < LoopIterations; i++) {
            primitive++;
            primitive *= 3;
            primitive--;
        }
        return primitive;
    }
    ...
}
```

Listing 114: The `Uint` method which tests operations on `uint` type in the `PrimitiveBenchmarks` class where division and modulo are removed.

Benchmark	Time (ms)	Package Energy (μJ)	DRAM Energy (μJ)
Int	0,734049	7311,213	408,496
Uint	0,734045	7318,311	408,665

Table A.229: Table showing the elapsed time and energy measurement for `int` and `uint`.

In Table A.229 we can see the results of the new benchmarks. The results are significantly closer to each other, which we also see when looking at the p-values. The p-values for Time, Package Energy, and DRAM Energy are 0,721, 0,793, and 0,533 respectively, all being above 0,05 meaning they are not significantly different. We can conclude that division and modulo are faster on unsigned integer datatypes compared to signed integer datatypes.

Short Integer Datatypes

There is one case where an unsigned integer datatype is less than 1% more efficient than a signed integer datatype, the `ushort` compared with the `short` datatype.

To figure out what the differences are, we compare the `short` and `ushort` benchmarks.

```
1  public class PrimitivesBenchmarks {
2      ...
3      [Benchmark("PrimitiveInteger", "Tests operation on primitive
   ↪   short")]
4      public static short Short() {
5          short primitive = 0;
6          for (int i = 0; i < LoopIterations; i++) {
7              primitive++;
8              primitive *= 3;
9              primitive /= 2;
10             primitive--;
11             primitive %= 20;
12         }
```

```
13          return primitive;
14      }
15      ...
16  }
```

Listing 115: The `Short` method which tests operations on `short` type in the `PrimitiveBenchmarks` class.

```
1   public class PrimitivesBenchmarks {
2       ...
3       [Benchmark("PrimitiveInteger", "Tests operation on primitive
    ↪   ushort")]
4       public static ushort Ushort() {
5           ushort primitive = 0;
6           for (int i = 0; i < LoopIterations; i++) {
7               primitive++;
8               primitive *= 3;
9               primitive /= 2;
10              primitive--;
11              primitive %= 20;
12          }
13          return primitive;
14      }
15      ...
16  }
```

Listing 116: The `Ushort` method which tests operations on `ushort` type in the `PrimitiveBenchmarks` class.

The benchmarks for `short` and `ushort` can be seen in Listing 115 and Listing 116. Here we can see that the only difference between the benchmarks is the datatype. This difference does not explain why there is no significant difference between the `ushort` and `short` datatypes with regards to Package Energy consumption, therefore we move to the next step of result analysis.

There is no documentation for the `short` and `ushort` datatypes specifically, so we look in the Common Language Infrastructure standard [109,

p.317] for the `int16` and `uint16` datatypes. Here we find the reason why `short`, `ushort`, `byte` and `sbyte` are less efficient than the other integer datatypes, as it says "Loading from these locations onto the stack converts them to 4-byte ...", meaning these datatypes are converted to 4-byte (32-bit) integers, which means additional operations occur in the background. This explain the difference in behaviour between the `short` datatypes and the rest of the integer datatypes, as the 4-byte integer it is turned into is signed and therefore the `div.un` and `rem.un` instructions may not be used. To confirm this explanation we look at the IL code for the benchmarks.

```
IL_0008: ldloc.0
IL_0009: ldc.i4.1
IL_000a: add
IL_000b: conv.i2
IL_000c: stloc.0

IL_000d: ldloc.0
IL_000e: ldc.i4.3
IL_000f: mul
IL_0010: conv.i2
IL_0011: stloc.0

IL_0012: ldloc.0
IL_0013: ldc.i4.2
IL_0014: div
IL_0015: conv.i2
IL_0016: stloc.0

IL_0017: ldloc.0
IL_0018: ldc.i4.1
IL_0019: sub
IL_001a: conv.i2
IL_001b: stloc.0

IL_001c: ldloc.0
IL_001d: ldc.i4.s     20
IL_001f: rem
IL_0020: conv.i2
IL_0021: stloc.0
```

Listing 117: The IL code for the operations in the loop for the `short` benchmark.

In Listing 117 we can see the IL code for the `short` benchmark. It shows that `add` occurs first, then `mul`, then `div`, then `sub` and at the end `rem`. Important to note is that the `conv.i2` instruction occurs, which converts the value on top of the stack to a signed 2-byte (16-bit) integer.

```
1  IL_0008: ldloc.0
2  IL_0009: ldc.i4.1
3  IL_000a: add
4  IL_000b: conv.u2
5  IL_000c: stloc.0
6
7  IL_000d: ldloc.0
8  IL_000e: ldc.i4.3
9  IL_000f: mul
10 IL_0010: conv.u2
11 IL_0011: stloc.0
12
13 IL_0012: ldloc.0
14 IL_0013: ldc.i4.2
15 IL_0014: div
16 IL_0015: conv.u2
17 IL_0016: stloc.0
18
19 IL_0017: ldloc.0
20 IL_0018: ldc.i4.1
21 IL_0019: sub
22 IL_001a: conv.u2
23 IL_001b: stloc.0
24
25 IL_001c: ldloc.0
26 IL_001d: ldc.i4.s     20
27 IL_001f: rem
28 IL_0020: conv.u2
29 IL_0021: stloc.0
```

Listing 118: The IL code for the operations in the loop for the `ushort` benchmark.

In Listing 118 we can see the IL code for the `ushort` benchmark. Here we can see that the `conv.u2` instruction occurs, that converts the value on top of the stack to an unsigned 2-byte (16-bit) integer instead of a signed 2-byte (16-bit) integer as the `short` benchmark does. Furthermore, to note, the `div` and `rem` instructions do not use the unsigned counterpart, because the unsigned short is turned into a signed integer when doing the calculation. This explains why the difference between `short` and `ushort` is so small with regards to Package Energy consumption. The fact that there still is a significant difference with regards to their p-value, has to do with variances in the results, which is explored in Chapter 8.

However, the same instructions are created for the `byte` and `sbyte` integer datatypes, despite the `byte` datatypes following the same behaviour as the other integer datatypes. In [109] we can see in Table I.1 on page 20 that `sbyte` is not a Common Language Specification (CLS) type, meaning the `sbyte` datatype does not have as many requirements as the `byte` datatype, which could have the effect that `sbyte` is less efficient. The requirements of a CLS type can be seen in [109, p.69-71]. This could be an explanation by itself, but we look at the assembly code for further exploration.

```
1  // primitive++;
2  mov         eax,dword ptr [rbp+2Ch]
3  inc         eax
4  movzx       eax,al
5  mov         dword ptr [rbp+2Ch],eax
6  // primitive *= 3;
7  mov         eax,dword ptr [rbp+2Ch]
8  lea         eax,[rax+rax*2]
9  movzx       eax,al
10 mov         dword ptr [rbp+2Ch],eax
11 // primitive /= 2;
12 mov         eax,dword ptr [rbp+2Ch]
13 shr         eax,1Fh
14 add         eax,dword ptr [rbp+2Ch]
15 sar         eax,1
16 movzx       eax,al
17 mov         dword ptr [rbp+2Ch],eax
```

```
// primitive--;
mov         eax,dword ptr [rbp+2Ch]
dec         eax
movzx       eax,al
mov         dword ptr [rbp+2Ch],eax
// primitive %= 20;
mov         eax,dword ptr [rbp+2Ch]
mov         ecx,14h
cdq
idiv        eax,ecx
lea         eax,[rax+rax*4]
shl         eax,2
mov         edx,dword ptr [rbp+2Ch]
sub         edx,eax
movzx       eax,dl
mov         dword ptr [rbp+2Ch],eax
```

Listing 119: The assembly code for the operations in the loop for the `byte` benchmark.

In Listing 119 we can see the five operations done in the `byte` benchmark in assembly code.

```
// primitive++;
mov         eax,dword ptr [rbp+2Ch]
inc         eax
movsx       rax,al
mov         dword ptr [rbp+2Ch],eax
// primitive *= 3;
mov         eax,dword ptr [rbp+2Ch]
lea         eax,[rax+rax*2]
movsx       rax,al
mov         dword ptr [rbp+2Ch],eax
// primitive /= 2;
mov         eax,dword ptr [rbp+2Ch]
shr         eax,1Fh
add         eax,dword ptr [rbp+2Ch]
sar         eax,1
movsx       rax,al
```

```
17 mov         dword ptr [rbp+2Ch],eax
18 // primitive--;
19 mov         eax,dword ptr [rbp+2Ch]
20 dec         eax
21 movsx       rax,al
22 mov         dword ptr [rbp+2Ch],eax
23 // primitive %= 20;
24 mov         eax,dword ptr [rbp+2Ch]
25 mov         ecx,14h
26 cdq
27 idiv        eax,ecx
28 lea         eax,[rax+rax*4]
29 shl         eax,2
30 mov         edx,dword ptr [rbp+2Ch]
31 sub         edx,eax
32 movsx       rax,dl
33 mov         dword ptr [rbp+2Ch],eax
```

Listing 120: The assembly code for the operations in the loop for the `sbyte` benchmark.

In Listing 120 we can see the five operations done in the `sbyte` benchmark in the assembly code. The only difference between the two benchmarks is that `sbyte` uses the `movsx` instruction instead of the `movzx` instruction. The `movsx` instruction extends the given variable to a 32-bit integer with the same signed bit as the original variable (meaning if it is negative, the converted variable is also negative) while the `movzx` instruction extends the given variable to a 32-bit integer with zero-extension, meaning all the unused bits are 0's meaning it will always be a positive integer after converting. This does not give an explanation besides the `movsx` instruction being less efficient than the `movzx` instruction, therefore we leave further exploration of this result to future work.

Unsigned Integer

The result that `uint` is more efficient compared to all other integer datatypes is interesting, as the computer runs 64-bit, and we therefore expect `ulong` to be more efficient.

The first step for finding the differences is looking at the `uint` and `ulong` benchmarks.

```
public class PrimitivesBenchmarks {
    ...
    [Benchmark("PrimitiveInteger", "Tests operation on primitive
    ↪ uint")]
    public static uint Uint() {
        uint primitive = 0;
        for (int i = 0; i < LoopIterations; i++) {
            primitive++;
            primitive *= 3;
            primitive /= 2;
            primitive--;
            primitive %= 20;
        }
        return primitive;
    }
    ...
}
```

Listing 121: The `Uint` method which tests operations on `uint` type in the `PrimitiveBenchmarks` class where division and modulo are removed.

```
public class PrimitivesBenchmarks {
    ...
    [Benchmark("PrimitiveInteger", "Tests operation on primitive
    ↪ ulong")]
    public static ulong Ulong() {
        ulong primitive = 0;
        for (int i = 0; i < LoopIterations; i++) {
            primitive++;
            primitive *= 3;
            primitive /= 2;
            primitive--;
            primitive %= 20;
        }
        return primitive;
    }
```

```
15      ...
16  }
```

Listing 122: The `Ulong` method which tests operations on `ulong` type in the `PrimitiveBenchmarks` class where division and modulo are removed.

The benchmarks for `uint` and `ulong` can be seen in Listing 121 and Listing 122. Here we can see that the only difference between the benchmarks is the datatype. This difference does not explain why there is no significant difference between the `uint` and `ulong` datatypes with regards to Package Energy consumption, as such we move to the next step of result analysis.

In [113] there is no information that could give insight into why the `uint` datatype is the most efficient datatype. Besides [113], there is not much information, because of this we move to the next result analysis step and look at the IL code.

We look at the IL code for the five operations done inside the loop in the benchmarks for `uint` and `ulong` to see what the difference between the operations are.

```
1   IL_0008: ldloc.0
2   IL_0009: ldc.i4.1
3   IL_000a: add
4   IL_000b: stloc.0
5
6   IL_000c: ldloc.0
7   IL_000d: ldc.i4.3
8   IL_000e: mul
9   IL_000f: stloc.0
10
11  IL_0010: ldloc.0
12  IL_0011: ldc.i4.2
13  IL_0012: div.un
14  IL_0013: stloc.0
15
16  IL_0014: ldloc.0
17  IL_0015: ldc.i4.1
18  IL_0016: sub
19  IL_0017: stloc.0
```

```
20
21 IL_0018: ldloc.0
22 IL_0019: ldc.i4.s     20
23 IL_001b: rem.un
24 IL_001c: stloc.0
```

Listing 123: The IL code for the operations in the loop for the `uint` benchmark.

In Listing 123 we see the IL code for the `uint` benchmark. It shows that `add` occurs first, then `mul`, then `div`, then `sub` and at the end `rem`.

```
1  IL_0009: ldloc.0
2  IL_000a: ldc.i4.1
3  IL_000b: conv.i8
4  IL_000c: add
5  IL_000d: stloc.0
6
7  IL_000e: ldloc.0
8  IL_000f: ldc.i4.3
9  IL_0010: conv.i8
10 IL_0011: mul
11 IL_0012: stloc.0
12
13 IL_0013: ldloc.0
14 IL_0014: ldc.i4.2
15 IL_0015: conv.i8
16 IL_0016: div.un
17 IL_0017: stloc.0
18
19 IL_0018: ldloc.0
20 IL_0019: ldc.i4.1
21 IL_001a: conv.i8
22 IL_001b: sub
23 IL_001c: stloc.0
24
25 IL_001d: ldloc.0
26 IL_001e: ldc.i4.s     20
27 IL_0020: conv.i8
```

```
28 IL_0021: rem.un
29 IL_0022: stloc.0
```

Listing 124: The IL code for the operations in the loop for the `ulong` benchmark.

In Listing 124 we can see the IL code for the `ulong` benchmark. Here we can see that the difference between `uint` and `ulong` is that the instruction `conv.i8` occurs for all the operations for `ulong`. This difference is expected and is the same for the signed variant, where there is no significant difference between `int` and `long`, therefore this is not an explanation for the large difference between `uint` and `ulong`. Because of this, we move to the next step of result analysis and look at the assembly code.

```
1  // primitive++;
2  mov         eax,dword ptr [rbp+2Ch]
3  inc         eax
4  mov         dword ptr [rbp+2Ch],eax
5  // primitive *= 3;
6  mov         eax,dword ptr [rbp+2Ch]
7  lea         eax,[rax+rax*2]
8  mov         dword ptr [rbp+2Ch],eax
9  // primitive /= 2;
10 mov         eax,dword ptr [rbp+2Ch]
11 shr         eax,1
12 mov         dword ptr [rbp+2Ch],eax
13 // primitive--;
14 mov         eax,dword ptr [rbp+2Ch]
15 dec         eax
16 mov         dword ptr [rbp+2Ch],eax
17 // primitive %= 20;
18 mov         eax,dword ptr [rbp+2Ch]
19 mov         ecx,14h
20 xor         edx,edx
21 div         eax,ecx
22 mov         dword ptr [rbp+2Ch],edx
```

Listing 125: The assembly code for the operations in the loop for the `uint` benchmark.

In Listing 125 we can see the five operations done in the `uint` benchmark in assembly code.

```
// primitive++;
mov         eax,1
movsxd      rax,eax
add         rax,qword ptr [rbp+38h]
mov         qword ptr [rbp+38h],rax
// primitive *= 3;
mov         eax,3
movsxd      rax,eax
imul        rax,qword ptr [rbp+38h]
mov         qword ptr [rbp+38h],rax
// primitive /= 2;
mov         rax,qword ptr [rbp+38h]
mov         edx,2
movsxd      rcx,edx
xor         edx,edx
div         rax,rcx
mov         qword ptr [rbp+38h],rax
// primitive--;
mov         rax,qword ptr [rbp+38h]
mov         edx,1
movsxd      rdx,edx
sub         rax,rdx
mov         qword ptr [rbp+38h],rax
// primitive %= 20;
mov         rax,qword ptr [rbp+38h]
mov         edx,14h
movsxd      rcx,edx
xor         edx,edx
div         rax,rcx
mov         qword ptr [rbp+38h],rdx
```

Listing 126: The assembly code for the operations in the loop for the `ulong` benchmark.

In Listing 126 we can see the five operations done in the `ulong` benchmark in the assembly code. Here we can see that there are quite a few differences:

1. The addition and subtraction uses the `inc` and `dec` instructions for `uint`, while for `ulong`, 1 is added and subtracted,
2. The multiplication uses the `lea` instruction for `uint`, while using `imul` for `ulong`,
3. The division uses the `shr` instruction for `uint`, while using `xor` and `div` for `ulong`, and
4. The `movsxd` instruction is used for all the `ulong` operations.

These differences show that operations with `uint` are optimized by the compiler in a lot of cases. We conclude that this is why `uint` is more efficient than other unsigned integer datatypes.

A.4.2 Selection

Switch Statements vs If Statements

The first step to explain the differences between `switch` statements and `if` statements is checking the benchmarks. We only check traditional `switch` statements compared to an equivalent `if` statement, as the conclusion is the same for all types of `switch` statements.

```
1  public class SelectionBenchmarks {
2      ...
3      [Benchmark("SelectionSwitch", "Tests switch statement")]
4      public static int Switch() {
5          int count = 1;
6          for (int i = 0; i < LoopIterations; i++) {
7              switch (count) {
8                  case 1:
9                      count = 2;
10                     break;
11                 case 2:
12                     count = 3;
13                     break;
14                 ...
15                 case 9:
16                     count = 10;
```

```
                    break;
                default:
                    count = 1;
                    break;
            }
        }
        return count;
    }
    ...
}
```

Listing 127: The `Switch` method which tests `switch` statements in the `SelectionBenchmarks` class.

```
public class SelectionBenchmarks {
    ...
    [Benchmark("SelectionSwitch", "Tests if statement compared
    ↪ to switch")]
    public static int IfComparableWithSwitch() {
        int count = 1;
        for (int i = 0; i < LoopIterations; i++) {
            if (count == 1) {
                count = 2;
                continue;
            }
            if (count == 2) {
                count = 3;
                continue;
            }
            ...
            if (count == 9) {
                count = 10;
                continue;
            }
            count = 1;
        }
        return count;
```

```
23      }
24      ...
25  }
```

Listing 128: The `IfComparableWithSwitch` method which tests `if` statements that are similar to `switch` statements compared in the `SelectionBenchmarks` class.

In Listing 127 and Listing 128 we can see that there are differences in the benchmarks. The main difference is that in Listing 127 there is a `switch` statement with `cases`, while in Listing 128 all `if` statements before the one that is `true` is evaluated.

In [109, p.96] we can see that "The target of br, br.s, and the conditional branches, is the address specified. The targets of switch are all of the addresses specified in the jump table." and in [109, p.392] we can see that "The switch instruction implements a jump table. ... The switch instruction pops value off the stack and compares it, as an unsigned integer, to n. If value is less than n, execution is transferred to the value'th target, where targets are numbered from 0 (i.e., a value of 0 takes the first target, a value of 1 takes the second target, and so on). If value is not less than n, execution continues at the next instruction (fall through).". This implies that when using a `switch` statement, unconditional jumps are utilized, which would be different than when using an `if` statement, where the statements are evaluated before the code is executed. If true, this explains why `switch` statements are more efficient than `if` statements, as the switch condition is only evaluated once while in the `if` statements, it is evaluated every time we check if we are at the right code. To confirm this explanation, we move to the next step of result analysis and look at the IL code.

```
1  IL_0008: ldloc.0
2  IL_0009: stloc.3
3  IL_000a: ldloc.3
4  IL_000b: stloc.2
5  IL_000c: ldloc.2
6  IL_000d: ldc.i4.1
7  IL_000e: sub
8  IL_000f: switch       (IL_003a, IL_003e, IL_0042, IL_0046,
   ↪  IL_004a, IL_004e, IL_0052, IL_0056, IL_005b)
9  IL_0038: br.s         IL_0060
```

```
10
11 IL_003a: ldc.i4.2
12 IL_003b: stloc.0
13 IL_003c: br.s         IL_0064
14
15 IL_003e: ldc.i4.3
16 IL_003f: stloc.0
17 IL_0040: br.s         IL_0064
18 ...
19 IL_005b: ldc.i4.s     10
20 IL_005d: stloc.0
21 IL_005e: br.s         IL_0064
22
23 IL_0060: ldc.i4.1
24 IL_0061: stloc.0
25 IL_0062: br.s         IL_0064
```

Listing 129: The IL code for the operations in the for loop in the `Switch` benchmark.

In Listing 129 we can see the IL code for the `Switch` benchmark, the important instruction to notice is the `switch` instruction on line `IL_000f`, this instruction does as the documentation specifies, jumping to the relevant address depending on what is on top of the stack.

```
1  IL_000b: ldloc.0
2  IL_000c: ldc.i4.1
3  IL_000d: ceq
4  IL_000f: stloc.2
5  IL_0010: ldloc.2
6  IL_0011: brfalse.s    IL_0018
7  IL_0013: nop
8  IL_0014: ldc.i4.2
9  IL_0015: stloc.0
10 IL_0016: br.s         IL_0094
11
12 IL_0018: ldloc.0
13 IL_0019: ldc.i4.2
14 IL_001a: ceq
```

```
IL_001c: stloc.3
IL_001d: ldloc.3
IL_001e: brfalse.s    IL_0025
IL_0020: nop
IL_0021: ldc.i4.3
IL_0022: stloc.0
IL_0023: br.s         IL_0094
...
IL_0080: ldloc.0
IL_0081: ldc.i4.s     9
IL_0083: ceq
IL_0085: stloc.s      V_10
IL_0087: ldloc.s      V_10
IL_0089: brfalse.s    IL_0091
IL_008b: nop
IL_008c: ldc.i4.s     10
IL_008e: stloc.0
IL_008f: br.s         IL_0094

IL_0091: ldc.i4.1
IL_0092: stloc.0
```

Listing 130: The IL code for the operations in the for loop in the `IfComparableWithSwitch` benchmark.

In Listing 130 we can see the IL code for the `IfComparableWithSwitch` benchmark. It is clear that there is a difference between the two benchmarks, in Listing 130, if `count` is equal to 7, the first 7 if statements are evaluated before code is run, while in Listing 129 if `count` is equal to 7, a jump is executed and the code is run. Considering this difference, it makes sense that `switch` statements are more efficient than `if` statements in cases like this.

The exact same behaviour can be seen for the comparison switch statement vs the equivalent if statement.

Conditional Operator vs If Statements

The first step for looking at the differences between `conditional` operators and `if` statements is looking at the benchmarks.

```
public class SelectionBenchmarks {
    ...
    [Benchmark("SelectionConditional", "Tests if conditional
    ↪ operator")]
    public static int ConditionalOperator() {
        int count = 0;
        for (int i = 0; i < LoopIterations; i++) {
            count = i <= LoopIterations ? 1 : 2;
        }

        return count;
    }
    ...
}
```

Listing 131: The `ConditionalOperator` method which tests `conditional` operators in the `SelectionBenchmarks` class.

```
public class SelectionBenchmarks {
    ...
    [Benchmark("SelectionConditional", "Tests if else comparable
    ↪ with conditional operator")]
    public static int IfElseComparableWithConditional() {
        int count = 0;
        for (int i = 0; i < LoopIterations; i++) {
            if (i <= LoopIterations) {
                count = 1;
            }
            else {
                count = 2;
            }
        }
        return count;
    }
    ...
}
```

Listing 132: The `IfElseComparableWithConditional` method which tests `if` statements that are similar to the `ConditionalOperator` benchmark in the `SelectionBenchmarks` class.

In Listing 131 and Listing 132 we can see that the only difference is the statement used, in Listing 131 a `conditional` operator is used while in Listing 132, an `if` statement is used instead. This is not enough to explain the difference between the benchmarks, furthermore, there is no information in the documentation [114, 115] that can be used to explain the (small) differences between the two benchmarks with regards to their results, therefore we move onto looking at the IL code, the third step of result analysis.

```
1  IL_0008: ldloc.1
2  IL_0009: ldsfld       int32
   ↪  Benchmarks.SelectionBenchmarks::LoopIterations
3
4  IL_000e: ble.s        IL_0013
5  IL_0010: ldc.i4.2
6
7  IL_0011: br.s         IL_0014
8
9  IL_0013: ldc.i4.1
10 IL_0014: stloc.0
```

Listing 133: The IL code for the operations in the for loop in the `ConditionalOperator` benchmark.

In Listing 133 we can see the IL code for the `ConditionalOperator` benchmark. Important to note here is the `ble.s` instruction on line `IL_000e`, which jumps if the <= comparison is true.

```
1  IL_0008: ldloc.1
2  IL_0009: ldsfld       int32
   ↪  Benchmarks.SelectionBenchmarks::LoopIterations
3  IL_000e: cgt
4  IL_0010: ldc.i4.0
```

```
IL_0011: ceq
IL_0013: stloc.2
IL_0014: ldloc.2
IL_0015: brfalse.s    IL_001d

IL_0017: nop
IL_0018: ldc.i4.1
IL_0019: stloc.0
IL_001a: nop

IL_001b: br.s         IL_0021

IL_001d: nop
IL_001e: ldc.i4.2
IL_001f: stloc.0

IL_0020: nop
IL_0021: nop
```

Listing 134: The IL code for the operations in the for loop in the `IfElseComparableWithConditional` benchmark.

In Listing 134 we can see the IL code for the `IfElseComparableWithConditional` benchmark. Considering the differences in the IL code, it is surprising that the difference is so small, considering the `if` statement has approximately double the amount of instructions.A cause of this could be that the `if` statement uses `cgt`, `ceq` and `brfalse.s` instructions instead of `ble.s`. This implies that comparing using greater than and equals is more efficient than comparing using less-than or equals and that the compiler makes this optimization automatically in some cases. To see if we can gain further insight, we look at the assembly code, step four of result analysis.

```
// count = i <= LoopIterations ? 1 : 2;
mov         eax,dword ptr [rbp+38h]
cmp         eax,dword ptr [7FFA40046D34h]
jle         SelectionBenchmarks.ConditionalOperator()+047h
↪  (07FFA401473A7h)
mov         dword ptr [rbp+2Ch],2
```

```
6  jmp          SelectionBenchmarks.ConditionalOperator()+04Eh
   ↪  (07FFA401473AEh)
7  mov          dword ptr [rbp+2Ch],1
8  mov          eax,dword ptr [rbp+2Ch]
9  mov          dword ptr [rbp+3Ch],eax
```

Listing 135: The assembly code for the operations in the for loop in the `ConditionalOperator` benchmark.

In Listing 135 we see the operations done in the `ConditionalOperator` benchmark in assembly code.

```
1  // if (i <= LoopIterations)
2  mov          esi,dword ptr [rbp+38h]
3  mov          rcx,7FFA40046D00h
4  mov          edx,2
5  call         CORINFO_HELP_GETSHARED_NONGCSTATIC_BASE
   ↪  (07FFA9FB3AB90h)
6  cmp          esi,dword ptr [7FFA40046D34h]
7  setle        al
8  movzx        eax,al
9  mov          dword ptr [rbp+34h],eax
10 cmp          dword ptr [rbp+34h],0
11 je
   ↪  SelectionBenchmarks.IfElseComparableWithConditional()+06Bh
   ↪  (07FFA401414FBh)
12 // count = 1
13 mov          dword ptr [rbp+3Ch],1
14 jmp
   ↪  SelectionBenchmarks.IfElseComparableWithConditional()+074h
   ↪  (07FFA40141504h)
15 // else
16 // count = 2
17 mov          dword ptr [rbp+3Ch],2
```

Listing 136: The assembly code for the operations in the for loop in the `IfElseComparableWithConditional` benchmark.

In Listing 136 we can see the benchmark operations in assembly code. Again, we can see that there are extra instructions in Listing 136 to avoid doing a `jle` instruction, furthermore we can see that in Listing 135 the result is saved in a temporary variable until the end where it is assigned to `count` (at address rbp+3Ch). Besides this, we leave further exploration of these differences to future work.

If Statements vs If Else Statements vs If Else If Statements

The first step for looking at the differences between `if` statements, `if else` statements and `if else if` statements is looking at the benchmarks.

```
1  public class SelectionBenchmarks {
2      ...
3      [Benchmark("SelectionIf", "Tests if statement")]
4      public static int If() {
5          int halfLoopIteration = LoopIterations / 2;
6          int count = 0;
7          for (int i = 0; i < LoopIterations; i++) {
8              if (i < halfLoopIteration) {
9                  count++;
10                 continue;
11             }
12             if (i == halfLoopIteration) {
13                 count += 10;
14                 continue;
15             }
16             count--;
17             continue;
18         }
19         return count;
20     }
21     ...
22 }
```

Listing 137: The `If` method which tests `if` statements in the `SelectionBenchmarks` class.

```
public class SelectionBenchmarks {
    ...
    [Benchmark("SelectionIf", "Tests if and else statement")]
    public static int IfElse() {
        int count = 0;
        int halfLoopIteration = LoopIterations / 2;
        for (int i = 0; i < LoopIterations; i++) {
            if (i < halfLoopIteration) {
                count++;
                continue;
            }
            if (i == halfLoopIteration) {
                count += 10;
                continue;
            }
            else {
                count--;
                continue;
            }
        }
        return count;
    }
    ...
}
```

Listing 138: The `IfElse` method which tests `if else` statements in the `SelectionBenchmarks` class.

```
public class SelectionBenchmarks {
    ...
    [Benchmark("SelectionIf", "Tests if and if else statement")]
    public static int IfElseIf()
    {
        int count = 0;
        int halfLoopIteration = LoopIterations / 2;
        for (int i = 0; i < LoopIterations; i++) {
```

```
9               if (i < halfLoopIteration) {
10                  count++;
11                  continue;
12              }
13              else if (i == halfLoopIteration) {
14                  count += 10;
15                  continue;
16              }
17              else {
18                  count--;
19                  continue;
20              }
21          }
22          return count;
23      }
24      ...
25  }
```

Listing 139: The `IfElseIf` method which tests `if else if` statements in the `SelectionBenchmarks` class.

In Listing 137, Listing 138 and Listing 139 we can see that there are no differences besides what we would expect, furthermore, there is no information in the relevant documentation [115] that would explain the differences found in the results. Therefore, we look at the IL code for the relevant benchmarks.

```
1  IL_000f: nop
2  IL_0010: ldloc.2
3  IL_0011: ldloc.1
4  IL_0012: clt
5  IL_0014: stloc.3
6  IL_0015: ldloc.3
7  IL_0016: brfalse.s    IL_001f
8  IL_0018: nop
9  IL_0019: ldloc.0
10 IL_001a: ldc.i4.1
11 IL_001b: add
```

```
IL_001c: stloc.0
IL_001d: br.s         IL_0038
IL_001f: ldloc.2
IL_0020: ldloc.1
IL_0021: ceq
IL_0023: stloc.s      V_4
IL_0025: ldloc.s      V_4
IL_0027: brfalse.s    IL_0031
IL_0029: nop
IL_002a: ldloc.0
IL_002b: ldc.i4.s     10
IL_002d: add
IL_002e: stloc.0
IL_002f: br.s         IL_0038
IL_0031: nop
IL_0032: ldloc.0
IL_0033: ldc.i4.1
IL_0034: sub
IL_0035: stloc.0
IL_0036: br.s         IL_0038
```

Listing 140: The IL code for the operations in the `for` loop for all three benchmarks.

In Listing 140 we can see the IL code for all three benchmarks, the reason only one set of IL code is shown, is because they are all completely equivalent (except for the `nop` instruction on line IL_0031 which does not exist for the `if` benchmark), which further questions why there is a difference in their results.

We check if the assembly code is equivalent as well.

```
// if (i < halfLoopIteration)
mov         eax,dword ptr [rbp+34h]
cmp         eax,dword ptr [rbp+38h]
setl        al
movzx       eax,al
mov         dword ptr [rbp+30h],eax
cmp         dword ptr [rbp+30h],0
je          SelectionBenchmarks.If()+085h (07FFA507B05C5h)
```

```
9  // count++
10 mov         eax,dword ptr [rbp+3Ch]
11 inc         eax
12 mov         dword ptr [rbp+3Ch],eax
13 // continue
14 nop
15 jmp         SelectionBenchmarks.If()+0B2h (07FFA507B05F2h)
16
17 // if (i == halfLoopIteration) / else if (i ==
   ↪  halfLoopIteration)
18 mov         eax,dword ptr [rbp+34h]
19 cmp         eax,dword ptr [rbp+38h]
20 sete        al
21 movzx       eax,al
22 mov         dword ptr [rbp+2Ch],eax
23 cmp         dword ptr [rbp+2Ch],0
24 je          SelectionBenchmarks.If()+0A7h (07FFA507B05E7h)
25 // count += 10
26 mov         eax,dword ptr [rbp+3Ch]
27 add         eax,0Ah
28 mov         dword ptr [rbp+3Ch],eax
29 // continue
30 nop
31 jmp         SelectionBenchmarks.If()+0B2h (07FFA507B05F2h)
32
33 // else
34 nop
35 // count--
36 mov         eax,dword ptr [rbp+3Ch]
37 dec         eax
38 mov         dword ptr [rbp+3Ch],eax
39 // continue
40 nop
41 jmp         SelectionBenchmarks.If()+0B2h (07FFA507B05F2h)
```

Listing 141: The assembly code for the operations in the `for` loop for all three benchmarks.

In Listing 141 we can see the assembly code for all three benchmarks,

again they are all equivalent, therefore we only see one set of assembly code (again, the `else` statement does not exist in the `if` benchmark, therefore that nop does not exist, but besides that they are completely equivalent). Therefore we can conclude that there is some variance in our results that we have not been able to remove, which we will reflect over in Chapter 8.

A.4.3 Collections

Immutable List Get

The first step in analyzing why using `Immutable List get` consumes more energy than all other benchmarks in the `List Get` group except for `LinkedList`, is to look at the benchmark and compare it with the others.

```
1  public class ImmutableListBenchmarks {
2      ...
3      public static readonly ImmutableList<int> Data;
4      static ImmutableListBenchmarks() {
5          Data = ImmutableList<int>.Empty
6          .AddRange(CollectionsHelpers.RandomValues);
7      }
8      [Benchmark("ListGet", "Tests getting values sequentially
   ↪   from a ImmutableList")]
9      public static int ImmutableListGet() {
10         int sum = 0;
11         for (int i = 0; i < LoopIterations; i++) {
12             for (int j = 0; j < Data.Count; j++) {
13                 sum += Data[CollectionsHelpers
                       ↪   .SequentialIndices[j]];
14             }
15         }
16         return sum;
17     }
18     ...
19 }
```

Listing 142: The `ImmutableListGet` method, which tests getting an entry from an `ImmutableList` in the `ImmutableListBenchmarks` class.

```
public class ListBenchmarks {
    ...
    public static readonly List<int> Data = new(1000);
    static ListBenchmarks() {
        foreach (int value in CollectionsHelpers.RandomValues) {
            Data.Add(value);
        }
    }
    [Benchmark("ListGet", "Tests getting values sequentially
    ↪ from a List")]
    public static int ListGet() {
        int sum = 0;
        for (int i = 0; i < LoopIterations; i++) {
            for (int j = 0; j < Data.Count; j++) {
                sum += Data[CollectionsHelpers.
                ↪ SequentialIndices[j]];
            }
        }
        return sum;
    }
    ...
}
```

Listing 143: The `ListGet` method, which tests getting an entry from a `List` in the `ListBenchmarks` class.

Comparing Listing 142 and Listing 143 we see that the only differences are ones we would expect, i.e. that the `ImmutableListGet` benchmarks uses an `Immutablelist` and the `ListGet` benchmark uses a `List`. Otherwise they are identical and as such it does not explain the difference there is between the benchmarks. This leads us to going through the second step of the analysis process and look at the documentation for `ImmutableList` to see if we can explain the results. We have not found anything that explains why there is this difference in performance in the documentation [116], therefore we look at the the IL code, as the next step of the process.

```
IL_000a: ldloc.0
```

```
2 IL_000b: ldsfld       class
  ↪ [System.Collections.Immutable]System.Collections.Immutable.
  ↪ ImmutableList`1<int32>Benchmarks.Collections
  ↪ .List.ImmutableListBenchmarks::Data
3 IL_0010: ldsfld       int32[] Benchmarks.Collections.
  ↪ CollectionsHelpers::SequentialIndices
4 IL_0015: ldloc.2
5 IL_0016: ldelem.i4
6 IL_0017: callvirt     instance !0/*int32*/ class
  ↪ [System.Collections.Immutable]System.Collections.Immutable
  ↪ .ImmutableList`1<int32>::get_Item(int32)
7 IL_001c: add
8 IL_001d: stloc.0
```

Listing 144: The IL code for the operations in the for loop in the `ImmutableListGet` benchmark.

```
1 IL_000a: ldloc.0
2 IL_000b: ldsfld       class
  ↪ [System.Collections]System.Collections.Generic.List`1<int32>
  ↪ Benchmarks.Collections.List.ListBenchmarks::Data
3 IL_0010: ldsfld       int32[]
  ↪ Benchmarks.Collections.CollectionsHelpers::SequentialIndices
4 IL_0015: ldloc.2
5 IL_0016: ldelem.i4
6 IL_0017: callvirt     instance !0/*int32*/ class
  ↪ [System.Collections]System.Collections
  ↪ .Generic.List`1<int32>::get_Item(int32)
7 IL_001c: add
8 IL_001d: stloc.0
```

Listing 145: The IL code for the operations in the for loop in the `ListGet` benchmark.

In Listing 144 we see the IL code for the `ImmutableListGet` benchmark. Comparing it with Listing 145, we can see that there is no difference in the operations performed between the IL code, the only difference between these two are the types of lists that are utilized. Because of this we go onto step four of the analysis process and look at the assembly code.

```
// sum += Data[CollectionsHelpers.SequentialIndices[j]];
mov         edx,dword ptr [rbp+4Ch]
mov         dword ptr [rbp+2Ch],edx
mov         rdx,28025762EA0h
mov         rdx,qword ptr [rdx]
mov         ecx,dword ptr [rbp+44h]
cmp         ecx,dword ptr [rdx+8]
jb          ImmutableListBenchmarks.ImmutableListGet()+074h
    (07FFA0F0F3FE4h)
call        00007FFA6ED7E750
mov         eax,ecx
lea         rdx,[rdx+rax*4+10h]
mov         edx,dword ptr [rdx]
mov         rcx,28025762E98h
mov         rcx,qword ptr [rcx]
cmp         dword ptr [rcx],ecx
call        qword ptr [Pointer to:
    CLRStub[MethodDescPrestub]@7ffa0f0f3868 (07FFA0F2958E0h)]
mov         dword ptr [rbp+28h],eax
mov         eax,dword ptr [rbp+2Ch]
add         eax,dword ptr [rbp+28h]
mov         dword ptr [rbp+4Ch],eax
```

Listing 146: The assembly code for the `ImmutableListGet` operation.

```
// sum += Data[CollectionsHelpers.SequentialIndices[j]];
mov         ecx,dword ptr [rbp+5Ch]
mov         dword ptr [rbp+3Ch],ecx
mov         rcx,28025762E90h
mov         rcx,qword ptr [rcx]
mov         qword ptr [rbp+28h],rcx
mov         rcx,7FFA0F196D48h
mov         edx,7
call        CORINFO_HELP_GETSHARED_NONGCSTATIC_BASE
    (07FFA6EC5AB90h)
mov         rdx,28025762EA0h
mov         rdx,qword ptr [rdx]
```

```
12 mov          ecx,dword ptr [rbp+54h]
13 cmp          ecx,dword ptr [rdx+8]
14 jb           ListBenchmarks.ListGet()+0A3h (07FFA0F0F06B3h)
15 call         00007FFA6ED7E750
16 mov          eax,ecx
17 lea          rdx,[rdx+rax*4+10h]
18 mov          edx,dword ptr [rdx]
19 mov          dword ptr [rbp+34h],edx
20 mov          edx,dword ptr [rbp+34h]
21 mov          rcx,qword ptr [rbp+28h]
22 cmp          dword ptr [rcx],ecx
23 call         qword ptr [Pointer to:
   ↪ CLRStub[MethodDescPrestub]@7ffa0f0e0a98 (07FFA0F1E9A80h)]
24 mov          dword ptr [rbp+38h],eax
25 mov          eax,dword ptr [rbp+3Ch]
26 add          eax,dword ptr [rbp+38h]
27 mov          dword ptr [rbp+5Ch],eax
```

Listing 147: The assembly code for the `ListGet` operation.

In Listing 146 and Listing 147 we see the assembly for the two benchmarks. We see there are more operations in the `ListGet`, both some `mov` operations, as well as a call to `CORINFO_HELP_GETSHARED_NONGCSTATIC_BASE`. The added operations does not give an explanation as to why `ImmutableList` is less efficient, and if anything would point in the other direction.

We leave further exploration into this to future work, as it requires knowledge of the C# compiler.

Linked and Immutable List Removal

To determine why both `Linked List Removal` and `Immutable List Removal` consumes a third of the energy the other constructs we first look at the benchmarks themselves to see if there is a reason for the differences here.

```
1 public class LinkedListBenchmarks {
2     ...
3         public static readonly LinkedList<int> Data = new();
4         static LinkedListBenchmarks() {
```

```
5                 foreach (int value in
                  ↪  CollectionsHelpers.RandomValues) {
6                         Data.AddLast(value);
7                 }
8         }
9     ...
10        [Benchmark("ListRemoval", "Tests removal from a
  ↪  LinkedList")]
11        public static int LinkedListRemoval() {
12                int result = 0;
13                LinkedList<int> target = new();
14                for (int i = 0; i < LoopIterations; i++) {
15                        for (int j = 0; j < Data.Count; j++) {
16                                target.AddLast(j);
17                        }
18                        for (int index = (Data.Count - 1) / 2;
                          ↪  index >= 0; index--) {
19                                target.Remove(target.ElementAt(index));
20                        }
21                        result += target.Count;
22                }
23                return result;
24        }
25    ...
26 }
```

Listing 148: The `LinkedListRemoval` method, which tests getting an entry from an `LinkedList` in the `LinkedListBenchmarks` class.

```
1  public class ImmutableListBenchmarks {
2      ...
3          public static readonly ImmutableList<int> Data;
4          static ImmutableListBenchmarks() {
5                  Data = ImmutableList<int>.Empty
6                  .AddRange(CollectionsHelpers.RandomValues);
7          }
8      ...
```

```
9           [Benchmark("ListRemoval", "Tests removal from a
    ↪  ImmutableList")]
10          public static int ImmutableListRemoval() {
11                  int result = 0;
12                  ImmutableList<int> target =
                    ↪  ImmutableList<int>.Empty;
13                  for (int i = 0; i < LoopIterations; i++) {
14                          for (int j = 0; j < Data.Count; j++) {
15                                  target = target.Add(j);
16                          }
17                          for (int index = (Data.Count - 1) / 2;
                            ↪  index >= 0; index--) {
18                                  target = target.RemoveAt(index);
19                          }
20                          result += target.Count;
21                  }
22                  return result;
23          }
24      ...
25  }
```

Listing 149: The `ImmutableListRemoval` method, which tests getting an entry from an `ImmutableList` in the `ImmutableListBenchmarks` class.

```
1   public class ListBenchmarks {
2       ...
3           public static readonly List<int> Data = new(1000);
4           static ListBenchmarks() {
5                   foreach (int value in
                    ↪  CollectionsHelpers.RandomValues) {
6                           Data.Add(value);
7                   }
8           }
9       ...
10          [Benchmark("ListRemoval", "Tests removal from a List")]
11          public static int ListRemoval() {
12                  int result = 0;
```

```
13                  List<int> target = new List<int>();
14                  for (int i = 0; i < LoopIterations; i++) {
15                          for (int j = 0; j < Data.Count; j++) {
16                                  target.Add(j);
17                          }
18                          for (int index = (Data.Count - 1) / 2;
                             ↪  index >= 0; index--) {
19                                  target.RemoveAt(index);
20                          }
21                          result += target.Count;
22                  }
23                  return result;
24          }
25      ...
26  }
```

Listing 150: The `ListRemoval` method, which tests getting an entry from an `List` in the `ListBenchmarks` class.

In Listing 148 and Listing 149 we see minor differences where most are as expected, i.e. that the `ImmutableListRemoval` benchmark uses an `ImmutableList`, `LinkedListRemoval` a `LinkedList` and the `ListRemoval` benchmark uses a `List`. There is also a difference in how elements are added to the lists. While adding elements to these lists is not the focus of the benchmark it may have influenced the results and therefor we look at it. According to [140] adding elements to an `ImmutableList` returns a new `ImmutableList`, which would imply that this should be consume more energy and be slower. While this is not the case here for the removal benchmarks, we can see in Table A.85 how that intuition matches reality. However it seems that removal is a significantly more energy intensive and makes this adding of elements negligible. Table A.85 also show how both `LinkedList` and `List` have relatively good energy consumption, as such we do not look more into the effect of adding elements.

`LinkedList` removal is done with `target.Remove(target.ElementAt(index));`. According to [117] "This method is an O(1) operation.", which explains the relatively low energy consumption compared to other constructs in the group. There is a difference once more when looking at how elements are removed, `ImmutableList` we utilize `target = target.RemoveAt(index);`.

According to [141] it returns "A new list with the element removed.", which could imply greater energy consumption by construction of a new list, however as this is note the case the documentation does not explain the behavior. `List` uses `target.RemoveAt(index);`, which according to [142] is "This method is an O(n) operation...", which tells us why this might be more intensive that the `LinkedList`. As the behavior of removal from these list have not been explained by the documentation, we move onto step three of the analysis and look at the IL code.

```
1 IL_003b: ldloc.1      // target
2 IL_003c: ldloc.1      // target
3 IL_003d: ldloc.s      index
4 IL_003f: call         !!0/*int32*/
  ↪ [System.Linq]System.Linq.Enumerable::ElementAt<int32>(class
  ↪ [System.Runtime]System.Collections.Generic.IEnumerable`1<!!0/*int32*/>,
  ↪ int32)
5 IL_0044: callvirt     instance bool class
  ↪ [System.Collections]System.Collections.Generic.
  ↪ LinkedList`1<int32>::Remove(!0/*int32*/)
6 IL_0049: pop
```

Listing 151: The IL code for the remove operation in the for loop in the `LinkedListRemoval` benchmark.

```
1 IL_003b: ldloc.1      // target
2 IL_003c: ldloc.s      index
3 IL_003e: callvirt     instance class
  ↪ [System.Collections.Immutable]System.Collections.
  ↪ Immutable.ImmutableList`1<!0/*int32*/> class
  ↪ [System.Collections.Immutable]System.Collections.
  ↪ Immutable.ImmutableList`1<int32>::RemoveAt(int32)
4 IL_0043: stloc.1      // target
```

Listing 152: The IL code for the remove operation in the for loop in the `ImmutableListRemoval` benchmark.

```
IL_003a: ldloc.1      // target
IL_003b: ldloc.s      index
IL_003d: callvirt     instance void class
    [System.Collections]System.Collections.
    Generic.List`1<int32>::RemoveAt(int32)
```

Listing 153: The IL code for the remove operation in the for loop in the `ListRemoval` benchmark.

In `cs:LinkedListRemovalIL`, `cs:ImmutableListRemovalIL` and `cs:ListRemovalIL`, we see some differences which could explain the different energy consumption. Looking at `cs:ImmutableListRemovalIL` and `cs:ListRemovalIL` we see the only differences between the two benchmarks, besides the type of the table, is the `stloc.1` instruction in `cs:ImmutableListRemovalIL`. This value pops a value from stack into local variable 1, and the value is assumed to be the new `ImmutableList`. This does not however explain why `ImmutableList` removal consumes less energy than `List` removal, as as such we must look at the assembly code for and explanation of this difference.

Comparing texttttcs:LinkedListRemovalIL and `cs:ListRemovalIL`, we see a more stark difference, beyond the type of the table. There is an extra `ldloc.1` instruction, next we have the `call` instruction calling the `ElementAt` method and lastly the `pop` instruction. Having these extra instruction would imply higher energy consumption, however that is not the case and therefore we move to step four and look at the assembly code.

```
// target.Remove(target.ElementAt(index));
mov         rcx,qword ptr [rbp+80h]
mov         qword ptr [rbp+38h],rcx
mov         rcx,qword ptr [rbp+80h]
mov         edx,dword ptr [rbp+70h]
call        CLRStub[MethodDescPrestub]@7ffa0f0e05d8
    (07FFA0F0E05D8h)
mov         dword ptr [rbp+34h],eax
mov         rcx,qword ptr [rbp+38h]
mov         edx,dword ptr [rbp+34h]
cmp         dword ptr [rcx],ecx
call        qword ptr [Pointer to:
    CLRStub[MethodDescPrestub]@7ffa0f0e0830 (07FFA0F25A770h)]
```

```
12 mov         dword ptr [rbp+30h],eax
13 nop
```

Listing 154: The assembly code for the relevant code in the `LinkedListRemoval` benchmark.

```
1 // target = target.RemoveAt(index);
2 mov         rcx,qword ptr [rbp+70h]
3 mov         edx,dword ptr [rbp+60h]
4 cmp         dword ptr [rcx],ecx
5 call        CLRStub[MethodDescPrestub]@7ffa0f285b48
  ↪ (07FFA0F285B48h)
6 mov         qword ptr [rbp+30h],rax
7 mov         rcx,qword ptr [rbp+30h]
8 mov         qword ptr [rbp+70h],rcx
```

Listing 155: The relevant code for the `ImmutableListRemoval` benchmark.

```
1 // target.RemoveAt(index);
2 mov         rcx,qword ptr [rbp+60h]
3 mov         edx,dword ptr [rbp+50h]
4 cmp         dword ptr [rcx],ecx
5 call        qword ptr [Pointer to:
  ↪ CLRStub[MethodDescPrestub]@7ffa0f0d0c30 (07FFA0F1D9B18h)]
6 nop
```

Listing 156: The relevant code for the `ListRemoval` benchmark.

Comparing Listing 154 and Listing 156 we can see that the assembly code for `LinkedList` has several more `mov` instructions and even an extra `call` instruction, this makes `LinkedList`'s performance unexpected as the extra struction implies greater energy consumption. The explanation might lie in the documentation, as described earlier, but further analysis is needed to be certain, this is left for future work. Comparing Listing 155 and Listing 156, we see several more `mov` instructions for `ImmutableList`, again making the performance of `ImmutableList` compared to `List` unexpected, further analysis of this is relegated to future work.

Hashtable Get & Get Randoml

To find out why `Hashtable Get` and `Hashtable Get Random` consumes more than three times as much energy as `Dictionary Get` and `Dictionary Get Random` we follow the procedure for result analysis. First we look at the benchmarks to determine if there is difference here that could explain the changes. We start by looking at `Hashtable Get` and `Dictionary Get`.

```
1  public class HashtableBenchmarks {
2      ...
3      public static readonly Hashtable Data = new(1000);
4          static HashtableBenchmarks() {
5                  foreach ((int index, int value) in
                   ↪ CollectionsHelpers.RandomValues.WithIndex())
                   ↪ {
6                          Data.Add(index, value);
7                  }
8          }
9      ...
10         [Benchmark("TableGet", "Tests getting values
   ↪ sequentially from a Hashtable")]
11         public static int HashtableGet() {
12                 int sum = 0;
13                 for (int i = 0; i < LoopIterations; i++) {
14                         for (int j = 0; j < Data.Count; j++) {
15                                 sum +=
                                   ↪ (int)Data[CollectionsHelpers
16                                 .SequentialIndices[j]];
17                         }
18                 }
19                 return sum;
20         }
21     ...
22 }
```

Listing 157: The `HashtableGet` method, which tests getting an entry from a `ImmutableList` in the `HashtableBenchmarks` class.

```
1  public class DictionaryBenchmarks {
2      ...
3      public static readonly Dictionary<int, int> Data =
       ↪  new(1000);
4          static DictionaryBenchmarks() {
5                  foreach ((int index, int value) in
                   ↪  CollectionsHelpers.RandomValues.WithIndex())
                   ↪  {
6                          Data.Add(index, value);
7                  }
8          }
9      ...
10         [Benchmark("TableGet", "Tests getting values
   ↪   sequentially from a Dictionary")]
11         public static int DictionaryGet() {
12                 int sum = 0;
13                 for (int i = 0; i < LoopIterations; i++) {
14                         for (int j = 0; j < Data.Count; j++) {
15                                 sum += Data[CollectionsHelpers
16                                 .SequentialIndices[j]];
17                         }
18                 }
19                 return sum;
20         }
21     ...
22 }
```

Listing 158: The `DictionaryGet` method, which tests getting an entry from a `Dictionary` in the `DictionaryBenchmarks` class.

In Listing 157 and Listing 158 there is one difference beyond what was expected, which is the casting to an `int` in Listing 157 on line 15, where the value is unboxed. According to [118] unboxing is a computationally expensive processes and "When unboxing, the casting process can take four times as long as an assignment.", which explains the increased time and energy consumption needed by `Hashtable Get` compared to`Dictionary Get`. As `Hashtable Get Random` also utilizes unboxing and `Dictionary Get Random` does not, we can conclude that the increase in energy consumption in this

case is also based on the extra computations needed for unboxing.

Read Only Dictionary Get vs Dictionary Get

The first step in figuring out why the `Get` method on `ReadOnlyDictionary` consumes more energy than the `Get` method on `Dictionary`, is looking at the benchmarks.

```
1  public class ReadOnlyDictionaryBenchmarks {
2      ...
3      public static readonly ReadOnlyDictionary<int, int> Data;
4      static ReadOnlyDictionaryBenchmarks() {
5          Data = new ReadOnlyDictionary<int,
           ↪  int>(CollectionsHelpers.RandomValues.WithIndex()
6              .ToDictionary(tuple => tuple.index, tuple =>
               ↪  tuple.value));
7      }
8      ...
9      [Benchmark("TableGet", "Tests getting values sequentially
   ↪   from a ReadOnlyDictionary")]
10     public static int ReadOnlyDictionaryGet() {
11         int sum = 0;
12         for (int i = 0; i < LoopIterations; i++) {
13             for (int j = 0; j < Data.Count; j++) {
14                 sum +=
                   ↪  Data[CollectionsHelpers.SequentialIndices[j]];
15             }
16         }
17         return sum;
18     }
19     ...
20 }
```

Listing 159: The `ReadOnlyDictionaryGet` method which tests getting an integer from a `ReadOnlyDictionary` in the `ReadOnlyDictionaryBenchmarks` class.

```
1  public class DictionaryBenchmarks {
2      ...
```

```
3       public static readonly Dictionary<int, int> Data =
        ↪ new(1000);
4       static DictionaryBenchmarks() {
5           foreach ((int index, int value) in
            ↪ CollectionsHelpers.RandomValues.WithIndex()) {
6               Data.Add(index, value);
7           }
8       }
9       ...
10      [Benchmark("TableGet", "Tests getting values sequentially
    ↪   from a Dictionary")]
11      public static int DictionaryGet() {
12          int sum = 0;
13          for (int i = 0; i < LoopIterations; i++) {
14              for (int j = 0; j < Data.Count; j++) {
15                  sum +=
                    ↪ Data[CollectionsHelpers.SequentialIndices[j]];
16              }
17          }
18          return sum;
19      }
20      ...
21  }
```

Listing 160: The `DictionaryGet` method which tests getting an integer from a `ReadOnlyDictionary` in in the `DictionaryBenchmarks` class.

In Listing 159 and Listing 160 we can see that the only differences are the ones we would expect, i.e. that the `ReadOnlyDictionaryGet` benchmark uses a `ReadOnlyDictionary` and the `DictionaryGet` uses an ordinary `Dictionary`. There is no information in the documentation [119, 120] that explains the difference in performance between these two types, therefore we look at the IL code.

```
1   IL_000d: ldloc.0
2   IL_000e: ldsfld       class
    ↪ [System.ObjectModel]System.Collections.ObjectModel.ReadOnlyDictionary`2<int32,
    ↪ int32>
    ↪ Benchmarks.Collections.Table.ReadOnlyDictionaryBenchmarks::Data
```

```
IL_0013: ldsfld       int32[]
↪  Benchmarks.Collections.CollectionsHelpers::SequentialIndices
IL_0018: ldloc.2
IL_0019: ldelem.i4
IL_001a: callvirt     instance !1/*int32*/ class
↪  [System.ObjectModel]System.Collections.ObjectModel.ReadOnlyDictionary`2<int32,
↪  int32>::get_Item(!0/*int32*/)
IL_001f: add
IL_0020: stloc.0
```

Listing 161: The IL code for the operations in the for loop in the `ReadOnlyDictionaryGet` benchmark.

In Listing 161 we can see the IL code for the `ReadOnlyDictionaryGet` benchmark.

```
IL_000d: ldloc.0
IL_000e: ldsfld       class
↪  [System.Collections]System.Collections.Generic.Dictionary`2<int32,
↪  int32>
↪  Benchmarks.Collections.Table.DictionaryBenchmarks::Data
IL_0013: ldsfld       int32[]
↪  Benchmarks.Collections.CollectionsHelpers::SequentialIndices
IL_0018: ldloc.2
IL_0019: ldelem.i4
IL_001a: callvirt     instance !1/*int32*/ class
↪  [System.Collections]System.Collections.Generic.Dictionary`2<int32,
↪  int32>::get_Item(!0/*int32*/)
IL_001f: add
IL_0020: stloc.0
```

Listing 162: The IL code for the operations in the for loop in the `DictionaryGet` benchmark.

In Listing 162 we can see the IL code for the `DictionaryGet` benchmark. Here we can see that there are no differences between the two benchmarks, besides the type of the table, meaning we need to look into the assembly code to get further insight.

```
// sum += Data[CollectionsHelpers.SequentialIndices[j]];
mov         edx,dword ptr [rbp+4Ch]
mov         dword ptr [rbp+2Ch],edx
mov         rdx,14FD9612EA0h
mov         rdx,qword ptr [rdx]
mov         ecx,dword ptr [rbp+44h]
cmp         ecx,dword ptr [rdx+8]
jb
    ReadOnlyDictionaryBenchmarks.ReadOnlyDictionaryGet()+074h
    (07FFA95B85714h)
call        00007FFAF565E750
mov         eax,ecx
lea         rdx,[rdx+rax*4+10h]
mov         edx,dword ptr [rdx]
mov         rcx,14FD9612E98h
mov         rcx,qword ptr [rcx]
cmp         dword ptr [rcx],ecx
call        qword ptr [Pointer to:
    CLRStub[MethodDescPrestub]@7ffa95b85330 (07FFA95C0B148h)]
mov         dword ptr [rbp+28h],eax
mov         eax,dword ptr [rbp+2Ch]
add         eax,dword ptr [rbp+28h]
mov         dword ptr [rbp+4Ch],eax
```

Listing 163: The assembly code for the relevant code in the `ReadOnlyDictionaryGet` benchmark.

In Listing 163 we can see the assembly code for the `ReadOnlyDictionaryGet` benchmark.

```
// sum += Data[CollectionsHelpers.SequentialIndices[j]];
mov         ecx,dword ptr [rbp+5Ch]
mov         dword ptr [rbp+3Ch],ecx
mov         rcx,14FD9612E90h
mov         rcx,qword ptr [rcx]
mov         qword ptr [rbp+28h],rcx
mov         rcx,7FFA95A76D50h
```

```
8  mov          edx,7
9  call         CORINFO_HELP_GETSHARED_NONGCSTATIC_BASE
   ↪  (07FFAF553AB90h)
10 mov          rdx,14FD9612EA0h
11 mov          rdx,qword ptr [rdx]
12 mov          ecx,dword ptr [rbp+54h]
13 cmp          ecx,dword ptr [rdx+8]
14 jb           DictionaryBenchmarks.DictionaryGet()+0A3h
   ↪  (07FFA959D09E3h)
15 call         00007FFAF565E750
16 mov          eax,ecx
17 lea          rdx,[rdx+rax*4+10h]
18 mov          edx,dword ptr [rdx]
19 mov          dword ptr [rbp+34h],edx
20 mov          edx,dword ptr [rbp+34h]
21 mov          rcx,qword ptr [rbp+28h]
22 cmp          dword ptr [rcx],ecx
23 call         qword ptr [Pointer to:
   ↪  CLRStub[MethodDescPrestub]@7ffa959d0328 (07FFA95B59680h)]
24 mov          dword ptr [rbp+38h],eax
25 mov          eax,dword ptr [rbp+3Ch]
26 add          eax,dword ptr [rbp+38h]
27 mov          dword ptr [rbp+5Ch],eax
```

Listing 164: The assembly code for the relevant code in the `DictionaryGet` benchmark.

In Listing 164 we can see the assembly code for the `DictionaryGet` benchmark. Here we see that the `DictionaryGet` benchmark has an extra `call` instruction among other extra instructions, which makes the performance of `DictionaryGet` unexpected, further analysis of this is needed, which we leave for future work.

A.4.4 String Concatenation

Similar Approaches

The first step for examining the differences between similar approaches to `string` concatenation is looking at the benchmarks. In this case, we look

at `string.Format` and `string` interpolation, as these approaches are similar but have a large difference in results.

```
public class StringBenchmarks {
    ...
    [Benchmark("StringConcat", "Tests string.Format")]
    public static string Format() {
        string str = "";
        string iStr = "I ";
        string am = "am ";
        string a = "a ";
        string stringStr = "string ";
        string with = "with ";
        string integer = "integer ";
        int myInt = 42;
        for (int i = 0; i < LoopIterations; i++) {
            str = "";
            str = string.Format("{0}{1}{2}{3}{4}{5}{6}", iStr,
            ↪  am, a, stringStr, with, integer, myInt);
        }
        return str;
    }
    ...
}
```

Listing 165: The `Format` method which tests `string` concatenation utilizing the `string.Fomat` method in the `StringBenchmarks` class.

```
public class StringBenchmarks {
    ...
    [Benchmark("StringConcat", "Tests string interpolation")]
    public static string Interpolation() {
        string str = "";
        string iStr = "I ";
        string am = "am ";
        string a = "a ";
```

```
9           string stringStr = "string ";
10          string with = "with ";
11          string integer = "integer ";
12          int myInt = 42;
13          for (int i = 0; i < LoopIterations; i++) {
14              str = "";
15              str =
                ↪  $"{iStr}{am}{a}{stringStr}{with}{integer}{myInt}";
16          }
17          return str;
18      }
19      ...
20  }
```

Listing 166: The `Interpolation` method which tests `string` concatenation utilizing the `string` interpolation in the `StringBenchmarks` class.

In Listing 165 and Listing 166 we can see that the only difference is that Listing 165 utilizes `string.Format` while Listing 166 utilizes `string` interpolation. In [121] it is written that `string` interpolation is recommended over `string.Format` because it is more flexible and readable, however this does not explain why `string` interpolation is more efficient. In [122] it is also written that `string` interpolation is more readable than `string.Format`. Besides this, there is limited relevant information regarding `string.Format` vs `string` interpolation, therefore we examine the IL code between the two benchmarks.

```
1   IL_0038: ldstr       ""
2   IL_003d: stloc.0
3   IL_003e: ldstr       "{0}{1}{2}{3}{4}{5}{6}"
4   IL_0043: ldc.i4.7
5   IL_0044: newarr      [System.Runtime]System.Object
6   IL_0049: dup
7
8   IL_004a: ldc.i4.0
9   IL_004b: ldloc.1
10  IL_004c: stelem.ref
11  IL_004d: dup
```

```
...
IL_0065: ldc.i4.6
IL_0066: ldloc.s      myInt
IL_0068: box          [System.Runtime]System.Int32
IL_006d: stelem.ref
IL_006e: call         string
    [System.Runtime]System.String::Format(string, object[])
IL_0073: stloc.0
```

Listing 167: The IL code for the operations in the for loop in the `Format` benchmark.

In Listing 167 we can see the IL code for the `Format` benchmark.

```
IL_0038: ldstr        ""
IL_003d: stloc.0
IL_003e: ldloca.s     V_9
IL_0040: ldc.i4.0
IL_0041: ldc.i4.7
IL_0042: call         instance void
    [System.Runtime]System.Runtime.
    CompilerServices.DefaultInterpolatedStringHandler::
    .ctor(int32, int32)

IL_0047: ldloca.s     V_9
IL_0049: ldloc.1
IL_004a: call         instance void
    [System.Runtime]System.Runtime.
    CompilerServices.DefaultInterpolatedStringHandler::
    AppendFormatted(string)
IL_004f: nop
...
IL_0080: ldloca.s     V_9
IL_0082: ldloc.s      myInt
IL_0084: call         instance void
    [System.Runtime]System.Runtime.
    CompilerServices.DefaultInterpolatedStringHandler::
    AppendFormatted<int32>(!!0/*int32*/)
IL_0089: nop
```

```
17
18 IL_008a: ldloca.s     V_9
19 IL_008c: call         instance string
   ↪  [System.Runtime]System.Runtime.
   ↪  CompilerServices.DefaultInterpolatedStringHandler::
   ↪  ToStringAndClear()
20 IL_0091: stloc.0
```

Listing 168: The IL code for the operations in the for loop in the `Interpolation` benchmark.

In Listing 168 we can see the IL code for the `Interpolation` benchmark. We discover that there are differences between the compiled implementations of the benchmarks. The differences explains why there is a difference between the two benchmarks.

The `string.Format` method creates an array to keep track of all the different elements we want to put into the string. It does this by replacing elements in the array and duplicating the reference to it. After this is done, at the end the built-in `Format` method on `System.String` is called which creates the final string.

The `string` interpolation works by creating a temporary `DefaultInterpolatedStringHandler` object. The object is used together with each of the objects we want to insert individually, in a call to its built-in `AppendFormatted` method. After this is done, the built-in `ToStringAndClear` method is called on the temporary object which creates the final string.

These differences show that the differences in performance is explained by how the compiler implements the `string` interpolation. To see if we can get further insight, we look at the assembly code.

```
1 // str = "";
2 mov         rcx,11BE4983020h
3 mov         rcx,qword ptr [rcx]
4 mov         qword ptr [rbp+88h],rcx
5 // str = string.Format("{0}{1}{2}{3}{4}{5}{6}", iStr, am, a,
   ↪  stringStr, with, integer, myInt);
6 mov         rcx,11BE49861E0h
7 mov         rcx,qword ptr [rcx]
8 mov         qword ptr [rbp+38h],rcx
```

```
9  mov         rcx,7FFA0F09B578h
10 mov         edx,7
11 call        CORINFO_HELP_NEWARR_1_OBJ (07FFA6EC5AB40h)
12 mov         qword ptr [rbp+30h],rax
13 mov         rcx,qword ptr [rbp+30h]
14 mov         r8,qword ptr [rbp+80h]
15 xor         edx,edx
16 call
   ↪ System.Runtime.CompilerServices.CastHelpers.StelemRef(System.Array,
   ↪ Int32, System.Object) (07FFA0F0A8190h)
17 mov         rcx,qword ptr [rbp+30h]
18 mov         r8,qword ptr [rbp+78h]
19 mov         edx,1
20 call
   ↪ System.Runtime.CompilerServices.CastHelpers.StelemRef(System.Array,
   ↪ Int32, System.Object) (07FFA0F0A8190h)
21 ...
22 mov         rcx,7FFA0F139480h
23 call        CORINFO_HELP_NEWSFAST (07FFA6EC5A9B0h)
24 mov         qword ptr [rbp+28h],rax
25 mov         r8,qword ptr [rbp+28h]
26 mov         ecx,dword ptr [rbp+54h]
27 mov         dword ptr [r8+8],ecx
28 mov         r8,qword ptr [rbp+28h]
29 mov         rcx,qword ptr [rbp+30h]
30 mov         edx,6
31 call
   ↪ System.Runtime.CompilerServices.CastHelpers.StelemRef(System.Array,
   ↪ Int32, System.Object) (07FFA0F0A8190h)
32 mov         rcx,qword ptr [rbp+38h]
33 mov         rdx,qword ptr [rbp+30h]
34 call        CLRStub[MethodDescPrestub]@7ffa0f0a8810
   ↪ (07FFA0F0A8810h)
35 mov         qword ptr [rbp+20h],rax
36 mov         rax,qword ptr [rbp+20h]
37 mov         qword ptr [rbp+88h],rax
```

Listing 169: The assembly code for the operations in the for loop in the `Format` benchmark.

In Listing 169 we can see the assembly code for the `Format` benchmark.

```
// str = "";
mov         rcx,15D18003020h
mov         rcx,qword ptr [rcx]
mov         qword ptr [rbp+98h],rcx
// str = $"{iStr}{am}{a}{stringStr}{with}{integer}{myInt}";
lea         rcx,[rbp+38h]
xor         edx,edx
mov         r8d,7
call        CLRStub[MethodDescPrestub]@7ffa0f0e0208
 ↪  (07FFA0F0E0208h)
lea         rcx,[rbp+38h]
mov         rdx,qword ptr [rbp+90h]
call        CLRStub[MethodDescPrestub]@7ffa0f0e0290
 ↪  (07FFA0F0E0290h)
nop
lea         rcx,[rbp+38h]
mov         rdx,qword ptr [rbp+88h]
call        CLRStub[MethodDescPrestub]@7ffa0f0e0290
 ↪  (07FFA0F0E0290h)
nop
...
lea         rcx,[rbp+38h]
call        CLRStub[MethodDescPrestub]@7ffa0f0e0238
 ↪  (07FFA0F0E0238h)
mov         qword ptr [rbp+20h],rax
mov         rax,qword ptr [rbp+20h]
mov         qword ptr [rbp+98h],rax
```

Listing 170: The assembly code for the operations in the for loop in the `Interpolation` benchmark.

In Listing 170 we can see the assembly code for the `Interpolation` benchmark. Here we can see that there are differences in the benchmarks, like we could see in the IL code. We can see that different built-in methods

are called for both of these benchmarks which is likely to be the reason between the performance differences. To get further insight into this, we need insight into the C# compiler, which we leave for future work.

StringBuilder

To figure out why `StringBuilder` is the most efficient method of concatenating strings, we examine the benchmark compared to the second most efficient method of concatenating (non-constant) strings, in this case interpolation.

```
public class StringBenchmarks {
    ...
    [Benchmark("StringConcat", "Tests operation on
    ↪ stringbuilder")]
    public static string StringBuilder() {
        StringBuilder sb = new StringBuilder();
        string str = "";
        string iStr = "I ";
        string am = "am ";
        string a = "a ";
        string stringStr = "string ";
        string with = "with ";
        string integer = "integer ";
        int myInt = 42;
        for (int i = 0; i < LoopIterations; i++) {
            str = "";
            sb.Clear();
            sb.Append(iStr);
            sb.Append(am);
            sb.Append(a);
            sb.Append(stringStr);
            sb.Append(with);
            sb.Append(integer);
            sb.Append(myInt);
            str = sb.ToString();
        }
        return str;
    }
```

```
28      ...
29  }
```

Listing 171: The `Format` method which tests `string` concatenation utilizing the `string.Fomat` method in the `StringBenchmarks` class.

```
1   public class StringBenchmarks {
2       ...
3       [Benchmark("StringConcat", "Tests string interpolation")]
4       public static string Interpolation() {
5           string str = "";
6           string iStr = "I ";
7           string am = "am ";
8           string a = "a ";
9           string stringStr = "string ";
10          string with = "with ";
11          string integer = "integer ";
12          int myInt = 42;
13          for (int i = 0; i < LoopIterations; i++) {
14              str = "";
15              str =
                ↪  $"{iStr}{am}{a}{stringStr}{with}{integer}{myInt}";
16          }
17          return str;
18      }
19      ...
20  }
```

Listing 172: The `Interpolation` method which tests `string` concatenation utilizing the `string` interpolation in the `StringBenchmarks` class.

In Listing 171 and Listing 172 we can see the only difference is that Listing 171 utilizes a `StringBuilder` while Listing 172 utilizes interpolation with `strings`. In [123] there is information regarding how the memory is allocated when using a `StringBuilder`, but there is no information which explains the difference in efficiency between the `StringBuilder` and

`Interpolation` benchmarks. We therefore examine the benchmarks' IL code.

```
IL_003f: ldstr      ""
IL_0044: stloc.1

IL_0045: ldloc.0
IL_0046: callvirt   instance class
    ↪ [System.Runtime]System.Text.StringBuilder
    ↪ [System.Runtime]System.Text.StringBuilder::Clear()
IL_004b: pop

IL_004c: ldloc.0
IL_004d: ldloc.2
IL_004e: callvirt   instance class
    ↪ [System.Runtime]System.Text.StringBuilder
    ↪ [System.Runtime]System.Text.StringBuilder::Append(string)
...
IL_0080: ldloc.0
IL_0081: ldloc.s    myInt
IL_0083: callvirt   instance class
    ↪ [System.Runtime]System.Text.StringBuilder
    ↪ [System.Runtime]System.Text.StringBuilder::Append(int32)
IL_0088: pop

IL_0089: ldloc.0
IL_008a: callvirt   instance string
    ↪ [System.Runtime]System.Object::ToString()
IL_008f: stloc.1
IL_0053: pop
```

Listing 173: The IL code for the operations in the for loop in the `StringBuilder` benchmark.

In Listing 173 we can see the IL code for the `StringBuilder` benchmark.

```
IL_0038: ldstr      ""
IL_003d: stloc.0
```

```
IL_003e: ldloca.s     V_9
IL_0040: ldc.i4.0
IL_0041: ldc.i4.7
IL_0042: call         instance void
    [System.Runtime]System.Runtime.
    CompilerServices.DefaultInterpolatedStringHandler::
    .ctor(int32, int32)

IL_0047: ldloca.s     V_9
IL_0049: ldloc.1
IL_004a: call         instance void
    [System.Runtime]System.Runtime.
    CompilerServices.DefaultInterpolatedStringHandler::
    AppendFormatted(string)
IL_004f: nop
...
IL_0080: ldloca.s     V_9
IL_0082: ldloc.s      myInt
IL_0084: call         instance void
    [System.Runtime]System.Runtime.
    CompilerServices.DefaultInterpolatedStringHandler::
    AppendFormatted<int32>(!!0/*int32*/)
IL_0089: nop

IL_008a: ldloca.s     V_9
IL_008c: call         instance string
    [System.Runtime]System.Runtime.
    CompilerServices.DefaultInterpolatedStringHandler::
    ToStringAndClear()
IL_0091: stloc.0
```

Listing 174: The IL code for the operations in the for loop in the `Interpolation` benchmark.

In Listing 174 we can see the IL code for the `Interpolation` benchmark. The main difference is that a `virtcall` is used in the `StringBuilder` benchmark, however this still does not explain the differences, in fact, intuition would say that a `virtcall` is more expensive than `call` as `virtcall` would check if the method has been overridden.

This means the difference is in how a `StringBuilder` is implemented by the compiler compared to how `string` interpolation is implemented. Therefore we look into the assembly code to see if we can gain further insight.

```
// str = "";
mov         rcx,1B673373020h
mov         rcx,qword ptr [rcx]
mov         qword ptr [rbp+0C0h],rcx
// sb.Clear();
mov         rcx,qword ptr [rbp+0C8h]
cmp         dword ptr [rcx],ecx
call        CLRStub[MethodDescPrestub]@7ffa0f0d0840
 ↪  (07FFA0F0D0840h)
mov         qword ptr [rbp+68h],rax
nop
// sb.Append(iStr);
mov         rcx,qword ptr [rbp+0C8h]
mov         rdx,qword ptr [rbp+0B8h]
cmp         dword ptr [rcx],ecx
call        Method stub for:
 ↪  System.Text.StringBuilder.Append(System.String)
 ↪  (07FFA0F0D0880h)
mov         qword ptr [rbp+60h],rax
nop
// sb.Append(am);
mov         rcx,qword ptr [rbp+0C8h]
mov         rdx,qword ptr [rbp+0B0h]
cmp         dword ptr [rcx],ecx
call        Method stub for:
 ↪  System.Text.StringBuilder.Append(System.String)
 ↪  (07FFA0F0D0880h)
mov         qword ptr [rbp+58h],rax
nop
...
// sb.Append(myInt);
mov         rcx,qword ptr [rbp+0C8h]
mov         edx,dword ptr [rbp+8Ch]
cmp         dword ptr [rcx],ecx
```

```
30 call        CLRStub[MethodDescPrestub]@7ffa0f0d0908
   ↪  (07FFA0F0D0908h)
31 mov         qword ptr [rbp+30h],rax
32 nop
33 // str = sb.ToString();
34 mov         rcx,qword ptr [rbp+0C8h]
35 mov         rax,qword ptr [rbp+0C8h]
36 mov         rax,qword ptr [rax]
37 mov         rax,qword ptr [rax+40h]
38 call        qword ptr [rax+8]
39 mov         qword ptr [rbp+28h],rax
40 mov         rax,qword ptr [rbp+28h]
41 mov         qword ptr [rbp+0C0h],rax
```

Listing 175: The assembly code for the operations in the for loop in the `StringBuilder` benchmark.

In Listing 175 we can see the assembly code for the `Format` benchmark.

```
1  // str = "";
2  mov         rcx,15D18003020h
3  mov         rcx,qword ptr [rcx]
4  mov         qword ptr [rbp+98h],rcx
5  // str = $"{iStr}{am}{a}{stringStr}{with}{integer}{myInt}";
6  lea         rcx,[rbp+38h]
7  xor         edx,edx
8  mov         r8d,7
9  call        CLRStub[MethodDescPrestub]@7ffa0f0e0208
   ↪  (07FFA0F0E0208h)
10 lea         rcx,[rbp+38h]
11 mov         rdx,qword ptr [rbp+90h]
12 call        CLRStub[MethodDescPrestub]@7ffa0f0e0290
   ↪  (07FFA0F0E0290h)
13 nop
14 lea         rcx,[rbp+38h]
15 mov         rdx,qword ptr [rbp+88h]
16 call        CLRStub[MethodDescPrestub]@7ffa0f0e0290
   ↪  (07FFA0F0E0290h)
```

```
nop
...
lea          rcx,[rbp+38h]
call         CLRStub[MethodDescPrestub]@7ffa0f0e0238
 ↪  (07FFA0F0E0238h)
mov          qword ptr [rbp+20h],rax
mov          rax,qword ptr [rbp+20h]
mov          qword ptr [rbp+98h],rax
```

Listing 176: The assembly code for the operations in the for loop in the `Interpolation` benchmark.

In Listing 176 we can see the assembly code for the `Interpolation` benchmark. Here we can see that there are differences in the benchmarks, like we could see in the IL code. We can see that different built-in methods are called for both of these benchmarks which is likely to be the reason between the performance differences. To get further insight into this, we need insight into the C# compiler, which we leave for future work.

A.4.5 Invocation

Local Function vs Local Function Invocation

The first step for examining why the correlation between Elapsed Time and Package Energy does not hold for `LocalFunctionInvocation` but it holds for all the other local function benchmarks is checking the benchmarks. In this case we look at all four relevant benchmarks, `LocalFunction`, `LocalFunctionInvocation`, `LocalStaticFunction` and `LocalStaticFunctionInvocation`.

```
public class LocalFunctionBenchmarks {
    ...
    [Benchmark("InvocationLocalFunction", "Tests invocation
 ↪  using an local function")]
    public static int LocalFunction() {
        int Calc() {
            InvocationHelper.StaticField++;
            return InvocationHelper.StaticField + 2;
        }
```

```
9           int result = 0;
10          for (int i = 0; i < LoopIterations; i++) {
11              result += Calc() + i;
12          }
13          return result;
14      }
15      ...
16  }
```

Listing 177: The `LocalFunction` method which tests a call to a local function in the `LocalFunctionBenchmarks` class.

```
1   public class LocalFunctionBenchmarks {
2       ...
3       [Benchmark("InvocationLocalFunction", "Tests invocation
    ↪   using an local function")]
4       public static int LocalFunctionInvocation() {
5           int Calc() {
6               return InstanceObject.Calculate();
7           }
8           int result = 0;
9           for (int i = 0; i < LoopIterations; i++) {
10              result += Calc() + i;
11          }
12          return result;
13      }
14      ...
15  }
```

Listing 178: The `LocalFunctionInvocation` method which tests a call to a local function that calls a method in the `LocalFunctionBenchmarks` class.

```
1   public class LocalFunctionBenchmarks {
2       ...
```

```
    [Benchmark("InvocationLocalFunction", "Tests invocation
↪   using a static local function")]
    public static int LocalStaticFunction() {
        static int Calc() {
            InvocationHelper.StaticField++;
            return InvocationHelper.StaticField + 2;
        }
        int result = 0;
        for (int i = 0; i < LoopIterations; i++) {
            result += Calc() + i;
        }
        return result;
    }
    ...
}
```

Listing 179: The `LocalStaticFunction` method which tests a call to a local static function in the `LocalFunctionBenchmarks` class.

```
public class LocalFunctionBenchmarks {
    ...
    [Benchmark("InvocationLocalFunction", "Tests invocation
↪   using a static local function")]
    public static int LocalStaticFunctionInvocation() {
        static int Calc() {
            return InvocationHelper.CalculateStatic();
        }
        int result = 0;
        for (int i = 0; i < LoopIterations; i++) {
            result += Calc() + i;
        }
        return result;
    }
    ...
}
```

Listing 180: The `LocalStaticFunctionInvocation` method which tests a call to a local static function that calls a static method in the `LocalFunctionBenchmarks` class.

In Listing 177, Listing 178, Listing 179, and Listing 180 we can see that the only differences in the benchmarks are that the `static` variants uses `static` methods, and that the `invocation` benchmarks uses a function that calls a method. We have been unable to find information in relevant documentation [124] that would explain these results, therefore we look at the IL code.

```
1  // int Calc()
2  IL_0001: ldsfld       int32
   ↪  Benchmarks.HelperObjects.InvocationHelper::StaticField
3  IL_0006: ldc.i4.1
4  IL_0007: add
5  IL_0008: stsfld       int32
   ↪  Benchmarks.HelperObjects.InvocationHelper::StaticField
6
7  IL_000d: ldsfld       int32
   ↪  Benchmarks.HelperObjects.InvocationHelper::StaticField
8  IL_0012: ldc.i4.2
9  IL_0013: add
10 IL_0014: stloc.0
11 IL_0015: br.s         IL_0017
12
13 // Inside for loop
14 IL_0009: ldloc.0
15 IL_000a: call         int32
   ↪  Benchmarks.Invocation.LocalFunctionBenchmarks::'<LocalFunction>g__Calc|3_0'()
16 IL_000f: ldloc.1
17 IL_0010: add
18 IL_0011: add
19 IL_0012: stloc.0
```

Listing 181: The IL code for the operations in the for loop and in the local function in the `LocalFunction` benchmark.

In Listing 181 we can see the relevant IL code for the `LocalFunction` benchmark.

```
// static int Calc()
IL_0001: ldsfld       class
    ↪ Benchmarks.HelperObjects.InvocationHelper
    ↪ Benchmarks.Invocation.LocalFunctionBenchmarks::InstanceObject
IL_0006: callvirt     instance int32
    ↪ Benchmarks.HelperObjects.InvocationHelper::Calculate()
IL_000b: stloc.0
IL_000c: br.s         IL_000e

// Inside for loop
IL_0009: ldloc.0
IL_000a: call         int32
    ↪ Benchmarks.Invocation.LocalFunctionBenchmarks::'<LocalFunctionInvocation>g__Calc
IL_000f: ldloc.1
IL_0010: add
IL_0011: add
IL_0012: stloc.0

// Method called
IL_0001: ldarg.0
IL_0002: ldarg.0
IL_0003: ldfld        int32
    ↪ Benchmarks.HelperObjects.InvocationHelper::Field
IL_0008: ldc.i4.1
IL_0009: add
IL_000a: stfld        int32
    ↪ Benchmarks.HelperObjects.InvocationHelper::Field

IL_000f: ldarg.0
IL_0010: ldfld        int32
    ↪ Benchmarks.HelperObjects.InvocationHelper::Field
IL_0015: ldc.i4.2
IL_0016: add
IL_0017: stloc.0
IL_0018: br.s         IL_001a
```

Listing 182: The IL code for the operations in the for loop, in the local function and in the method that is called in the `LocalFunctionInvocation` benchmark.

In Listing 182 we can see the relevant IL code for the `LocalFunctionInvocation` benchmark.

```
// static int Calc()
IL_0001: ldsfld       int32
    ↪ Benchmarks.HelperObjects.InvocationHelper::StaticField
IL_0006: ldc.i4.1
IL_0007: add
IL_0008: stsfld       int32
    ↪ Benchmarks.HelperObjects.InvocationHelper::StaticField

IL_000d: ldsfld       int32
    ↪ Benchmarks.HelperObjects.InvocationHelper::StaticField
IL_0012: ldc.i4.2
IL_0013: add
IL_0014: stloc.0
IL_0015: br.s         IL_0017

// Inside for loop
IL_0009: ldloc.0
IL_000a: call         int32
    ↪ Benchmarks.Invocation.LocalFunctionBenchmarks::'<LocalStaticFunction>g__Calc|4_0
IL_000f: ldloc.1
IL_0010: add
IL_0011: add
IL_0012: stloc.0
```

Listing 183: The IL code for the operations in the for loop and in the local function in the `LocalStaticFunction` benchmark.

In Listing 183 we can see the relevant IL code for the `LocalStaticFunction` benchmark.

```
// static int Calc()
IL_0001: call         int32
    ↪ Benchmarks.HelperObjects.InvocationHelper::CalculateStatic()
```

```
IL_0006: stloc.0
IL_0007: br.s          IL_0009

// Inside for loop
IL_0009: ldloc.0
IL_000a: call          int32
    ↪ Benchmarks.Invocation.LocalFunctionBenchmarks::'<LocalStaticFunctionInvocation>g
IL_000f: ldloc.1
IL_0010: add
IL_0011: add
IL_0012: stloc.0

// Method called
IL_0001: ldsfld        int32
    ↪ Benchmarks.HelperObjects.InvocationHelper::StaticField
IL_0006: ldc.i4.1
IL_0007: add
IL_0008: stsfld        int32
    ↪ Benchmarks.HelperObjects.InvocationHelper::StaticField

IL_000d: ldsfld        int32
    ↪ Benchmarks.HelperObjects.InvocationHelper::StaticField
IL_0012: ldc.i4.2
IL_0013: add
IL_0014: stloc.0
IL_0015: br.s          IL_0017
```

Listing 184: The IL code for the operations in the for loop, in the local function and in the method that is called in the `LocalStaticFunctionInvocation` benchmark.

In Listing 184 we can see the relevant IL code for the `LocalStaticFunctionInvocation` benchmark. We can see that the main difference is that a method call is done in the invocation benchmarks, while the fields are changed directly in the other benchmarks. The reason why the invocation benchmarks have better performance is because they are accessing fields in their own class instead of in a different class. We are unable to see a reason in the IL code for why the correlation does not hold for `LocalFunctionInvocation`, while it does for the other benchmarks. As such we examine the assembly code.

```
// int Calc()
// InvocationHelper.StaticField++;
mov         rcx,7FFA4F196D18h
mov         edx,5
call        CORINFO_HELP_GETSHARED_NONGCSTATIC_BASE
    ↪ (07FFAAEC6AB90h)
mov         rcx,7FFA4F196D18h
mov         edx,5
call        CORINFO_HELP_GETSHARED_NONGCSTATIC_BASE
    ↪ (07FFAAEC6AB90h)
inc         dword ptr [7FFA4F196D54h]
// return InvocationHelper.StaticField + 2;
mov         rcx,7FFA4F196D18h
mov         edx,5
call        CORINFO_HELP_GETSHARED_NONGCSTATIC_BASE
    ↪ (07FFAAEC6AB90h)
mov         eax,dword ptr [7FFA4F196D54h]
add         eax,2
mov         dword ptr [rbp+2Ch],eax
nop
jmp
    ↪ LocalFunctionBenchmarks.<LocalFunction>g__Calc|2_0()+06Fh
    ↪ (07FFA4F0F182Fh)

// Inside for loop
// result += Calc() + i;
mov         eax,dword ptr [rbp+3Ch]
mov         dword ptr [rbp+2Ch],eax
call        Method stub for:
    ↪ LocalFunctionBenchmarks.<LocalFunction>g__Calc|2_0()
    ↪ (07FFA4F0EFD40h)
mov         dword ptr [rbp+28h],eax
mov         eax,dword ptr [rbp+2Ch]
add         eax,dword ptr [rbp+28h]
add         eax,dword ptr [rbp+38h]
mov         dword ptr [rbp+3Ch],eax
```

Listing 185: The assembly code for the relevant code in the `LocalFunction` benchmark.

In Listing 185 we see the relevant assembly code in the `LocalFunction` benchmark.

```
// int Calc()
// return InstanceObject.Calculate();
mov         rcx,277F0AC2E90h
mov         rcx,qword ptr [rcx]
cmp         dword ptr [rcx],ecx
call        CLRStub[MethodDescPrestub]@7ffa4f0f0530
 ↪  (07FFA4F0F0530h)
mov         dword ptr [rbp+28h],eax
mov         eax,dword ptr [rbp+28h]
mov         dword ptr [rbp+2Ch],eax
nop
jmp
 ↪  LocalFunctionBenchmarks.<LocalFunctionInvocation>g__Calc|4_0()+03Eh
 ↪  (07FFA4F2A50AEh)

// Inside for loop
// result += Calc() + i;
mov         eax,dword ptr [rbp+3Ch]
mov         dword ptr [rbp+2Ch],eax
call        CLRStub[MethodDescPrestub]@7ffa4f0efd50
 ↪  (07FFA4F0EFD50h)
mov         dword ptr [rbp+28h],eax
mov         eax,dword ptr [rbp+2Ch]
add         eax,dword ptr [rbp+28h]
add         eax,dword ptr [rbp+38h]
mov         dword ptr [rbp+3Ch],eax

// Method called
// Field++;
mov         rax,qword ptr [rbp+50h]
inc         dword ptr [rax+8]
// return Field + 2;
mov         rax,qword ptr [rbp+50h]
mov         eax,dword ptr [rax+8]
```

```
31 add         eax,2
32 mov         dword ptr [rbp+2Ch],eax
33 nop
34 jmp         InvocationHelper.Calculate()+039h (07FFA4F2A5189h)
```

Listing 186: The assembly code for the relevant code in the `LocalFunctionInvocation` benchmark.

In Listing 186 we see the relevant assembly code in the `LocalFunctionInvocation` benchmark.

```
1  // static int Calc()
2  // InvocationHelper.StaticField++;
3  inc         dword ptr [7FFA4F196D54h]
4  // return InvocationHelper.StaticField + 2;
5  mov         eax,dword ptr [7FFA4F196D54h]
6  add         eax,2
7  mov         dword ptr [rbp+2Ch],eax
8  nop
9  jmp
   ↪ LocalFunctionBenchmarks.<LocalStaticFunction>g__Calc|3_0()+033h
   ↪ (07FFA4F2A1573h)
10
11 // Inside for loop
12 // result += Calc() + i;
13 mov         eax,dword ptr [rbp+3Ch]
14 mov         dword ptr [rbp+2Ch],eax
15 call        Method stub for:
   ↪ LocalFunctionBenchmarks.<LocalStaticFunction>g__Calc|3_0()
   ↪ (07FFA4F0EFD48h)
16 mov         dword ptr [rbp+28h],eax
17 mov         eax,dword ptr [rbp+2Ch]
18 add         eax,dword ptr [rbp+28h]
19 add         eax,dword ptr [rbp+38h]
20 mov         dword ptr [rbp+3Ch],eax
```

Listing 187: The assembly code for the relevant code in the `LocalStaticFunction` benchmark.

In Listing 187 we see the relevant assembly code in the `LocalStaticFunction` benchmark.

```
1  // static int Calc()
2  // return InvocationHelper.CalculateStatic();
3  call        CLRStub[MethodDescPrestub]@7ffa4f0f0540
   ↪  (07FFA4F0F0540h)
4  mov         dword ptr [rbp+28h],eax
5  mov         eax,dword ptr [rbp+28h]
6  mov         dword ptr [rbp+2Ch],eax
7  nop
8  jmp
   ↪  LocalFunctionBenchmarks.<LocalStaticFunctionInvocation>g__Calc|5_0()+02Fh
   ↪  (07FFA4F2ABCCFh)
9
10 // Inside for loop
11 // result += Calc() + i;
12 mov         eax,dword ptr [rbp+3Ch]
13 mov         dword ptr [rbp+2Ch],eax
14 call        CLRStub[MethodDescPrestub]@7ffa4f0efd58
   ↪  (07FFA4F0EFD58h)
15 mov         dword ptr [rbp+28h],eax
16 mov         eax,dword ptr [rbp+2Ch]
17 add         eax,dword ptr [rbp+28h]
18 add         eax,dword ptr [rbp+38h]
19 mov         dword ptr [rbp+3Ch],eax
20
21 // Method called
22 // StaticField++;
23 inc         dword ptr [7FFA4F196D54h]
24 // return StaticField + 2;
25 mov         eax,dword ptr [7FFA4F196D54h]
26 add         eax,2
27 mov         dword ptr [rbp+2Ch],eax
28 nop
```

```
29 jmp          InvocationHelper.CalculateStatic()+033h
   ↪ (07FFA4F2ABE13h)
```

Listing 188: The assembly code for the relevant code in the `LocalStaticFunctionInvocation` benchmark.

In Listing 188 we see the relevant assembly code in the `LocalStaticFunctionInvocation` benchmark. The assembly code gives the same picture as the IL code, therefore we are unable to find the reason for why the correlation does not hold and we therefore leave that for future work.

Reflection

The first step for looking at why delegate using reflection is more efficient compared to other types of reflection, is to examine the benchmarks for `ReflectionDelegate` and `ReflectionFlags`.

```
1  public class ReflectionBenchmarks {
2      ...
3      private static readonly MethodInfo MethodReflectionDelegate
       ↪  =
       ↪  typeof(InvocationHelper).GetMethod(nameof(InvocationHelper.Calculate),
       ↪  BindingFlags.Public | BindingFlags.Instance |
       ↪  BindingFlags.InvokeMethod);
4      ...
5      private static readonly Func<InvocationHelper, int>
       ↪  ReflectionDelegateInt = (Func<InvocationHelper,
       ↪  int>)Delegate.CreateDelegate(typeof(Func<InvocationHelper,
       ↪  int>), MethodReflectionDelegate);
6      ...
7      [Benchmark("InvocationReflection", "Tests invocation using a
   ↪  reflection on an instance method using delegate")]
8      public static int ReflectionDelegate() {
9          int result = 0;
10         for (int i = 0; i < LoopIterations; i++) {
11             result += ReflectionDelegateInt(InstanceObject) + i;
12         }
13         return result;
```

```
14      }
15      ...
16  }
```

Listing 189: The `ReflectionDelegate` method which tests a call using reflection with delegates.

```
1   public class ReflectionBenchmarks {
2       ...
3       private static readonly MethodInfo MethodReflectionFlags =
        ↪  typeof(InvocationHelper).GetMethod(nameof(InvocationHelper.Calculate),
        ↪  BindingFlags.Public | BindingFlags.Instance |
        ↪  BindingFlags.InvokeMethod);
4       ...
5       [Benchmark("InvocationReflection", "Tests invocation using a
    ↪   reflection on an instance method")]
6       public static int ReflectionFlags() {
7           int result = 0;
8           for (int i = 0; i < LoopIterations; i++) {
9               result +=
                ↪  (int)MethodReflectionFlags.Invoke(InstanceObject,
                ↪  Array.Empty<object>())! + i;
10          }
11          return result;
12      }
13      ...
14  }
```

Listing 190: The `ReflectionFlags` method which tests a call using reflection with flags.

In Listing 189 and Listing 190 we can see that the only difference is that Listing 189 uses a `Delegate` while Listing 190 uses reflection with flags. According to [68] "Delegates are similar to C++ function pointers, but delegates are fully object-oriented, and unlike C++ pointers to member functions, delegates encapsulate both an object instance and a method.", which implies that pointers are used behind the scenes, which would make this more efficient than the alternatives. Because of this, it is possible that the

reflection with deletages only needs to find the method once instead of every time the method is called. To get further insight, we examine the IL code for these benchmarks.

```
// MethodReflectionDelegate
IL_0078: ldtoken      Benchmarks.HelperObjects.InvocationHelper
IL_007d: call         class [System.Runtime]System.Type
  ↪  [System.Runtime]System.Type::GetTypeFromHandle(valuetype
  ↪  [System.Runtime]System.RuntimeTypeHandle)
IL_0082: ldstr        "Calculate"
IL_0087: ldc.i4       276 // 0x00000114
IL_008c: call         instance class
  ↪  [System.Runtime]System.Reflection.MethodInfo
  ↪  [System.Runtime]System.Type::GetMethod(string, valuetype
  ↪  [System.Runtime]System.Reflection.BindingFlags)
IL_0091: stsfld       class
  ↪  [System.Runtime]System.Reflection.MethodInfoBenchmarks.Invocation.ReflectionBenc

// ReflectionDelegateInt
IL_00b4: ldtoken      class [System.Runtime]System.Func`2<class
  ↪  Benchmarks.HelperObjects.InvocationHelper, int32>
IL_00b9: call         class [System.Runtime]System.Type
  ↪  [System.Runtime]System.Type::GetTypeFromHandle(valuetype
  ↪  [System.Runtime]System.RuntimeTypeHandle)
IL_00be: ldsfld       class
  ↪  [System.Runtime]System.Reflection.MethodInfo
  ↪  Benchmarks.Invocation.ReflectionBenchmarks::MethodReflectionDelegate
IL_00c3: call         class [System.Runtime]System.Delegate
  ↪  [System.Runtime]System.Delegate::CreateDelegate(class
  ↪  [System.Runtime]System.Type, class
  ↪  [System.Runtime]System.Reflection.MethodInfo)
IL_00c8: castclass    class [System.Runtime]System.Func`2<class
  ↪  Benchmarks.HelperObjects.InvocationHelper, int32>
IL_00cd: stsfld       class [System.Runtime]System.Func`2<class
  ↪  Benchmarks.HelperObjects.InvocationHelper,
  ↪  int32>Benchmarks.Invocation.ReflectionBenchmarks::ReflectionDelegateInt

// Inside the for loop
IL_0008: ldloc.0
```

```
IL_0009: ldsfld       class [System.Runtime]System.Func`2<class
    ↪ Benchmarks.HelperObjects.InvocationHelper, int32>
    ↪ Benchmarks.Invocation.ReflectionBenchmarks::ReflectionDelegateInt
IL_000e: ldsfld       class
    ↪ Benchmarks.HelperObjects.InvocationHelper
    ↪ Benchmarks.Invocation.ReflectionBenchmarks::InstanceObject
IL_0013: callvirt     instance !1/*int32*/ class
    ↪ [System.Runtime]System.Func`2<class
    ↪ Benchmarks.HelperObjects.InvocationHelper,
    ↪ int32>::Invoke(!0/*class
    ↪ Benchmarks.HelperObjects.InvocationHelper*/)
IL_0018: ldloc.1
IL_0019: add
IL_001a: add
IL_001b: stloc.0
```

Listing 191: The IL code for the relevant operations in the `ReflectionDelegate` benchmark.

In Listing 191 we can see the relevant IL code for the `ReflectionDelegate` benchmark.

```
// MethodReflectionFlags
IL_003c: ldtoken      Benchmarks.HelperObjects.InvocationHelper
IL_0041: call         class [System.Runtime]System.Type
    ↪ [System.Runtime]System.Type::GetTypeFromHandle(valuetype
    ↪ [System.Runtime]System.RuntimeTypeHandle)
IL_0046: ldstr        "Calculate"
IL_004b: ldc.i4       276
IL_0050: call         instance class
    ↪ [System.Runtime]System.Reflection.MethodInfo
    ↪ [System.Runtime]System.Type::GetMethod(string, valuetype
    ↪ [System.Runtime]System.Reflection.BindingFlags)
IL_0055: stsfld       class
    ↪ [System.Runtime]System.Reflection.MethodInfo
    ↪ Benchmarks.Invocation.ReflectionBenchmarks::MethodReflectionFlags

// Inside the for loop
```

```
IL_0008: ldloc.0
IL_0009: ldsfld       class
    ↪  [System.Runtime]System.Reflection.MethodInfo
    ↪  Benchmarks.Invocation.ReflectionBenchmarks::MethodReflectionFlags
IL_000e: ldsfld       class
    ↪  Benchmarks.HelperObjects.InvocationHelper
    ↪  Benchmarks.Invocation.ReflectionBenchmarks::InstanceObject
IL_0013: call         !!0/*object*/[]
    ↪  [System.Runtime]System.Array::Empty<object>()
IL_0018: callvirt     instance object
    ↪  [System.Runtime]System.Reflection.MethodBase::Invoke(object,
    ↪  object[])
IL_001d: unbox.any    [System.Runtime]System.Int32
IL_0022: ldloc.1
IL_0023: add
IL_0024: add
IL_0025: stloc.0
```

Listing 192: The IL code for the relevant operations in the `ReflectionFlags` benchmark.

In Listing 192 we can see the relevant IL code for the `ReflectionFlags` benchmark. From this we observe that the main differences are in the built-in methods in the compiler, therefore this does not help gain further insight into why the difference exists between these two benchmarks. We examine the assembly code to see if we can get further insight.

```
// result += ReflectionDelegateInt(InstanceObject) + i;
mov         ecx,dword ptr [rbp+3Ch]
mov         dword ptr [rbp+2Ch],ecx
mov         rcx,143AFE12EA8h
mov         rcx,qword ptr [rcx]
mov         qword ptr [rbp+20h],rcx
mov         rcx,qword ptr [rbp+20h]
mov         rcx,qword ptr [rcx+8]
mov         rdx,143AFE12E90h
mov         rdx,qword ptr [rdx]
mov         rax,qword ptr [rbp+20h]
call        qword ptr [rax+18h]
```

```
13 mov         dword ptr [rbp+28h],eax
14 mov         eax,dword ptr [rbp+2Ch]
15 add         eax,dword ptr [rbp+28h]
16 add         eax,dword ptr [rbp+38h]
17 mov         dword ptr [rbp+3Ch],eax
```

Listing 193: The assembly code for the relevant code in the `ReflectionDelegate` benchmark.

In Listing 193 we see the relevant assembly code in the `ReflectionDelegate` benchmark.

```
1  // result += (int)MethodReflectionFlags.Invoke(InstanceObject,
   ↪  Array.Empty<object>())! + i;
2  mov         rcx,7FFA4F1A6D20h
3  mov         edx,5
4  call        CORINFO_HELP_GETSHARED_NONGCSTATIC_BASE
   ↪  (07FFAAEC6AB90h)
5  mov         rcx,143AFE12E98h
6  mov         rcx,qword ptr [rcx]
7  mov         qword ptr [rbp+48h],rcx
8  mov         rcx,7FFA4F1A6D20h
9  mov         edx,5
10 call        CORINFO_HELP_GETSHARED_NONGCSTATIC_BASE
   ↪  (07FFAAEC6AB90h)
11 mov         rcx,143AFE12E90h
12 mov         rcx,qword ptr [rcx]
13 mov         qword ptr [rbp+40h],rcx
14 mov         ecx,dword ptr [rbp+5Ch]
15 mov         dword ptr [rbp+3Ch],ecx
16 mov         rcx,7FFA4F278680h
17 call        CLRStub[MethodDescPrestub]@7ffa4f1000d8
   ↪  (07FFA4F1000D8h)
18 mov         qword ptr [rbp+30h],rax
19 mov         rcx,qword ptr [rbp+48h]
20 mov         rdx,qword ptr [rbp+40h]
21 mov         r8,qword ptr [rbp+30h]
22 cmp         dword ptr [rcx],ecx
```

```
call        CLRStub[MethodDescPrestub]@7ffa4f0e6148
↪ (07FFA4F0E6148h)
mov         qword ptr [rbp+28h],rax
mov         rdx,qword ptr [rbp+28h]
mov         rcx,7FFA4F169480h
call        CLRStub[MethodDescPrestub]@7ffa4f0d7478
↪ (07FFA4F0D7478h)
mov         eax,dword ptr [rax]
add         eax,dword ptr [rbp+3Ch]
add         eax,dword ptr [rbp+58h]
mov         dword ptr [rbp+5Ch],eax
```

Listing 194: The assembly code for the relevant code in the `ReflectionFlags` benchmark.

In Listing 194 we see the relevant assembly code in the `ReflectionFlags` benchmark. Here we see that the instruction `mov` is used extensively in both benchmarks, however the `ReflectionDelegate` benchmark is significantly shorter and only has one `call` instruction compared to the five `call` instructions in the `ReflectionFlags` benchmark. Furthermore, the `call` instruction in Listing 193 is more simple compared to the five `call` instructions in Listing 194. This implies that the compiler can optimize a lot more when using `delegates` compared to using ordinary `reflection`, even with flags. Because of this, it makes sense that `ReflectionDelegate` has better performance, as fewer instructions needs to be utilized compared to `ReflectionFlags`.

A.4.6 Object Invocation

Class Method vs Class Method Static

The first step analyzing the differences in performance between a class method and a static class method is looking at the benchmarks, together with the helper class.

```
public class ClassHelper {
    public static int StaticField = 4;
    public int Field = 4;
    public int Calculate() {
        Field++;
```

```
6           return Field + 2;
7       }
8       public static int CalculateStatic() {
9           StaticField++;
10          return StaticField + 2;
11      }
12  }
```

Listing 195: The `ClassHelper` class that has the helper methods and fields needed for the benchmarks relevant.

```
1   public class ObjectsBenchmarks {
2       ...
3       [Benchmark("ObjectInvocation", "Tests invocation of a method
    ↪   on a class")]
4       public static int ClassMethod() {
5           int result = 0;
6           ClassHelper classObject = new ClassHelper();
7           for (int i = 0; i < LoopIterations; i++) {
8               result += classObject.Calculate() + i;
9           }
10          return result;
11      }
12      ...
13  }
```

Listing 196: The `ClassMethod` method which tests a call to an object method in the `ObjectsBenchmarks` class.

```
1   public class ObjectsBenchmarks {
2       ...
3       [Benchmark("ObjectInvocation", "Tests invocation of a static
    ↪   method on a class")]
4       public static int ClassMethodStatic() {
5           int result = 0;
```

```
        ClassHelper unused = new ClassHelper();
        for (int i = 0; i < LoopIterations; i++) {
            result += ClassHelper.CalculateStatic() + i;
        }
        return result;
    }
    ...
}
```

Listing 197: The `ClassMethodStatic` method which tests a call to a static object method in the `ObjectsBenchmarks` class.

In Listing 196 and Listing 197 we can see that the only difference is the method called in the `ClassHelper` class, seen in Listing 195. This is not enough to explain the difference in performance. There is no relevant documentation that explains the difference, therefore we look at the IL code.

```
// instance method called
IL_0001: ldarg.0
IL_0002: ldarg.0
IL_0003: ldfld       int32
 ↪  Benchmarks.HelperObjects.Objects.ClassHelper::Field
IL_0008: ldc.i4.1
IL_0009: add
IL_000a: stfld       int32
 ↪  Benchmarks.HelperObjects.Objects.ClassHelper::Field

IL_000f: ldarg.0
IL_0010: ldfld       int32
 ↪  Benchmarks.HelperObjects.Objects.ClassHelper::Field
IL_0015: ldc.i4.2
IL_0016: add
IL_0017: stloc.0
IL_0018: br.s        IL_001a

IL_001a: ldloc.0
IL_001b: ret

```

```
// result += classObject.Calculate() + i;
IL_000e: ldloc.0
IL_000f: ldloc.1
IL_0010: callvirt     instance int32
 ↪  Benchmarks.HelperObjects.Objects.ClassHelper::Calculate()
IL_0015: ldloc.2
IL_0016: add
IL_0017: add
IL_0018: stloc.0
```

Listing 198: The IL code for the operations in the for loop and in the instance method called in the `ClassMethod` benchmark.

In Listing 198 we can see the relevant IL code for the `ClassMethod` benchmark.

```
// static method called
IL_0001: ldsfld       int32
 ↪  Benchmarks.HelperObjects.Objects.ClassHelper::StaticField
IL_0006: ldc.i4.1
IL_0007: add
IL_0008: stsfld       int32
 ↪  Benchmarks.HelperObjects.Objects.ClassHelper::StaticField

IL_000d: ldsfld       int32
 ↪  Benchmarks.HelperObjects.Objects.ClassHelper::StaticField
IL_0012: ldc.i4.2
IL_0013: add
IL_0014: stloc.0
IL_0015: br.s         IL_0017

IL_0017: ldloc.0
IL_0018: ret

// result += classObject.Calculate() + i;
IL_000e: ldloc.0
IL_000f: call         int32
 ↪  Benchmarks.HelperObjects.Objects.ClassHelper::CalculateStatic()
```

```
IL_0014: ldloc.2
IL_0015: add
IL_0016: add
IL_0017: stloc.0
```

Listing 199: The IL code for the operations in the for loop and in the static method called in the `ClassMethodStatic` benchmark.

In Listing 199 we can see the relevant IL code for the `ClassMethodStatic` benchmark. From Listing 198 and Listing 199 we get the same picture as the benchmark, where the only difference is that one uses an instance method and the other uses a static method. Therefore, we look at the assembly code.

```
// result += classObject.Calculate() + i;
mov         ecx,dword ptr [rbp+4Ch]
mov         dword ptr [rbp+24h],ecx
mov         rcx,qword ptr [rbp+40h]
cmp         dword ptr [rcx],ecx
call        CLRStub[MethodDescPrestub]@7ffa959d0090
 ↪  (07FFA959D0090h)
mov         dword ptr [rbp+20h],eax
mov         eax,dword ptr [rbp+24h]
add         eax,dword ptr [rbp+20h]
add         eax,dword ptr [rbp+3Ch]
mov         dword ptr [rbp+4Ch],eax

// Field++;
mov         rax,qword ptr [rbp+50h]
inc         dword ptr [rax+8]

// return Field + 2;
mov         rax,qword ptr [rbp+50h]
mov         eax,dword ptr [rax+8]
add         eax,2
mov         dword ptr [rbp+2Ch],eax
nop
jmp         ClassHelper.Calculate()+039h (07FFA959D1AF9h)
```

Listing 200: The assembly code for the relevant code in the `ClassMethod` benchmark.

In Listing 200 we can see the relevant assembly code in the `ClassMethod` benchmark.

```
// result += ClassHelper.CalculateStatic() + i;
mov         eax,dword ptr [rbp+4Ch]
mov         dword ptr [rbp+24h],eax
call        CLRStub[MethodDescPrestub]@7ffa959d0098
    (07FFA959D0098h)
mov         dword ptr [rbp+20h],eax
mov         eax,dword ptr [rbp+24h]
add         eax,dword ptr [rbp+20h]
add         eax,dword ptr [rbp+3Ch]
mov         dword ptr [rbp+4Ch],eax

// StaticField++;
mov         rcx,7FFA95A76D08h
mov         edx,3
call        CORINFO_HELP_GETSHARED_NONGCSTATIC_BASE
    (07FFAF553AB90h)
mov         rcx,7FFA95A76D08h
mov         edx,3
call        CORINFO_HELP_GETSHARED_NONGCSTATIC_BASE
    (07FFAF553AB90h)
inc         dword ptr [7FFA95A76D40h]

// return StaticField + 2;
mov         rcx,7FFA95A76D08h
mov         edx,3
call        CORINFO_HELP_GETSHARED_NONGCSTATIC_BASE
    (07FFAF553AB90h)
mov         eax,dword ptr [7FFA95A76D40h]
add         eax,2
mov         dword ptr [rbp+2Ch],eax
nop
jmp         ClassHelper.CalculateStatic()+06Fh (07FFA959D206Fh)
```

Listing 201: The assembly code for the relevant code in the `ClassMethodStatic` benchmark.

In Listing 201 we can see the relevant assembly code in the `ClassMethodStatic` benchmark. When comparing the assembly code we can see that there are some differences, mainly that there are some `call` instructions to built-in methods when changing a `static` field compared to when changing an `instance` field. These differeneces could explain why there is a correlation mismatch between Elapsed Time and Package Energy, however more analysis is needed to make sure this is the case, which we leave for future work.

A.4.7 Inheritance

Inheritance Virtual

The first step analyzing why calling a virtual method on a class, like in `Inheritance Virtual`, is more efficient compared to `Inheritance` is looking at the benchmark compared with a the `Inheritance` benchmark.

```
1  public class VirtualHelper {
2      private int _field = 4;
3      public virtual int UpdateAndGetValue() {
4          _field++;
5          return _field;
6      }
7  }
8  public class InheritanceBenchmarks {
9      ...
10     [Benchmark("Inheritance", "Tests getting and updating a
   ↪   value using a virtual method")]
11     public static int InheritanceVirtual() {
12         int result = 0;
13         VirtualHelper helper = new VirtualHelper();
14         for (int i = 0; i < LoopIterations; i++) {
15             result += helper.UpdateAndGetValue();
16         }
```

```
17          return result;
18      }
19      ...
20  }
```

Listing 202: The `InheritanceVirtual` method which tests a call to an object method in the `InheritanceBenchmarks` class and the class that contains the method.

```
1   public class ClassHelper {
2       private int _field = 4;
3       public int UpdateAndGetValue() {
4           _field++;
5           return _field;
6       }
7   }
8   public class InheritanceHelper : ClassHelper {  }
9   public class InheritanceBenchmarks {
10      ...
11      [Benchmark("Inheritance", "Tests getting and updating a
    ↪   value with inheritance")]
12      public static int Inheritance() {
13          int result = 0;
14          ClassHelper helper = new InheritanceHelper();
15          for (int i = 0; i < LoopIterations; i++) {
16              result += helper.UpdateAndGetValue();
17          }
18          return result;
19      }
20      ...
21  }
```

Listing 203: The `Inheritance` method which tests a call to an object method, where the object inherits the method in `InheritanceBenchmarks` class and the classes that contain the relevant methods.

In Listing 202 and Listing 203 we can see that the differences are that the `Inheritance` benchmark calls a method on an object that inherits from

another class, while the `InheritanceVirtual` benchmark calls a method on a virtual class. This is not enough to explain the differences in performance, therefore we look at the documentation. In [125] it is stated that "When a virtual method is invoked, the run-time type of the object is checked for an overriding member." which implies that a check is done when a `virtual` method is called, this does not explain why calling a `virtual` method is more efficient compared to an inherited method, in fact it would imply the opposite, making this result unexpected. Besides this, there is not any known documentation that would explain the differences in performance, therefore we look at the IL code.

```
1  // UpdateAndGetValue()
2  IL_0001: ldarg.0
3  IL_0002: ldarg.0
4  IL_0003: ldfld        int32
   ↪  Benchmarks.HelperObjects.Inheritance.ClassHelper::_field
5  IL_0008: ldc.i4.1
6  IL_0009: add
7  IL_000a: stfld        int32
   ↪  Benchmarks.HelperObjects.Inheritance.ClassHelper::_field
8  IL_000f: ldarg.0
9  IL_0010: ldfld        int32
   ↪  Benchmarks.HelperObjects.Inheritance.ClassHelper::_field
10 IL_0015: stloc.0
11 IL_0016: br.s         IL_0018
12 IL_0018: ldloc.0
13 IL_0019: ret
14
15 // result += helper.UpdateAndGetValue();
16 IL_000e: ldloc.0
17 IL_000f: ldloc.1
18 IL_0010: callvirt     instance int32
   ↪  Benchmarks.HelperObjects.Inheritance.VirtualHelper::UpdateAndGetValue()
19 IL_0015: add
20 IL_0016: stloc.0
```

Listing 204: The IL code for the operations in the for loop and the method called in the `InheritanceVirtual` benchmark and `Inheritance` benchmark.

In Listing 204 we can see the IL code for both the `InheritanceVirtual` and the `Inheritance` benchmarks, the reason only one set of IL code is shown is because they are equivalent. This does not help us with getting an explanation for the differences in performance, therefore we look at the assembly code.

```
1  // result += helper.UpdateAndGetValue();
2  mov         ecx,dword ptr [rbp+4Ch]
3  mov         dword ptr [rbp+24h],ecx
4  mov         rcx,qword ptr [rbp+40h]
5  mov         rax,qword ptr [rbp+40h]
6  mov         rax,qword ptr [rax]
7  mov         rax,qword ptr [rax+40h]
8  call        qword ptr [rax+20h]
9  mov         dword ptr [rbp+20h],eax
10 mov         eax,dword ptr [rbp+24h]
11 add         eax,dword ptr [rbp+20h]
12 mov         dword ptr [rbp+4Ch],eax
13
14 // _field++;
15 mov         rax,qword ptr [rbp+50h]
16 inc         dword ptr [rax+8]
17
18 // return _field;
19 mov         rax,qword ptr [rbp+50h]
20 mov         eax,dword ptr [rax+8]
21 mov         dword ptr [rbp+2Ch],eax
22 nop
23 jmp         ClassHelper.UpdateAndGetValue()+036h
   ↪  (07FFA959E2236h)
```

Listing 205: The assembly code for the relevant code in the `InheritanceVirtual` benchmark.

In Listing 205 we can see the relevant assembly code in the `InheritanceVirtual` benchmark. Important to note that the `call` instruction does not call a `CLRStub...` method.

```
// result += helper.UpdateAndGetValue();
mov         ecx,dword ptr [rbp+4Ch]
mov         dword ptr [rbp+24h],ecx
mov         rcx,qword ptr [rbp+40h]
cmp         dword ptr [rcx],ecx
call        CLRStub[MethodDescPrestub]@7ffa959dfd58
 ↪  (07FFA959DFD58h)
mov         dword ptr [rbp+20h],eax
mov         eax,dword ptr [rbp+24h]
add         eax,dword ptr [rbp+20h]
mov         dword ptr [rbp+4Ch],eax

// _field++;
mov         rax,qword ptr [rbp+50h]
inc         dword ptr [rax+8]

// return _field;
mov         rax,qword ptr [rbp+50h]
mov         eax,dword ptr [rax+8]
mov         dword ptr [rbp+2Ch],eax
nop
jmp         ClassHelper.UpdateAndGetValue()+036h
 ↪  (07FFA959E2236h)
```

Listing 206: The assembly code for the relevant code in the `Inheritance` benchmark.

In Listing 206 we can see the relevant assembly code in the `Inheritance` benchmark. Here we can see that the `call` instruction calls a `CLRStub...` method, which previous experiments have shown to be less efficient than calling straight to a pointer. The specifics of how this works requires knowledge of the C# compiler, which we leave to future work, however this could be an explanation for the difference in performance. An interesting finding here is that even though the IL code is equivalent, the assembly code had differences.

No Inheritance

The first step analyzing why not using `inheritance` is less efficient than using `inheritance` is looking at the benchmarks relevant, being the `NoInheritance` benchmark and the `Inheritance` benchmark.

```
public class ClassHelper {
    private int _field = 4;
    public int UpdateAndGetValue() {
        _field++;
        return _field;
    }
}
public class InheritanceBenchmarks {
    ...
    [Benchmark("Inheritance", "Tests getting and updating a
    ↪ value with no inheritance")]
    public static int NoInheritance() {
        int result = 0;
        ClassHelper helper = new ClassHelper();
        for (int i = 0; i < LoopIterations; i++) {
            result += helper.UpdateAndGetValue();
        }
        return result;
    }
    ...
}
```

Listing 207: The `NoInheritance` method which tests a call to an object method, where the object does not inherit the method called in `InheritanceBenchmarks` class and the classes that contain the relevant methods.

```
public class ClassHelper {
    private int _field = 4;
    public int UpdateAndGetValue() {
        _field++;
        return _field;
```

```
    }
}
public class InheritanceHelper : ClassHelper {  }
public class InheritanceBenchmarks {
    ...
    [Benchmark("Inheritance", "Tests getting and updating a
    ↪ value with inheritance")]
    public static int Inheritance() {
        int result = 0;
        ClassHelper helper = new InheritanceHelper();
        for (int i = 0; i < LoopIterations; i++) {
            result += helper.UpdateAndGetValue();
        }
        return result;
    }
    ...
}
```

Listing 208: The `Inheritance` method which tests a call to an object method, where the object inherits the method in `InheritanceBenchmarks` class and the classes that contain the relevant methods.

In Listing 207 and Listing 208 we can see that the only difference is that one calls a method that is inherited from another class and the other calls the method directly. This is not enough to explain the difference, in fact it would make sense that the benchmark that calls the method directly would be more efficient, however this is not the case. There is no information in the relevant documentation [126] that explains the difference in performance, therefore we look at the IL code.

```
// result += helper.UpdateAndGetValue();
IL_000e: ldloc.0
IL_000f: ldloc.1
IL_0010: callvirt     instance int32
↪  Benchmarks.HelperObjects.Inheritance.ClassHelper::UpdateAndGetValue()
IL_0015: add
IL_0016: stloc.0

```

```
// UpdateAndGetValue()
IL_0001: ldarg.0
IL_0002: ldarg.0
IL_0003: ldfld       int32
 ↪  Benchmarks.HelperObjects.Inheritance.VirtualHelper::_field
IL_0008: ldc.i4.1
IL_0009: add
IL_000a: stfld       int32
 ↪  Benchmarks.HelperObjects.Inheritance.VirtualHelper::_field
IL_000f: ldarg.0
IL_0010: ldfld       int32
 ↪  Benchmarks.HelperObjects.Inheritance.VirtualHelper::_field
IL_0015: stloc.0
IL_0016: br.s        IL_0018
IL_0018: ldloc.0
IL_0019: ret
```

Listing 209: The IL code for the operations in the for loop and the method called in the `InheritanceVirtual` benchmark and `Inheritance` benchmark.

In Listing 209 we can see the IL code for both the `NoInheritance` and the `Inheritance` benchmarks, the reason only one set of IL code is shown is because they are equivalent. This does not help us with getting an explanation for the differences in performance, therefore we look at the assembly code.

```
// result += helper.UpdateAndGetValue();
mov         ecx,dword ptr [rbp+4Ch]
mov         dword ptr [rbp+24h],ecx
mov         rcx,qword ptr [rbp+40h]
cmp         dword ptr [rcx],ecx
call        CLRStub[MethodDescPrestub]@7ffa959d0160
 ↪  (07FFA959D0160h)
mov         dword ptr [rbp+20h],eax
mov         eax,dword ptr [rbp+24h]
add         eax,dword ptr [rbp+20h]
mov         dword ptr [rbp+4Ch],eax

```

```
12 // _field++;
13 mov         rax,qword ptr [rbp+50h]
14 inc         dword ptr [rax+8]
15
16 // return _field;
17 mov         rax,qword ptr [rbp+50h]
18 mov         eax,dword ptr [rax+8]
19 mov         dword ptr [rbp+2Ch],eax
20 nop
21 jmp         ClassHelper.UpdateAndGetValue()+036h
   ↪  (07FFA959D17F6h)
```

Listing 210: The assembly code for the relevant code in the `NoInheritance` benchmark.

In Listing 210 we can see the relevant assembly code in the `NoInheritance` benchmark. Important to note that the `call` instruction calls a `CLRStub...` method.

```
1  // result += helper.UpdateAndGetValue();
2  mov         ecx,dword ptr [rbp+4Ch]
3  mov         dword ptr [rbp+24h],ecx
4  mov         rcx,qword ptr [rbp+40h]
5  cmp         dword ptr [rcx],ecx
6  call        Method stub for: ClassHelper.UpdateAndGetValue()
   ↪  (07FFA959D0160h)
7  mov         dword ptr [rbp+20h],eax
8  mov         eax,dword ptr [rbp+24h]
9  add         eax,dword ptr [rbp+20h]
10 mov         dword ptr [rbp+4Ch],eax
11
12 // _field++;
13 mov         rax,qword ptr [rbp+50h]
14 inc         dword ptr [rax+8]
15
16 // return _field;
17 mov         rax,qword ptr [rbp+50h]
18 mov         eax,dword ptr [rax+8]
19 mov         dword ptr [rbp+2Ch],eax
```

```
20 nop
21 jmp         ClassHelper.UpdateAndGetValue()+036h
   ↪ (07FFA959D17F6h)
```

Listing 211: The assembly code for the relevant code in the `Inheritance` benchmark.

In Listing 211 we can see the relevant assembly code in the `Inheritance` benchmark. Important to note that the `call` instruction calls a `Method stub...` method. The difference between calling a `CLRStub...` and `Method stub...` is unknown as this requires knowledge of the C# compiler, which we leave for future work, however this seems to be what causes the difference in performance between these two benchmarks.

An unexpected finding here is that while the C# code and IL code for the `Inheritance` benchmark is completely equivalent in the Listing 203 and Listing 203, the assembly code has changed. The code is completely equivalent, however the code has changed depending on what other code has run. Further analysis of the `Inheritance` results are left for future work.

A.4.8 Exceptions

Try Catch vs If

The first step for looking at why there is a difference between catching an `exception` and checking if the `exception` would be thrown with an `if` statement, is checking the benchmarks.

```
1  public class ExceptionBenchmarks {
2      ...
3      [Benchmark("Exception",
4          "Tests try-catch exception thrown with catch all
   ↪  statement, throws floor(LoopIterations / 2) exceptions")]
5      public static int TryCatchAllE() {
6          int a = 1;
7          int b = 2;
8          for (int i = 0; i < LoopIterations; i++) {
9              try {
10                 a *= 2;
11                 a += 2;
12                 a %= 20;
```

```
                a /= b;
                b = 0;
            }
            catch (Exception) {
                a %= 10;
                b = 2;
            }
        }
        return a;
    }
    ...
}
```

Listing 212: The `TryCatchAllE` method which tests a `try-catch` statement where an `exception` is thrown every second run-through in the `ExceptionBenchmarks` class.

```
public class ExceptionBenchmarks {
    ...
    [Benchmark("Exception", "Tests try-catch 'equivalent' using
    ↪ an if statement to check for exception")]
    public static int TryCatchEWithIf() {
        int a = 1;
        int b = 2;
        for (int i = 0; i < LoopIterations; i++) {
            a *= 2;
            a += 2;
            a %= 20;
            if (b == 0) {
                a %= 10;
                b = 2;
            }
            else {
                a /= b;
                b = 0;
            }
        }
        return a;
```

```
    }
    ...
}
```

Listing 213: The `TryCatchEWithIf` method which tests a `try-catch` statement where an `exception` is replaced with an `if` statement in the `ExceptionBenchmarks` class.

In Listing 212 and Listing 213 we can see that the difference is that Listing 212 uses a `try-catch` statement instead of the `if` statement used in Listing 213. Therefore we look into the documentation to see how `exception` handling differs from `if` statements. In [127] it is stated that "When an exception occurs, the system searches for the nearest catch clause that can handle the exception, as determined by the run-time type of the exception." and in [128] it is stated that "When an exception is thrown, the common language runtime (CLR) looks for the catch statement that handles this exception. If the currently executing method does not contain such a catch block, the CLR looks at the method that called the current method, and so on up the call stack." meaning that when an `exception` occurs, the call stack is looked through to find a way to handle the `exception`. It makes sense that this is significantly more expensive than executing an `if` statement, therefore we can conclude that `exceptions` need to look through the call stack, which is more expensive than executing an `if` statement.

ArgumentException vs DivideByZeroException

The first step for looking at why there is a difference between throwing an `ArgumentException` and throwing a `DivideByZeroException`, is checking the benchmarks.

```
public class ExceptionBenchmarks {
    ...
    [Benchmark("Exception", "Tests try-catch exception thrown,
    ↪ throws floor(LoopIterations / 2) exception")]
    public static int TryCatchEWOMessage() {
        int a = 1;
        int b = 2;
        for (int i = 0; i < LoopIterations; i++) {
            try {
```

```
                if (b == 0) {
                    throw new ArgumentException();
                }

                b = 0;
            }
            catch (ArgumentException) {
                b = 2;
            }
        }

        return a;
    }
    ...
}
```

Listing 214: The `TryCatchEWOMessage` method which tests a `try-catch` statement where an `ArgumentException` is thrown every second run-through in the `ExceptionBenchmarks` class.

```
public class ExceptionBenchmarks {
    ...
    [Benchmark("Exception",
        "Tests try-catch exception thrown with specific catch
    ↪ statement, throws floor(LoopIterations / 2) exceptions")]
    public static int TryCatchSpecificE() {
        int a = 1;
        int b = 2;
        for (int i = 0; i < LoopIterations; i++) {
            try {
                a *= 2;
                a += 2;
                a %= 20;
                a /= b;
                b = 0;
            }
            catch (DivideByZeroException) {
```

```
17                  a %= 10;
18                  b = 2;
19              }
20          }
21
22          return a;
23      }
24      ...
25  }
```

Listing 215: The `TryCatchSpecificE` method which tests a `try-catch` statement where a `DivideByZeroException` is thrown every second run-through in the `ExceptionBenchmarks` class.

In Listing 214 and Listing 215 we can see that there are three big differences in the benchmarks. In Listing 214, an `ArgumentException` object is created, no operations are done, and an `ArgumentException` is thrown, while in Listing 215, no object is created, multiplication, addition, modulo, and division is done, and a `DivideByZeroExpception` is thrown. From this alone, we would expect Listing 215 to be less efficient than Listing 214 as we would expect a `DivideByZeroException` object to be created implicitly when dividing by zero, however Listing 214 is less efficient. Therefore, we look into the documentation for the two `exceptions` thrown. In [129] we can see that a lot of `exceptions` are derived from `ArgumentException`, which could imply that a lookup for which one is thrown is done, while in [130] we can see that no `exceptions` are derived from it. Besides this, there is no explanation in the documentation for the differences in these exceptions, therefore we look at the IL code.

```
1  .try
2  {
3  IL_000a: nop
4  IL_000b: ldloc.1
5  IL_000c: ldc.i4.0
6  IL_000d: ceq
7  IL_000f: stloc.3
8  IL_0010: ldloc.3
9  IL_0011: brfalse.s    IL_001a
```

```
IL_0013: nop
IL_0014: newobj       instance void
  ↪  [System.Runtime]System.ArgumentException::.ctor()
IL_0019: throw
IL_001a: ldc.i4.0
IL_001b: stloc.1
IL_001c: nop
IL_001d: leave.s      IL_0026
}
catch [System.Runtime]System.ArgumentException
{
IL_001f: pop
IL_0020: nop
IL_0021: ldc.i4.2
IL_0022: stloc.1
IL_0023: nop
IL_0024: leave.s      IL_0026
}
```

Listing 216: The IL code for the operations in the `for` loop in the `TryCatchEWOMessage` benchmark.

In Listing 216 we can see the IL code for the `TryCatchEWOMessage` benchmark.

```
.try
{
IL_000a: nop
IL_000b: ldloc.0
IL_000c: ldc.i4.2
IL_000d: mul
IL_000e: stloc.0
IL_000f: ldloc.0
IL_0010: ldc.i4.2
IL_0011: add
IL_0012: stloc.0
IL_0013: ldloc.0
IL_0014: ldc.i4.s     20
```

```
IL_0016: rem
IL_0017: stloc.0
IL_0018: ldloc.0
IL_0019: ldloc.1
IL_001a: div
IL_001b: stloc.0
IL_001c: ldc.i4.0
IL_001d: stloc.1
IL_001e: nop
IL_001f: leave.s      IL_002d
}
catch [System.Runtime]System.DivideByZeroException
{
IL_0021: pop
IL_0022: nop
IL_0023: ldloc.0
IL_0024: ldc.i4.s     10
IL_0026: rem
IL_0027: stloc.0
IL_0028: ldc.i4.2
IL_0029: stloc.1
IL_002a: nop
IL_002b: leave.s      IL_002d
}
```

Listing 217: The IL code for the operations in the `for` loop in the `TryCatchSpecificE` benchmark.

In Listing 217 we can see the IL code for the `TryCatchSpecificE` benchmark. From this we can see that the differences are the same as when looking at the C# code, therefore we try to gain further insight by looking at the assembly code for when `exceptions` are thrown.

```
// try
nop
// if (b == 0)
cmp         dword ptr [rbp+48h],0
sete        al
```

```
6  movzx       eax,al
7  mov         dword ptr [rbp+40h],eax
8  cmp         dword ptr [rbp+40h],0
9  je          ExceptionBenchmarks.TryCatchEWOMessage()+082h
   ↪  (07FFA507C0692h)
10 // throw new ArgumentException();
11 mov         rcx,7FFA50844B40h
12 call        CORINFO_HELP_NEWSFAST (07FFAB033A9B0h)
13 mov         qword ptr [rbp+30h],rax
14 mov         rcx,qword ptr [rbp+30h]
15 call        CLRStub[MethodDescPrestub]@7ffa5079a2c8
   ↪  (07FFA5079A2C8h)
16 mov         rcx,qword ptr [rbp+30h]
17 call        00007FFAB02CF3E0
18 // b = 0;
19 xor         eax,eax
20 mov         dword ptr [rbp+48h],eax
21 // catch (ArgumentException) - Not generated until the exception
   ↪  is thrown
22 mov         qword ptr [rbp+28h],rdx
23 // b = 2;
24 mov         dword ptr [rbp+48h],2
```

Listing 218: The assembly code for the operations in the `for` loop in the `TryCatchEWOMessage` benchmark.

In Listing 218 we can see the assembly code for the `TryCatchEWOMessage` benchmark, important to note is that the `catch` block is not generated until the `exception` is thrown.

```
1  // try
2  nop
3  // a *= 2;
4  mov         eax,dword ptr [rbp+4Ch]
5  add         eax,eax
6  mov         dword ptr [rbp+4Ch],eax
7  // a += 2;
8  mov         eax,dword ptr [rbp+4Ch]
```

```
add         eax,2
mov         dword ptr [rbp+4Ch],eax
// a %= 20;
mov         eax,dword ptr [rbp+4Ch]
mov         ecx,14h
cdq
idiv        eax,ecx
lea         eax,[rax+rax*4]
shl         eax,2
mov         edx,dword ptr [rbp+4Ch]
sub         edx,eax
mov         dword ptr [rbp+4Ch],edx
// a /= b;
mov         eax,dword ptr [rbp+4Ch]
cdq
idiv        eax,dword ptr [rbp+48h]
mov         dword ptr [rbp+4Ch],eax
// b = 0;
xor         eax,eax
mov         dword ptr [rbp+48h],eax
// catch (DivideByZeroException) - Not generated until the
↪  exception is thrown
mov         qword ptr [rbp+30h],rdx
// a %= 10;
mov         eax,dword ptr [rbp+4Ch]
mov         ecx,0Ah
cdq
idiv        eax,ecx
lea         eax,[rax+rax*4]
add         eax,eax
mov         edx,dword ptr [rbp+4Ch]
sub         edx,eax
mov         dword ptr [rbp+4Ch],edx
// b = 2;
mov         dword ptr [rbp+48h],2
```

Listing 219: The assembly code for the operations in the `for` loop in the `TryCatchSpecificE` benchmark.

In Listing 219 we can see the assembly code for the `TryCatchSpecificE` benchmark, important to note is that the `catch` block is not generated until the `exception` is thrown. Here we can see that there are some `call` instructions in the Listing 218, while no `call` instructions exist in the Listing 219. When stepping through the code, we can not continue after the `call` instruction to `CLRStub...` in the Listing 218, while in the Listing 219 we go straight from the `idiv` instruction on line 24 to the `catch` block immediately. From this we can conclude that throwing `ArgumentExceptions` manually compared to throwing `DivideByZeroExceptions` implicitly is more expensive, as something happens in the `call` instructions that impact performance. We leave further exploration into this to future work, as it requires knowledge of the C# compiler.

Try Catch with No Exception

The first step for looking at why there is a difference between having a `try-catch` block and not having a `try-catch` block, is checking the benchmarks.

```
1  public class ExceptionBenchmarks {
2      ...
3      [Benchmark("Exception", "Tests try-catch no exception
   ↪   thrown")]
4      public static int TryCatchNoE() {
5          int a = 1;
6          int b = 2;
7          for (int i = 0; i < LoopIterations; i++) {
8              try {
9                  a *= 2;
10                 a += 2;
11                 a %= 20;
12                 a /= b;
13             }
14             catch (Exception) {
15                 a %= 10;
16                 throw;
17             }
18         }
19         return a;
```

```
    }
    ...
}
```

Listing 220: The `TryCatchNoE` method which tests a `try-catch` statement where no `exception` is thrown in the `ExceptionBenchmarks` class.

```
public class ExceptionBenchmarks {
    ...
    [Benchmark("ExceptionFinally", "Tests try-finally equivalent
    ↪ no exception thrown")]
    public static int TryFinallyEquivNoE() {
        int a = 1;
        int b = 2;
        for (int i = 0; i < LoopIterations; i++) {
            a *= 2;
            a += 2;
            a %= 20;
            a /= b;
        }
        return a;
    }
    ...
}
```

Listing 221: The `TryFinallyEquivNoE` method which tests the equivalent to not having `exceptions` thrown, without a `try-catch` statement.

In Listing 220 and Listing 221 we can see that the only difference between the two benchmarks is that Listing 220 has a `try-catch` block, however this `try-catch` block never throws an exception. Therefore we look at the documentation for `try-catch` blocks. In [128] it is stated that "The try block contains the guarded code that may cause the exception. The block is executed until an exception is thrown or it is completed successfully." implying that `try` blocks have something extra that would not exist if the `try` block was not there. To figure out the differences, we look at the IL code.

```
.try
{
IL_000a: nop
IL_000b: ldloc.0
IL_000c: ldc.i4.2
IL_000d: mul
IL_000e: stloc.0
IL_000f: ldloc.0
IL_0010: ldc.i4.2
IL_0011: add
IL_0012: stloc.0
IL_0013: ldloc.0
IL_0014: ldc.i4.s     20
IL_0016: rem
IL_0017: stloc.0
IL_0018: ldloc.0
IL_0019: ldloc.1
IL_001a: div
IL_001b: stloc.0
IL_001c: nop
IL_001d: leave.s      IL_0028
}
catch [System.Runtime]System.Exception // Never reached
{
IL_001f: pop
IL_0020: nop
IL_0021: ldloc.0
IL_0022: ldc.i4.s     10
IL_0024: rem
IL_0025: stloc.0
IL_0026: rethrow
}
```

Listing 222: The IL code for the operations in the `for` loop in the `TryCatchNoE` benchmark.

In Listing 222 we can see the IL code for the `TryCatchNoE` benchmark.

```
IL_0009: nop
```

```
IL_000a: ldloc.0
IL_000b: ldc.i4.2
IL_000c: mul
IL_000d: stloc.0
IL_000e: ldloc.0
IL_000f: ldc.i4.2
IL_0010: add
IL_0011: stloc.0
IL_0012: ldloc.0
IL_0013: ldc.i4.s     20
IL_0015: rem
IL_0016: stloc.0
IL_0017: ldloc.0
IL_0018: ldloc.1
IL_0019: div
IL_001a: stloc.0
IL_001b: nop
```

Listing 223: The IL code for the operations in the `for` loop in the `TryFinallyEquivNoE` benchmark.

In Listing 223 we can see the IL code for the `TryFinallyEquivNoE` benchmark. The only differences here is that there is a `try` block around the operations, and a `leave.s` instruction at the end of the `try` block in the Listing 222. There is also a `catch` block in Listing 222, however this is never reached when running the code. To figure out more, we look at the assembly code for these benchmarks.

```
// try
nop
// a *= 2;
mov         eax,dword ptr [rbp+4Ch]
add         eax,eax
mov         dword ptr [rbp+4Ch],eax
// a += 2;
mov         eax,dword ptr [rbp+4Ch]
add         eax,2
mov         dword ptr [rbp+4Ch],eax
```

```
// a %= 20;
mov         eax,dword ptr [rbp+4Ch]
mov         ecx,14h
cdq
idiv        eax,ecx
lea         eax,[rax+rax*4]
shl         eax,2
mov         edx,dword ptr [rbp+4Ch]
sub         edx,eax
mov         dword ptr [rbp+4Ch],edx
// a /= b;
mov         eax,dword ptr [rbp+4Ch]
cdq
idiv        eax,dword ptr [rbp+48h]
mov         dword ptr [rbp+4Ch],eax
```

Listing 224: The assembly code for the operations in the `for` loop in the `TryCatchNoE` benchmark.

In Listing 222 we can see the assembly code for the `TryCatchNoE` benchmark.

```
// a *= 2;
mov         eax,dword ptr [rbp+3Ch]
add         eax,eax
mov         dword ptr [rbp+3Ch],eax
// a += 2;
mov         eax,dword ptr [rbp+3Ch]
add         eax,2
mov         dword ptr [rbp+3Ch],eax
// a %= 20;
mov         eax,dword ptr [rbp+3Ch]
mov         ecx,14h
cdq
idiv        eax,ecx
lea         eax,[rax+rax*4]
shl         eax,2
mov         edx,dword ptr [rbp+3Ch]
```

```
17 sub          edx,eax
18 mov          dword ptr [rbp+3Ch],edx
19 // a /= b;
20 mov          eax,dword ptr [rbp+3Ch]
21 cdq
22 idiv         eax,dword ptr [rbp+38h]
23 mov          dword ptr [rbp+3Ch],eax
```

Listing 225: The assembly code for the operations in the `for` loop in the `TryFinallyEquivNoE` benchmark.

In Listing 223 we can see the assembly code for the `TryFinallyEquivNoE` benchmark. Here we can see that the only difference is the `nop` instruction that exists where the `try` statement is, besides this there are 0 differences in the two benchmarks. Because of this, we can not conclude why there is a difference between these two benchmarks, and we leave further exploration of this to future work.