THE PENNSYLVANIA STATE UNIVERSITY
SCHREYER HONORS COLLEGE

DEPARTMENT OF COMPUTER SCIENCE

CHARACTERIZING AND INDUCING MEMORY LOCALITY IN OBJECT-ORIENTED PROGRAMS VIA STATIC ANALYSIS AND PROFILING

JACK KINSEY
SPRING 2019

A thesis
submitted in partial fulfillment
of the requirements
for a baccalaureate degree
in Computer Science
with honors in Computer Science

Reviewed and approved* by the following:

John Sampson
Assistant Professor of Computer Science
Thesis Supervisor

Jesse Barlow
Professor of Computer Science
Honors Adviser

*Signatures are on file in the Schreyer Honors College.

Table of Contents

List of Figures

List of Tables

Acknowledgements

My thanks go to Dr. John Sampson, who devoted countless hours to this project as my thesis advisor, and to my family, whose belief in me never wavered.

Chapter 1

Introduction

1.1 Introduction

In recent years, the increasing relative cost of data movement when compared to the speed of compute resources has warranted a reexamination of some of the ideas and architectures that emerged in the field of near-data processing (NDP) research. As might be expected, NDP focuses on moving computation as close to data as possible in order to minimize the amount the data itself must be moved [1]. Numerous challenges arise in any particular attempt to describe or implement an NDP architecture, ranging from the hardware to the system software to the programming language used to take advantage of the ability to do near-data computation, if indeed the implementation is not transparent; when considering every permutation of NDP, these problems are numerous indeed.

There is a notable gap in most high-level programming languages where NDP is concerned; [14] there are few if any popular paradigms with which the concept of executing data close to one or another piece of data can be described. We call this nearness *locality*, with this locality sharing many of the properties of the more common principle of locality but having a different nuance. Much research focuses on various NDP hardware configurations that exploit locality, but such architectures require programs that will run on them. Some designs avoid the problem by being transparent additions to a more usual architecture, but what this approach grants in usability it can potentially lose in maximal effectiveness [4].

An ideal NDP architecture would accept any code and perfectly map it onto processing resources in the best fashion possible; however, it is not necessarily clear what the best fashion would entail. The affinity of some arbitrary program for such a mapping is certainly not obvious from the outset; one could imagine programs falling into a range of suitabilities for use in any particular architecture [11]. Therefore it is desirable to be able to make statements regarding this suitability or affinity despite the common lack of explicit language-level awareness of locality.

We propose an approach for characterizing the implicit locality structure of a program

through analysis of its data types and the functions defined on them. Specifically, we refer to types that are constructed to be data encapsulated, such as classes in C++ object-oriented programming. By constructing a graph from the connections between types implied by their functions and weighting it through profiling, we are able to make statements about the affinity of a program for a particular division into several localities and the possible costs and benefits of such an arrangement.

We implement our approach with a pass designed for the LLVM compiler framework and a modified version of the K-means algorithm. While limited by the difficulty of compiling larger projects to fully linked LLVM bitcode, initial results from the implementation show promise for the characterization as a method for assessing and describing both the natural locality structure of a program and the suitability of a program for mapping into a more specific hardware setting.

The remainder of the paper is divided into several chapters. Chapter 2 provides more insight into other work in the field and a deeper understanding of locality. Chapter 3 describes the details of the characterization approach in the abstract, while Chapter 4 discusses the specifics of the implementation and presents the results from the analysis of several programs. Chapter 5 considers some avenues for future work and concludes the paper.

Chapter 2

Background

We begin with a discussion of terms and their usage, to better ground our work in the field of related research.

2.1 Near-data processing

In the field of computer architecture research, there is a clustering of ideas and patterns that is best described as the subfield of *near-data processing* (NDP) research. Architectures discussed and elaborated on in this field largely share a single commonality: They co-locate compute and memory resources far more than is traditional. The traditional draw of near-data processing is simple: Moving data takes time, and for no small number of computations the time it takes to move data to the CPU is what bottlenecks an entire program [11]. The study of NDP is just the study of different ways of decreasing the physical space between compute and data elements, in as many different forms as such a broad description can entail.

It is difficult to make concrete statements about all of NDP; the specifics of each architecture vary widely. In 1970, Stone [10] described a logic-in-memory computer, focusing on the potential performance benefits of doing simple arithmetic and logical operations on data in main memory, without having to bring data in to a central processor. Similar concepts are described by Patterson et al. [8], Draper et al. [4], and Gokhale, Holmes, and Iobst [5]. However, Yavits, Morad, and Ginosar [12] focuses on applying in-memory computation to a last-level cache replacement instead of using the technology in main memory. Working at yet another level of memory, De et al. [3] considers the possibility of an FPGA-based processing core located on the controller of an SSD.

An excellent example of the kind of distributed technology possible with NDP is the Execution Migration Machine (EM^2) described in Khan, Lis, and Devadas [6]. The key feature of the EM^2, which is based on large-scale multicore architectures, is that a specific set of data cannot exist in multiple places at once; the system will not copy data, only move

it. The primary consequence of this feature is that EM^2 migrates computation between its many cores when a piece of data not located on the currently-executing core is required, rather than copying the data into the cache of the core that is running.

Still another family of NDP architectures focuses on the applications of 3D die stacking to hardware organization. Zhang et al. [15] discusses the potential for stacking to create high-capacity, high-bandwidth in-memory processing architectures; in a similarly focused paper, Zhang et al. [14], the authors remark on the tileability of such an architecture. They also note in their concluding statements that the software ecosystem surrounding the NDP field is currently underdeveloped, describing a need for high-level abstractions that allow the programmer to express the "available locality" of an architecture.

2.2 Locality

In the classical von Neumann architecture with a memory hierarchy of caches, main memory, and disk memory, there exists a principle of locality. This is the idea driving the use of caches at all: If a given memory location has been accessed recently, it and the locations near it are likely to be accessed soon as well. While not true of every computational pattern, this empirical law of sorts holds often enough to make caches work. However, this is a different sense of locality from that used in the NDP field. For NDP, locality means the literal co-localness of data and the compute resources that have been programmed to operate on that data; in this paradigm, a locality is the literal physical nearness in space of a processor and the data it is computing with.

The nearer data is near to the processors that will compute with it, the less data movement need occur—this is the obvious premise behind the importance of locality in NDP. Therefore, instead of an observation we can make about how programs empirically tend to behave, locality is a state we must actively attempt to create, with numerous NDP architectures all presenting different methods of enforcing locality in both hardware and software to

maximize their efficiency.

However, as the principle of locality instructs, there is surely some degree to which this locality structure exists naturally in the programs we already write. Certain computations in a program access certain areas of data, rarely if ever stepping outside the walls that have been inadvertently implied by the methods we use to build our programs. To the degree that this assertion is true, we see that a certain degree of the locality we desire may already be encoded in programs that exist today. The question, then, is how to measure and make explicit this pre-existing locality so that it can be leveraged by some NDP hardware architecture.

Chapter 3

Characterization

3.1 Motivation

While there are any number of ways to attack the problem of characterizing locality, in this work we will use the concept of data types, specifically as they appear in the object-oriented programming of the C++ programming language. Type is a good heuristic due to the tendency of heavily object-oriented programs to engage in the best practice of *data encapsulation*, the intentional construction of class types to avoid sharing memory between classes of different types.

On such a class, methods are defined to provide instances of the type with behaviors. In a system of well-encapsulated types, methods will directly access only the data that is contained within an object, with all other computation occurring via method calls on other type instances, whether of the same type or others. These method calls model communication between the types involved, and if one type communicates with another, that communication could become a performance liability if the data involved for each type is being kept in two different places. Therefore we see that there may be something to take away from object-oriented programming as far as NDP scheduling is concerned.

If we know that we want to perform computation as close to our data as possible, then we must ensure that the data we are using is contained within a single area. However, a naive approach to allocating memory does not make any special consideration of this kind. Therefore we need some way to instruct an allocation algorithm to co-locate data that will be used simultaneously in the computations of our program. This can be done explicitly or implicitly, and given the preponderance of programs written in an object-oriented style and our previous observations about the nature of communication between objects and therefore between types, we see that we may be able to leverage the statically declared communication between types, as encoded by the functions on those types, to provide a heuristic for what types should be allocated near to each other and which can be allocated separately.

An example of a system that might motivate using such an approach is presented in

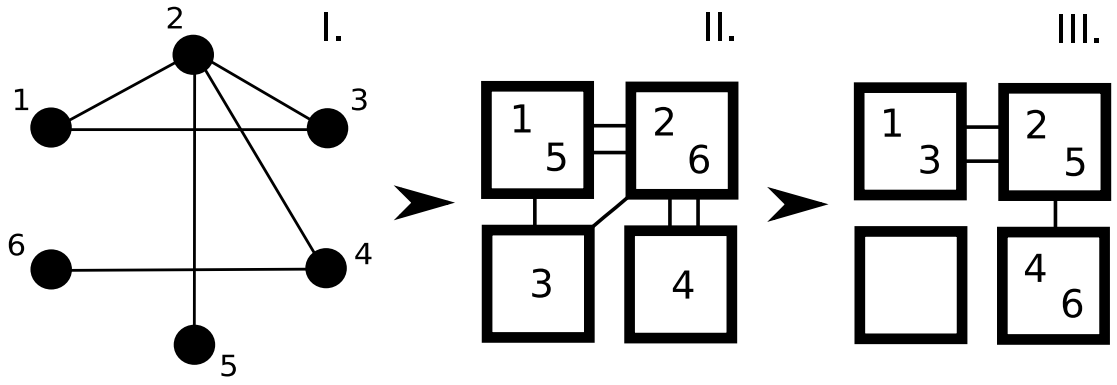

Figure 3.1: A visualization of the potential of using object-oriented structure as a heuristic for mapping data to localities.

Figure 3.1. We envision a set of types and functions between them as a graph (I). Such a system might be mapped onto a set of localities in a suboptimal way by a naive algorithm, creating more inter-locality communication than is necessary (II). We can group data more intelligently by taking advantage of the structure of the graph, reducing inter-locality communication (III).

We present a specific approach to performing a characterization of this nature, with the end goal of developing a quantification of the degree to which the possibly extant data encapsulation of a program can be exploited in an NDP context.

3.2 Approach

To begin our characterization, we describe the subset of programs that we intend to restrict ourselves to. Due to our focus on data encapsulated types, we cannot hope to apply our approach to every program; we therefore state some qualities we intend to assume exist in any program being characterized by our approach. While programs breaking with our assumptions will not render the results of the characterization incorrect, it is likely that very little of interest can be discovered about them via the method described here.

First, we expect that objects of a given type are largely used the same way by the program that defines the type. Second, we expect no one type to dominate the overall data usage

of a program. Third, types, specifically those defined by classes as they appear in C++ and other languages like it, are largely data-isolated; this is the previously referenced data encapsulation. Restricting our analysis to programs that meet these expectations provides us with some

In a program that meets our expectations, several structural consequences of those expectations provide useful hooks for the beginning of our characterization. Let us examine, in an abstract sense, such a program. We will consider a program to be first and foremost a collection of types, with each type having some number of functions defined on it. The functionality of the program, then, is encoded in these functions on types, with the programmer defining an algorithm by constructing a pattern of communication between their defined types. Functions can operate as communication between types, internal to a type, and internal to a single instance, though this last

Types themselves, aside from having functions defined on them, can composite several other types into a new unit of abstraction. However, this composition is of less interest to us than the composition implied by the definition of functions: Some function F, defined on some type T_0, and taking at least one argument, describes a communication between types. In order to perform the computation defined by F, members of the types of its arguments $(T_1, T_2, T_3, \dots)$ must, in an NDP context, be co-located in space. Therefore we find that function definitions imply constraints on the landscape of memory as seen through the lens of type.

Using types and functions as our primitive objects, we can construct an undirected graph that describes the memory structure of a program. We begin by describing the set $T(P)$ of all types in a program P. For the purposes of this mathematical analysis, we shall describe a type as a member $t \in \mathbb{N} \times P(\bar{F})$, where $\bar{F}$ is the set of all functions and $P(\bar{F})$ is then the power set of the set of all functions. The member $n \in \mathbb{N}$ is a description of the amount of memory a single instance of the type consumes. The member $f \in P(\bar{F})$ is, of course, the set

of all functions defined on the type. The set $T(P)$ can thus be described:

$$T(P) = \{t \in \mathbb{N} \times P(\bar{F}) \mid t \text{ appears in the program } P\}$$

To construct our graph G we begin by choosing

$$V(G) = T(P)$$

That is, the vertices of the graph are all the types in the program. Given types as our vertices, it follows naturally to construct the edges of our graph from the functions defined on a type. We can decompose the type signature of a function into several pairs of types and by this decomposition naturally describe the edges of our graph. A function $f \in \bar{F}$ can, for the purposes of our analysis, be described as a set of types; equivalently, we can state that

$$\bar{F} = P(\mathbb{N} \times P(\bar{F}))$$

We will not meditate overlong on the implications of this recursive definition. With a definition of functions as mathematical objects in hand, we can state the previously described decomposition rigorously:

$$D(f, t) = \{(t, t_f) \mid t_f \in f\}$$

where t will be the type on which f is defined. Describing the edges of our graph is now simple:

$$E(G) = \bigcup_{t \in V(G)} \bigcup_{f \in t} D(f, t)$$

We have described a graph that can be constructed quite naturally from any program that conforms to our assumptions. While a degree of rigor was useful in the act of constructing the graph, we will focus less attention on the specifics of the theory moving forward. In order to make use of our graph, henceforth the *type graph* of a program, we must augment

it slightly to reflect not only the static structure of the program it was constructed from but also the dynamic structure of that program as it runs on a given input. While we may be able to make some trivial deductions from the unaugmented type graph—that two types never communicate, meaning they need share no locality, or that some subset of types forms a completely inter-communicating network as described by a complete graph, implying that perhaps it is more likely that those types should be co-located—we cannot make very many strong deductions. It is entirely possible that such a complete subgraph is composed of several types that communicate with each other only once over the course of the entire program, while two other separate types might communicate several billion times. With only the statically constructed type graph at hand, it is impossible to know in general.

For this reason we will assume that after the construction of a program's type graph, the graph is weighted so that the weight of a vertex corresponds to the total amount of memory used by the type during the execution of the program, and the weight of an edge corresponds inversely to the total number of times the communication denoted by the edge occurs during the execution of the program. This is slightly distinct from a more direct measure such as "number of times a function was executed," since multiple separate functions can imply the same communication. For example, some type t_0 might have two functions defined on it, $f_1 = \{t_1, t_2\}, f_2 = \{t_1, t_3\}$. Thus the tallied executions of f_1 and f_2 should both be factored into the weight of the edge (t_0, t_1).

It is also worth noting that the weight of an edge should indeed decrease as a function of communication count: This helps model the desired structure of often inter-communicating types needing to be co-located. Additionally, it should be clarified that it is of course possibly, even likely, that a program might not have stable performance metrics across the entire domain of its input. For this reason, when we refer to statistics such as memory consumed and function execution count, we refer to the ideal metrics that best characterize the greatest subset of the domain of the program input. With these details managed, we may now proceed to the second major phase of the characterization.

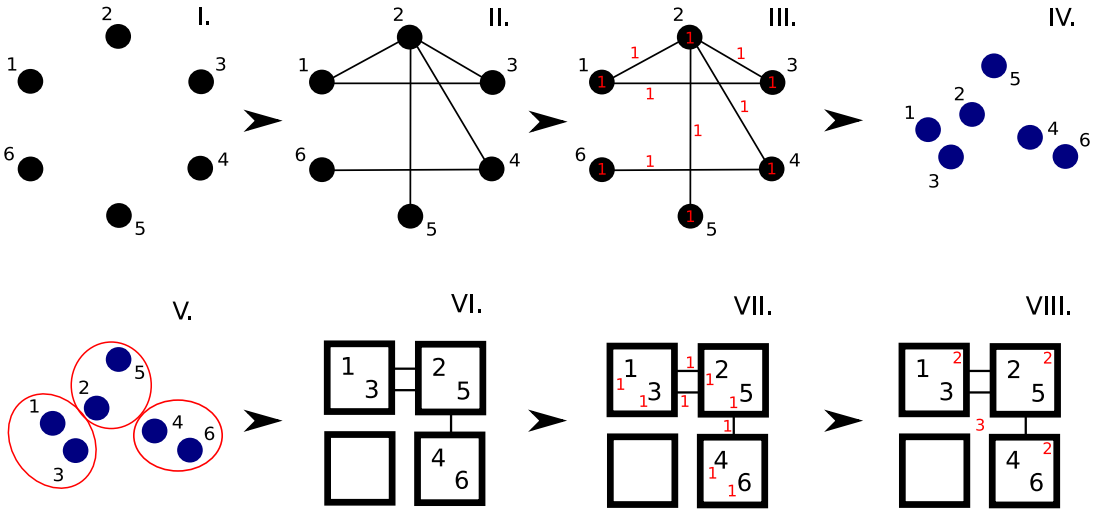

Figure 3.2: A visualization of the characterization approach for both the graph construction and K-means clustering phases. Types are found in the program (I), then functions determine the edges between them (II). The type graph is augmented (III) and converted to a point space for clustering (IV). The K-means algorithm groups points together (V) and those clusters describe localities (VI). We can bring back the weights from the graph (VII) to determine statistics about the clustering (VIII).

Having described the type graph and the results of its augmentation, if not the means by which it might be augmented, we can can make use of the structure it evidences to provide a concrete heuristic of the locality of the generating program. To do so we will present a version of the K-means algorithm tailored for the problem of recognizing—and to some degree, inducing—localities in our graph, which are equated with the clusters of the K-means algorithm. We choose K-means for its simplicity and for its natural fit to the more practical version of this problem: Scheduling code for execution in some set, architecture-specific number of localities N provides a useful constraint when considering the matter of clustering, and with the K-means method, we are able to choose $K = N$ and inspect how well a program's natural structure fits a potential NDP architecture.

We first recognize that the traditional mathematical graph has an equivalent representation as an adjacency matrix, and that the columns of this matrix constitute a set of vectors, each describing a point in some n-dimensional space and each corresponding exactly to a member of our original set of types. It is in this space that we will attempt to describe type

clusters, or equivalently, localities of type.

The usual K-means algorithm, in brief, takes as input a set of points and a desired number of clusters K, and provides as output the K clusters, which partition the original input set. The clusters are constructed in an attempt to minimize the Euclidean distance between any member of a cluster and the mean of every member of a cluster; the K-means algorithm itself is heuristic, for the clustering problem is NP-hard.

Considering the potential results of the usual K-means algorithm on our type space, it seems likely that programs meeting our expectations could demonstrate favorable clustering. Because the spatial coordinates of a type are determined by the frequency of its communication with other types, and the more frequently two types communicate the closer they are in space, it seems reasonable to expect that in programs that demonstrate structural locality in type, we might be able to identify that locality by the clusters formed in our type space.

Now that we possess a complete methodology for characterizing the structural locality of a program, we are able to provide some quantifiable metrics of locality. Perhaps the simplest such metric is the *communication overhead* C of a given clustering. C is calculated by modifying the augmented type graph of a program using the results of the K-means clustering. First, the edge weights must be recalculated to be proportional to the number of times a communication occurs, rather than inversely proportional. All edges between two vertices in the same cluster are discarded, and the remaining edges' weights are summed to become C. Formally,

$$R = \{(l, r) \in E(G) \mid l, r \text{ do not coexist in any of the } K \text{ clusters}\}$$

$$C = \sum_{e \in R} W(e)$$

This is, of course, the very metric we hope to minimize in all NDP contexts. However, we do so in an indirect manner, and an examination of our previously stated K-means method indicates that we have neglected one important piece of work. If we wished to minimize C

quite effectively, we could simply apply K-means while picking $K = 1$, and we would find that no matter the structure of the program, everything would be quite local. This is, of course, a degenerate case and is of no help to us. We must modify our K-means approach slightly to ensure that our results are maximally useful to us.

The feature we have neglected to account for is the bounded size of memory in a physical locality. If every type is grouped into a single locality, we have essentially ignored the very features we hoped to exploit in constructing an NDP architecture. We must provide some kind of back pressure to our K-means algorithm that prevents clusters from growing too large. While the disparate nature of NDP systems prevents us from drawing too much direct inspiration in how to describe this source of pressure, we can make the simplifying assumption that we wish to determine the local structure of a program given that any particular locality has a maximum memory size M that it is capable of accommodating. With this constraint, we can craft an objective function to add to our K-means algorithm:

$$B(c) = \alpha \cdot \max(c - M, 0)$$

This is a function of the size of a cluster, calculated by summing the weights of all the types contained in the cluster. So long as a cluster stays below the size M, there is essentially no penalty, but adding types that push the size above M incurs a linearly increasing cost. A function with linear growth was chosen to reflect the likely source of this cost in an actual NDP architecture: accesses back to some larger memory from the memory of the locality. The cost of these accesses is determined largely by the latency of main memory, and the time loss incurred by that latency will increase linearly as the number of accesses increases. While in this function we input only the total amount of memory that will be used, it seems reasonable to assume that the number of main memory accesses might scale roughly linearly with the size of the data overflow, at least at the high level at which we use this heuristic.

With the K-means algorithm reworked to use both this objective function and the usual

Euclidean distance to determine clusters, we see that the two will compete. For so long as a cluster is smaller than size M, its members will be determined entirely by distance from the cluster centroid, but for clusters that have grown to larger than size M, the objective function will heavily weight against the addition of any new points to the cluster. While possessing a tendency for instability, our modified K-means algorithm should nevertheless approximate clusters that approach our ideal of minimizing the communication overhead C.

It should be noted that our addition of the concept of memory limits to the characterization permits the description of another statistic that describes the affinity of a program for a certain clustering. This is the *latency overhead* L. Much as C describes the degree of inter-locality communication in a clustering, L describes the quantity of memory that overflows from localities back to main memory. We can describe L formally in terms of the objective function, where the α parameter is 1:

$$L = \sum_{c \in G} B(c)$$

L is dependent on the amount of type weight that overflows the memory limit M in each cluster in the clustering. While the back pressure function opposes the formation of degenerate clusterings, L provides a dimension in which that opposition can be quantified. With the two statistics C and L together, we can describe both how a clustering excels and how it fails.

We have accomplished our original goal of specifying a method by which the potential structural localities of a program meeting our expectations can be extracted and quantified. We will now discuss a practical implementation of the work and the results of testing with that implementation.

Chapter 4

Implementation

4.1 Methodology

In order to investigate the structure of real programs and determine whether or not they demonstrate the locality characteristics described in this work, it is necessary to perform some form of static code analysis. For this task, the LLVM compiler framework was chosen. LLVM is somewhat unique among compiler frameworks both in that it is easily extensible and in that it has only one primary internal representation (IR), the LLVM IR. This IR takes the form of a pseudo-assembly code with properties that make it workable for many kinds of analysis as well as modification and instrumentation.

While the most popular compiler in the LLVM ecosystem is by far the C/C++ compiler clang, by opting to work with LLVM we do not necessarily limit the scope of the work to only the analysis of C++ programs. LLVM frontends (compilers that output IR) exist for a decent number of programming languages, including the increasingly popular Rust language. However, in practice, the analysis code created for the sake of this work is unlikely to function in an unaltered state when used with IR generated from languages other than C++, for no other reason than a lack of testing. This could be an avenue for future work.

The primary LLVM extension, or *pass*, developed for the sake of the work is the Regraph analysis pass. This pass implements an algorithm designed to extract the type graph of a program, as discussed in Chapter 3. The Regraph pass also performs slight instrumentation of the program being analyzed to permit the collection of rudimentary profiling statistics. We will discuss the major decisions and compromises made in the process of developing Regraph.

The first adaptation required by the implementation was realistic definition of types and what it means for a type to be defined on a function. At the LLVM IR level, the class syntax of C++ no longer exists; there are simply loose functions and types. However, all functions defined on a class, when compiled to IR, take as their first parameter a pointer to the class they are defined on. It is for this reason that pointer types were adapted as the stand-in for

the abstract notion of type previous referenced. By taking interest in all functions that take a pointer type as their first parameter and further take at least one other pointer type as another parameter, we include every function that is defined on a class and takes parameters of a class type. However, this definition is not necessarily exclusive of functions that, at the C++ level, do not have these semantics.

Considering such a function for a moment, however, we see that it might have properties similar to those we already have a stated interest in. While less readily modeling a message-passing framework, a function taking several pointers as arguments nevertheless establishes a computational context that is highly likely, if not guaranteed, to make memory accesses to both types it is provided with pointers to. This premise implies that it is still very reasonable to include such a function in our analysis—the function still "connects" two types—even if it does not have the exact semantics we originally specified.

While the remaining details of the static analysis do not warrant description, the matter of weighting the graph using runtime profiling is of some interest. While edge weights are relatively trivial to calculate—the program being processed is simply instrumented to count every time one of the functions of interest is called—vertex weights are another matter.

First is the problem of determining a type's unit weight. While calculating the number of bytes filled by the members of a struct type, what a C++ class becomes at the IR level, is simple, the subject of types with pointer type members is worthy of note. Such a type is capable of having an arbitrary amount of memory assigned to a single instance and as such, it could be argued, cannot have a unit weight at all. We choose to make the assumption that types such as these will very often be collection types, their arbitrary memory regions filled with instances of some other type. Therefore it makes little sense to count this memory as belonging to the collection type, and for as far as our assumption holds true, we see that even those types with pointer members can have their unit weights readily determined.

The matter of instance counting, however, is less easily resolved. There are many ways to allocate memory to an instance of a type—first and foremost is the choice of heap or stack

allocation. Stack-allocated variables were ignored for the purposes of this implementation, since the memory restrictions of the stack mean most of the memory usage of a program cannot be stack-related. The tracking of heap allocations can be approached in a number of ways, but the final method selected counts every use of the C++ `new` operator, which is translated into a function call in the LLVM IR. While a good deal of the memory usage of a program can still escape notice due to the complexity of the instance counting problem, this method is still useful; it will never overestimate type weights, making it an admissible heuristic, and testing showed that counted weights were usually proportional to the total memory usage of a program.

Having established the implementation of both the type graph and the augmented type graph, we can move on to examining the choices made in implementing the custom K-means algorithm, which was given the name Aslo (for "assign localities"). Key decisions concern the treatment of edge and vertex weights during the process of analysis, as opposed to the original graph augmentation. The required inverse relationship between edge weight and distance between points was implemented with a multiplicative inverse; edge distances therefore range between 0 exclusive and 1 inclusive. To describe the distance between two points that do not share an edge, an arbitrary large number was selected in lieu of making any use of infinities. As for vertex weights, the count of instances multiplied by the unit weight of the type suffices for use in the K-means analysis.

For the purposes of testing the implementation, primarily the Parsec benchmark suite was used. Parsec is mostly written in C++ and represents a set of common workloads and workload types [2]. While not every benchmark is written in the object-oriented style required for the Regraph pass to function optimally, this will be considered as a factor in assessing the usability and functionality of the implementation.

4.2 Results

In total, eight programs were analyzed via the REGRAPH pass and ASLO clustering algorithm. Six[1] of these were PARSEC benchmarks (`blackscholes`, `canneal`, `fluidanimate`, `freqmine`, `streamcluster`, and `swaptions`), and two were nearest-neighbor search algorithms with implementations from Li et al. [7] (`AGH`, `NSH`). Several statistics were collected for each program. First and foremost were the C and L values, the communication overhead and latency overhead, described in Section 3.2. In presenting these values, we make a habit of normalizing them to a more comprehensible form. C is always normalized by the total edge weight of the type graph, and L is always normalized by the total vertex weight. The normalized values, which we henceforth will refer to without qualification, are therefore 1 in the worst case—when all communication between types is between separate localities, and when all of the memory accesses of a program will have to make reference to main memory—and 0 in the best case, when no communication occurs between localities and all memory fits into the localities described. We also make common reference to K, the number of localities, α, the linear scaling factor in the cluster objective function, and M, the memory limit specified for a set of localities (in bytes).

Also worthy of discussion is the Rand index. First discussed in Rand [9], we use the Rand index here as a measure of stability for the ASLO K-means algorithm output. Whenever a K-means was run, 9 additional K-means runs were made with the same constants, and the mean of the Rand indices of each compared to the initial was found. The Rand index is a description of how many pairings of points two clusterings agree on. A set of three points $\{a, b, c\}$ has three possible pairings of points: $\{(a, b), (b, c), (a, c)\}$. If such a set was clustered into two clusters $\{\{a, b\}, \{c\}\}$ first and then two clusters $\{\{a\}, \{b, c\}\}$ second, we would note that the two clusterings agree on a single pairing of points—namely, the pairing

[1] These PARSEC benchmarks were those that were both written in C++ and able to be compiled to LLVM IR for processing by the REGRAPH pass. All PARSEC data was collected using the `sim-large` input set, to avoid overwhelming the memory limit parameter.

Program	K	Rand index	Number of types	C	L index
`blackscholes`	—	—	—	—	—
`canneal`	2	0.654	40	0.0438	0.547
`fluidanimate`	2	0.771	9	0.0	1.0
`freqmine`	2	0.720	10	4.86e-08	0.956
`streamcluster`	2	0.745	11	3.35e-08	1.0
`swaptions`	2	0.407	4	5.62e-05	1.0
`AGH`	2	0.965	155	0.182	1.0
`NSH`	3	0.918	140	0.150	0.992

Table 4.1: Latent locality structure of the analyzed programs.

(a, c), which they both agree does not exist in any cluster. Therefore the Rand index of this pair of clusterings would be $\frac{1}{3}$. As the Rand index approaches 1, the number of pairings of points reproduced on average by the K-means algorithm increases; a mean Rand index of 1 computed this way implies it is unlikely for the K-means algorithm to cluster the provided points in any way other than the way that it did in the initial clustering.

Table 4.1[2] presents statistics collected from each program for a clustering made without the back pressure function. That is, the clustering was done, essentially, as a pure K-means run. This approach was chosen to investigate whether any program had a particularly local structure, with types very neatly clustered. K-means was run for each program from $K = 2$ up to either the number of points or 16, whichever was lesser. From these various clusterings, the one with the minimum Rand index was determined, and then the "empirical clustering," displayed here, was chosen by selecting the clustering with the greatest Rand index but K less than or equal to the K of the clustering with minimum Rand index. This process was chosen to maximize the number of clusterings inspected but avoid the degenerate case of picking a clustering with K close to the number of points, as these clusterings will always have a high Rand index but are of no interest to us. $K = 1$ was avoided throughout the procurement of all statistics for similar reasons.

The other statistic in Table 4.1 needing explanation is the L index. Since these clusterings

[2]Among the programs investigated, `blackscholes` was the only one to include absolutely no types or functions of any interest to Regraph, hence no data is reported for it. It is included here for the sake of completeness.

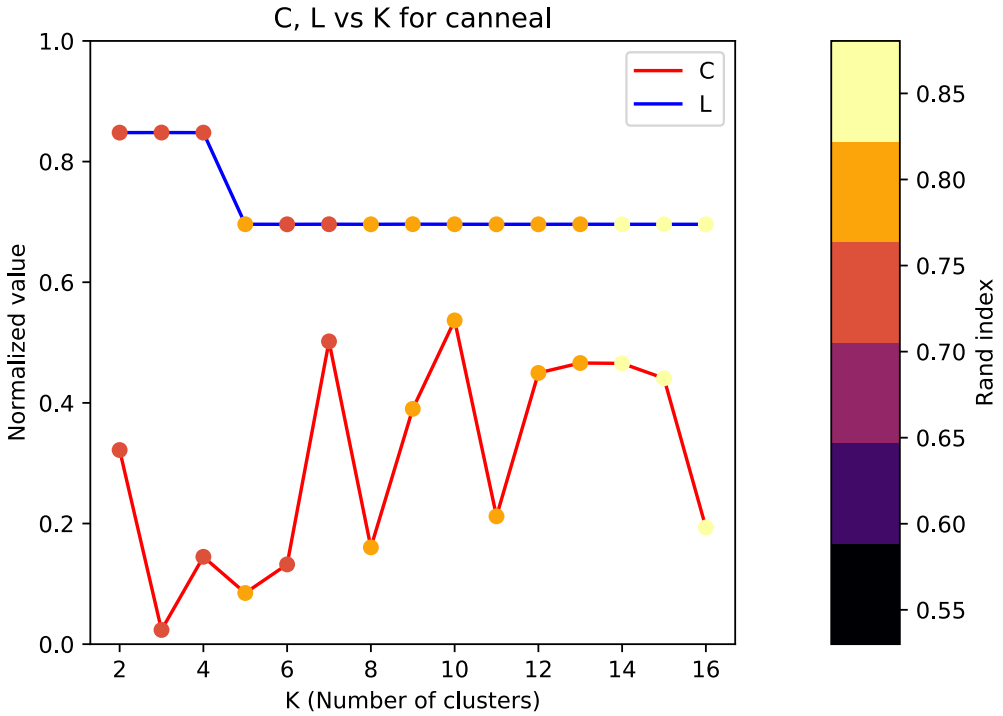

Figure 4.1: Locality inducement results for the `canneal` benchmark.

were found without use of an M value, the usual L could not be used. Instead, this index was computed by normalizing the weight of the largest cluster to the total vertex weight of the graph, to provide some insight into the distribution of type weight in the clustering.

Moving on to the matter of induced localities in the analyzed programs, we depict results for the `canneal` benchmark, easily the most interesting of the eight programs. Figure 4.1 presents the change in C and L for the program across a range of K values. The Rand index for each clustering is included as well. These results were all produced with parameters set to $\alpha = 1, M = 8 \star 10^6$. Graphs for the six other programs that produced results are available in the Appendix.

Finally, we present an attempt at describing the hardware design space implied by the results of the Aslo clusterings. Figure 4.2 presents a cross section of values for the α constant and the memory limit M. For each combination of parameters, several K-means runs were done as in Figure 4.1. C and L values were determined for each of these, and the

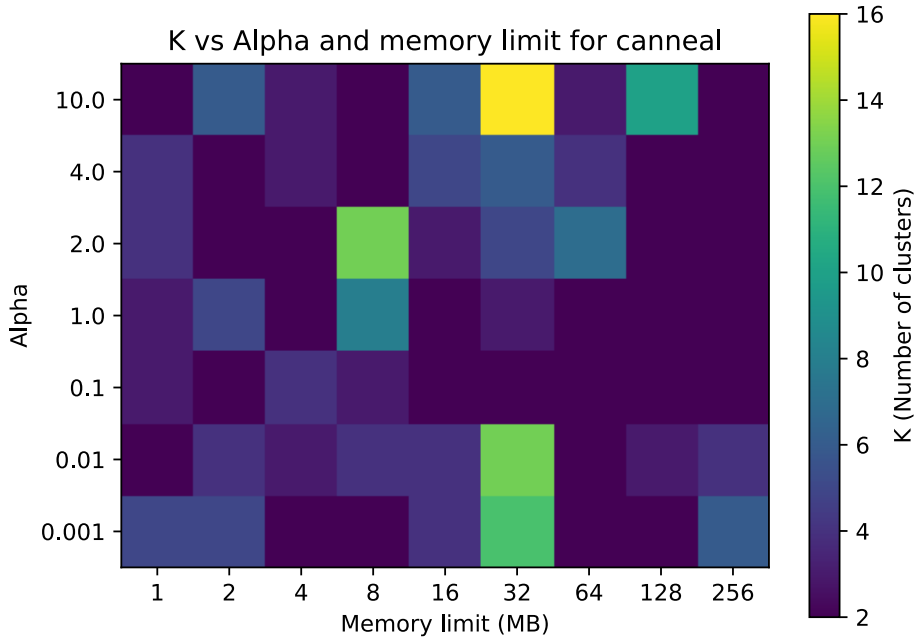

Figure 4.2: Hardware design space results for the `canneal` benchmark.

clustering with the lowest value $C + L$ was selected to represent the data point. Figure 4.2 therefore depicts what combinations of parameters were able to most effectively cluster the program. As before, graphs for the other programs are available in the Appendix.

The memory limit has an obvious analogue in hardware design; the alpha value, on the other hand, is meant to represent a spread of systems with varying effects of main memory access on run time. Together, they provide an insight into what parameters a hardware design might need to meet in order to effective make use of the locality present in the analyzed programs. While Figure 4.2 presents results solely for the `canneal` benchmark, Figure 4.3 presents the mean number of clusters selected for each design point across all programs, potentially suggesting some structure common across all the analyzed programs.

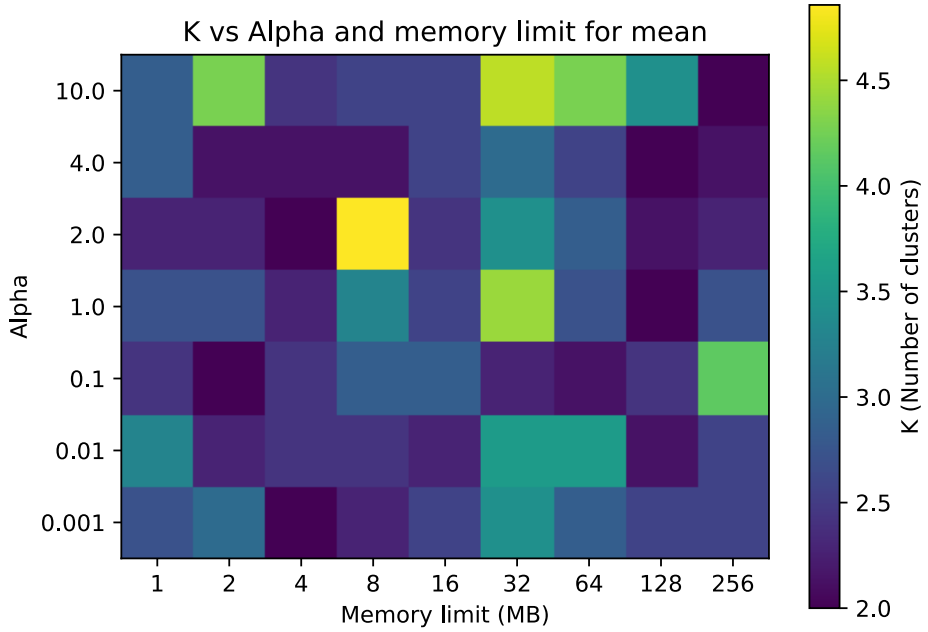

Figure 4.3: Mean of the hardware design space results for all programs.

4.3 Discussion

Beginning with the locality structures displayed in Table 4.1, it seems clear that a program that simply includes some object-oriented programming techniques, or makes use of an object-oriented library, is not sufficiently data encapsulated to demonstrate much in the way of clustering. None of the analyzed programs demonstrated stability at more than three clusters, and the majority could barely maintain two stable clusters, no matter the number of types recorded by the pass. Communication overhead was predictably low in all cases, and for the most part a single cluster accounted for the vast majority of the type weight, with the second cluster being largely superfluous. The `canneal` benchmark is a noteworthy exception to these patterns; while it too clustered at $K = 2$ and even then was not particularly stable, the two clusters seem to split the weight of the types roughly evenly and yet maintained a notably low communication overhead. This is a positive indicator of the behavior we desire.

Figure 4.1 corroborates the narrative of `canneal`'s encapsulation. As K increases we see a notable but not overwhelming increase in C, implying that as the clustering becomes more and more subdivided, communication overhead increases due to the gradual addition of functions between loosely connected types to the free edge weight, not from any large function being turned into overhead—though something like that pattern could explain the sharp jumps and dips that seem to self-correct at different values of K. `canneal`'s L trend implies that most of the type weight of the program is in a few very large nodes; as K, and thus the total memory of the system, increases, there is no change whatsoever in the latency overhead.

The hardware design space of `canneal` as depicted in Figure 4.2 strongly suggests that the benchmark has an affinity for a memory limit of 32 MB. However, it cannot be disambiguated from this figure whether the high-K clusterings reported resulted from a low C, low L, or both, merely that `canneal` is likely to perform well for higher clusterings at and around larger memory limits. The mean hardware design space seen in Figure 4.3 is somewhat more descriptive; the number of clusters averaged caps out at around $K = 5$, showing that the programs used do not cluster effectively for high K. There is some difficulty in attempting to pick out trends across the design space; mostly, it seems clear that higher cluster counts appear more often when the memory limit is high. This could potentially be a result of programs having multiple heavy types that communicate strongly that are forcibly split at smaller memory limits; having more memory available means these communications can avoid being made inter-locality.

While the results of the analysis of these eight programs do not imply a readily available implicit locality among many common workloads, particularly when written without much care for data encapsulation or object-oriented techniques, this is not such a surprise. The results from the `canneal` benchmark, slim as they are, do suggest that the characterization approach and implementation have merit and potential in the analysis and description of local structures in code.

Chapter 5

Conclusion

5.1 Future work

Opportunities for future work in this field are many and likely to be fruitful. One of the primary weaknesses of the Regraph pass is its position in the workflow of code generation. Complex projects more likely to take advantage of well-encapsulated code are exactly those projects that are difficult to compile to LLVM IR or bitcode, shutting out potential sources of interesting data. However, projects in the LLVM ecosystem such as the `emscripten` project, a largely drop-in C/C++ to JavaScript compiler suite using LLVM that provides convenience tools for instrumenting pre-existing build systems, could potentially be modified to ease the process of building large projects into forms ready for analysis and subsequent execution [13].

While the use of K-means in clustering types has several nice properties, other clustering methods could potentially reveal fine-grained or otherwise unusual clusters that K-means is unable to detect or exploit.

As types too large for any reasonable memory are an obvious drawback of this approach, splitting large types among multiple localities is an avenue of analysis that could bear potentially interesting results, particularly in moving towards a more practical locality scheduling algorithm.

5.2 Conclusion

Though the applicability of the Regraph LLVM pass and the Aslo analysis utility are currently limited by the scope of programs that can be compiled to LLVM bitcode, limited positive results suggest that the approach itself has merit in characterizing the locality of a program. None of the programs analyzed demonstrated stable clusterings for high cluster counts naturally, but inducing high cluster counts with back pressure preventing overly large grouping of nodes met with some success. The inter-locality communication overhead in these

cases was kept to a minimum by the properties of the type graph construction and analysis technique in K-means. Complementary negative results for programs not demonstrating data encapsulation in type also suggest the ability of the approach to differentiate programs with implicit local structure. Avenues of future work in expanding the functionality and applicability of the implementation are promising.

Bibliography

[1] R. Balasubramonian et al. "Near-Data Processing: Insights from a MICRO-46 Workshop". In: *IEEE Micro* 34.4 (July 2014), pp. 36–42. ISSN: 0272-1732. DOI: 10.1109/MM.2014.55.

[2] Christian Bienia et al. "The PARSEC Benchmark Suite: Characterization and Architectural Implications". In: *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*. PACT '08. Toronto, Ontario, Canada: ACM, 2008, pp. 72–81. ISBN: 978-1-60558-282-5. DOI: 10.1145/1454115.1454128. URL: http://doi.acm.org/10.1145/1454115.1454128.

[3] Arup De et al. "Minerva: Accelerating Data Analysis in Next-Generation SSDs". In: *Proceedings of the 2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines*. FCCM '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 9–16. ISBN: 978-0-7695-4969-9. DOI: 10.1109/FCCM.2013.46. URL: http://dx.doi.org/10.1109/FCCM.2013.46.

[4] Jeff Draper et al. "The Architecture of the DIVA Processing-in-memory Chip". In: *Proceedings of the 16th International Conference on Supercomputing*. ICS '02. New York, New York, USA: ACM, 2002, pp. 14–25. ISBN: 1-58113-483-5. DOI: 10.1145/514191.514197. URL: http://doi.acm.org/10.1145/514191.514197.

[5] Maya Gokhale, Bill Holmes, and Ken Iobst. "Processing in Memory: The Terasys Massively Parallel PIM Array". In: *Computer* 28.4 (Apr. 1995), pp. 23–31. ISSN: 0018-9162. DOI: 10.1109/2.375174. URL: https://doi.org/10.1109/2.375174.

[6] Omer Khan, Mieszko Lis, and Srini Devadas. "Em2: A scalable shared-memory multicore architecture". In: (2010).

[7] Wen Li et al. *Approximate Nearest Neighbor Search on High Dimensional Data — Experiments, Analyses, and Improvement (v1.0)*. 2016. arXiv: 1610.02455 [cs.DB].

[8] David Patterson et al. "A Case for Intelligent RAM". In: *IEEE Micro* 17.2 (Mar. 1997), pp. 34–44. ISSN: 0272-1732. DOI: 10.1109/40.592312. URL: https://doi.org/10.1109/40.592312.

[9] William M. Rand. "Objective Criteria for the Evaluation of Clustering Methods". In: *Journal of the American Statistical Association* 66.336 (1971), pp. 846–850. DOI: 10.1080/01621459.1971.10482356. eprint: https://www.tandfonline.com/doi/pdf/10.1080/01621459.1971.10482356. URL: https://www.tandfonline.com/doi/abs/10.1080/01621459.1971.10482356.

[10] Harold S. Stone. "A Logic-in-Memory Computer". In: *IEEE Trans. Comput.* 19.1 (Jan. 1970), pp. 73–78. ISSN: 0018-9340. DOI: 10.1109/TC.1970.5008902. URL: https://doi.org/10.1109/TC.1970.5008902.

[11] Xulong Tang et al. "Data Movement Aware Computation Partitioning". In: *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-50 '17. Cambridge, Massachusetts: ACM, 2017, pp. 730–744. ISBN: 978-1-4503-4952-9. DOI: 10.1145/3123939.3123954. URL: http://doi.acm.org/10.1145/3123939.3123954.

[12] L. Yavits, A. Morad, and R. Ginosar. "Computer Architecture with Associative Processor Replacing Last-Level Cache and SIMD Accelerator". In: *IEEE Transactions on Computers* 64.2 (Feb. 2015), pp. 368–381. ISSN: 0018-9340. DOI: 10.1109/TC.2013.220.

[13] Alon Zakai. "Emscripten: an LLVM-to-JavaScript compiler". In: Oct. 2011, pp. 301–312. DOI: 10.1145/2048147.2048224.

[14] Dong Ping Zhang et al. "A New Perspective on Processing-in-memory Architecture Design". In: *Proceedings of the ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*. MSPC '13. Seattle, Washington: ACM, 2013, 7:1–7:3. ISBN: 978-1-4503-2103-7. DOI: 10.1145/2492408.2492418. URL: http://doi.acm.org/10.1145/2492408.2492418.

[15] Dongping Zhang et al. "TOP-PIM: Throughput-oriented Programmable Processing in Memory". In: *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*. HPDC '14. Vancouver, BC, Canada: ACM, 2014, pp. 85–98. ISBN: 978-1-4503-2749-7. DOI: 10.1145/2600212.2600213. URL: http://doi.acm.org/10.1145/2600212.2600213.

Chapter 6

Appendix

Graphs of locality inducement results and hardware design space analysis for the remaining programs are presented here.

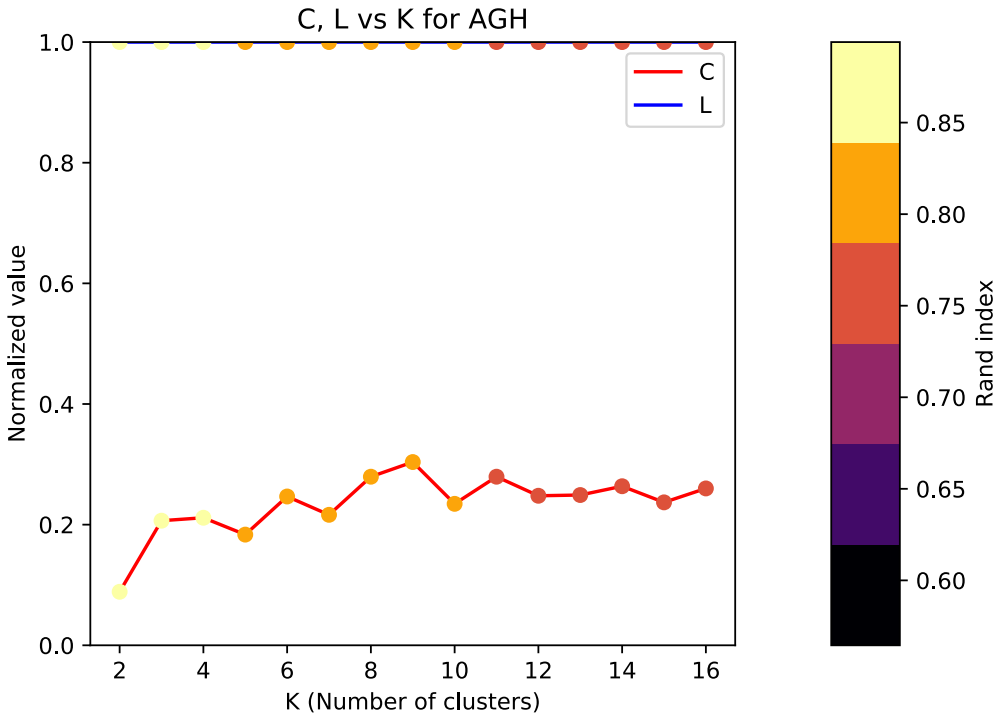

Figure 6.1: Locality inducement results for the `AGH` benchmark.

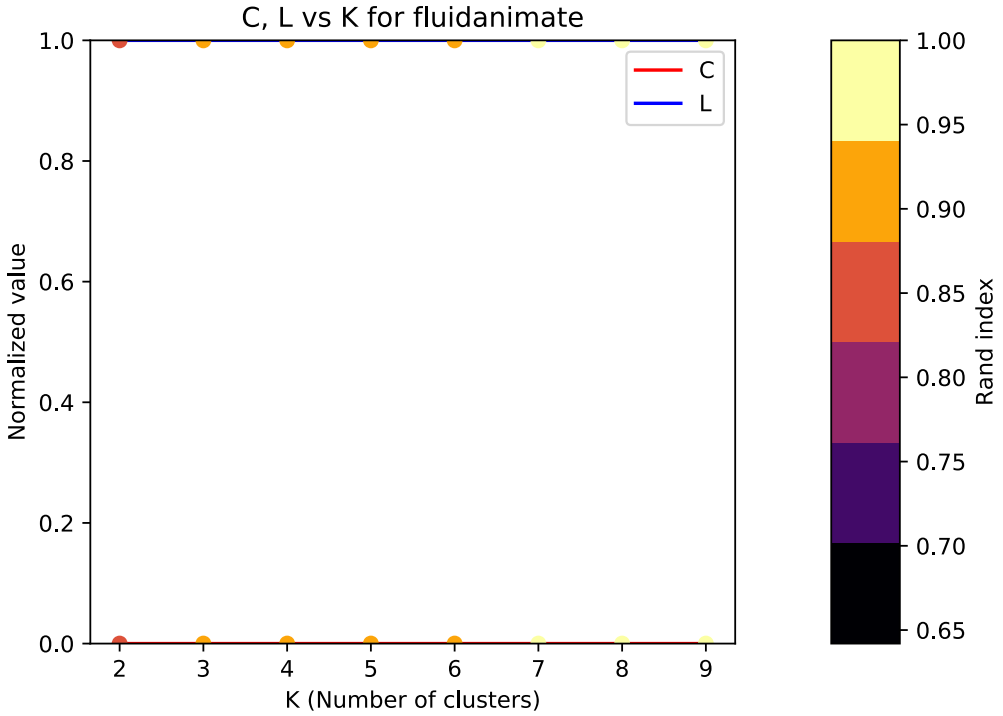

Figure 6.2: Locality inducement results for the `fluidanimate` benchmark.

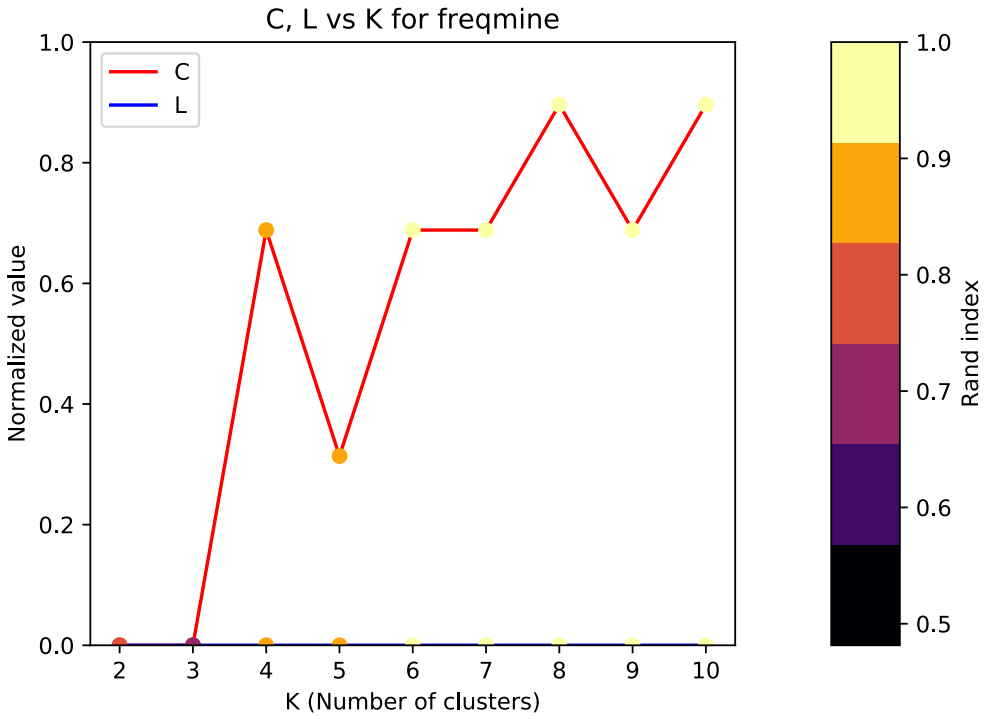

Figure 6.3: Locality inducement results for the `freqmine` benchmark.

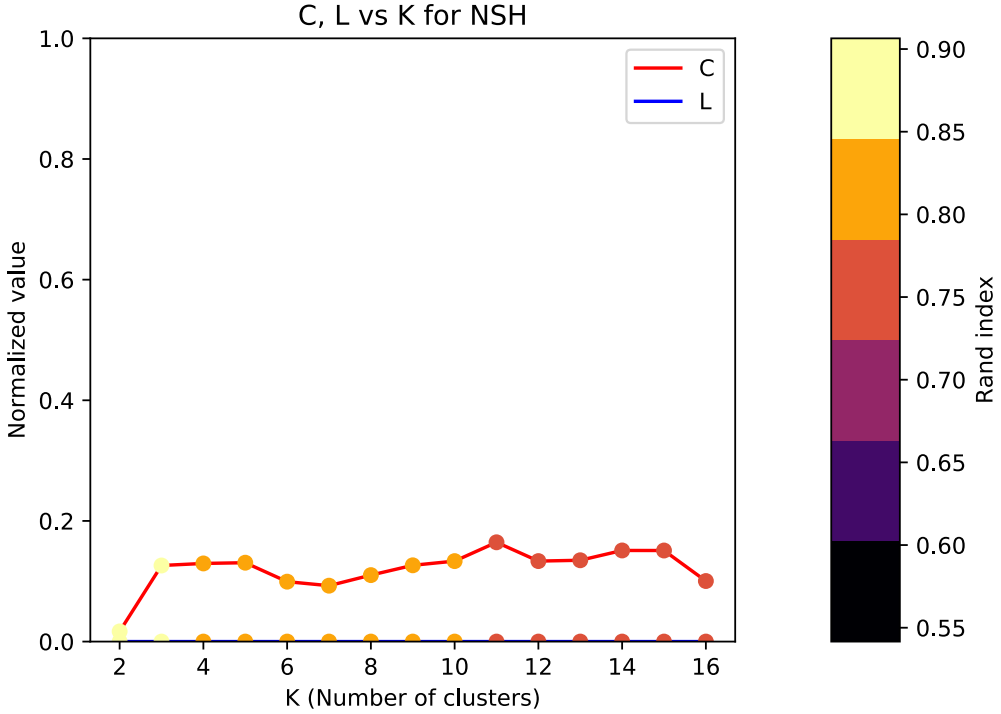

Figure 6.4: Locality inducement results for the `NSH` benchmark.

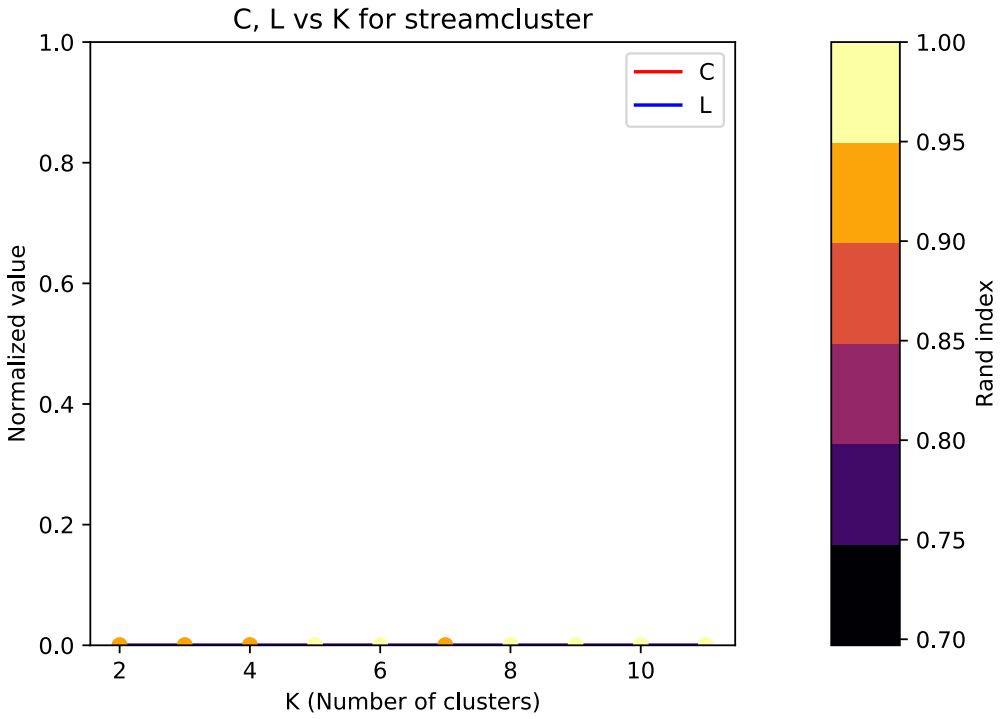

Figure 6.5: Locality inducement results for the `streamcluster` benchmark.

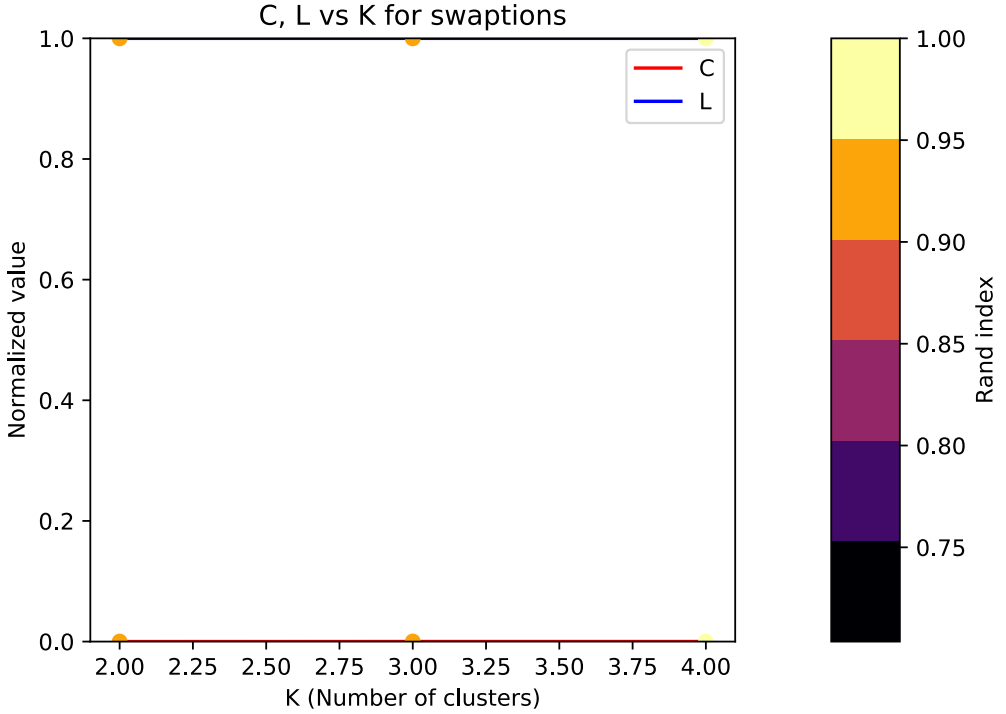

Figure 6.6: Locality inducement results for the `swaptions` benchmark.

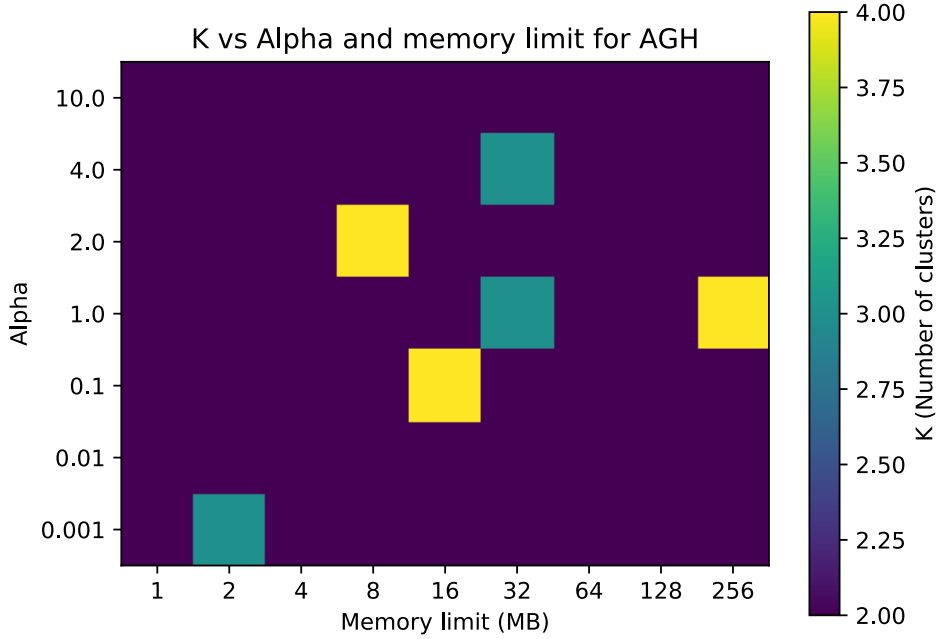

Figure 6.7: Hardware design space for the `AGH` benchmark.

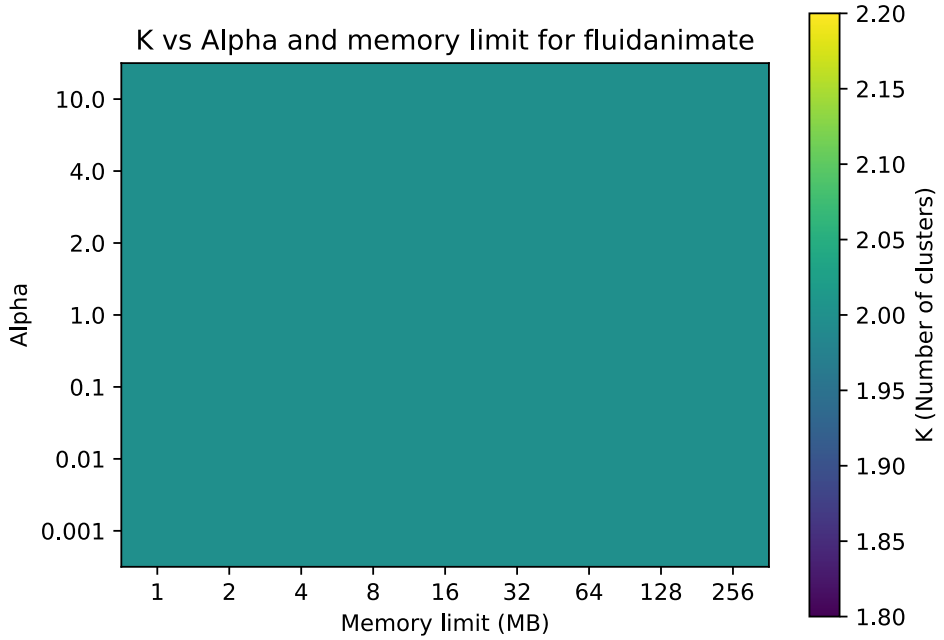

Figure 6.8: Hardware design space for the `fluidanimate` benchmark.

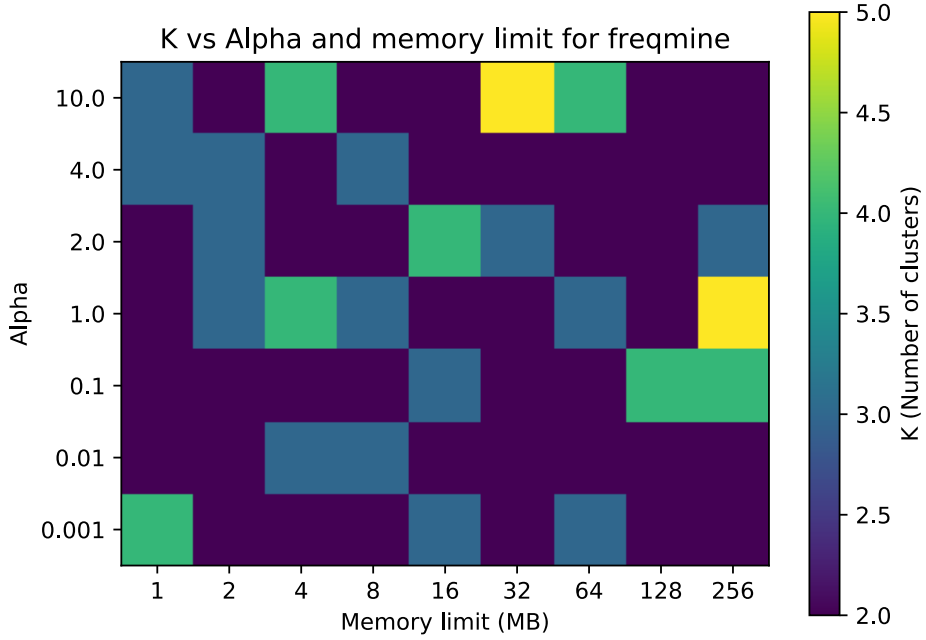

Figure 6.9: Hardware design space for the `freqmine` benchmark.

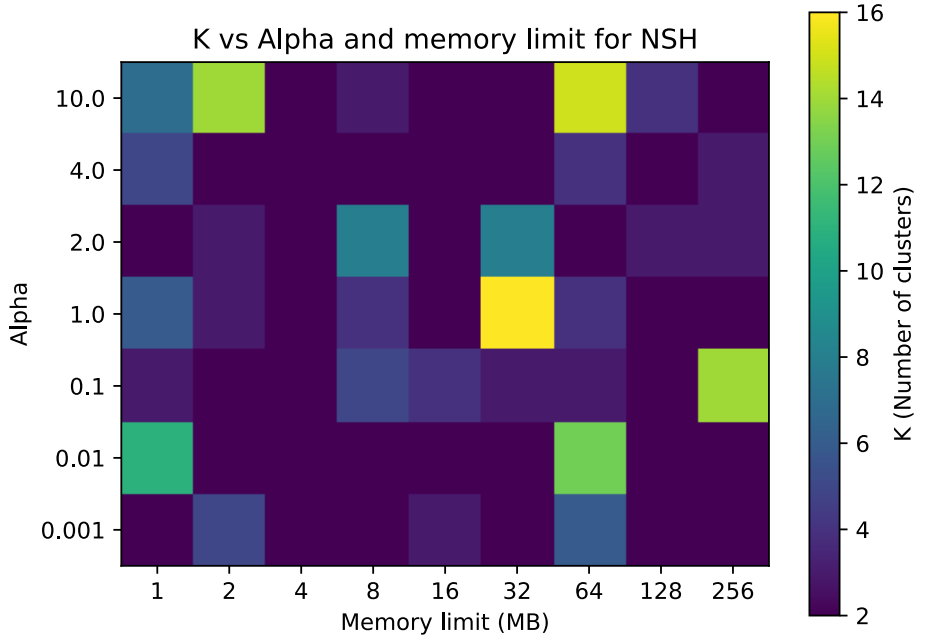

Figure 6.10: Hardware design space for the `NSH` benchmark.

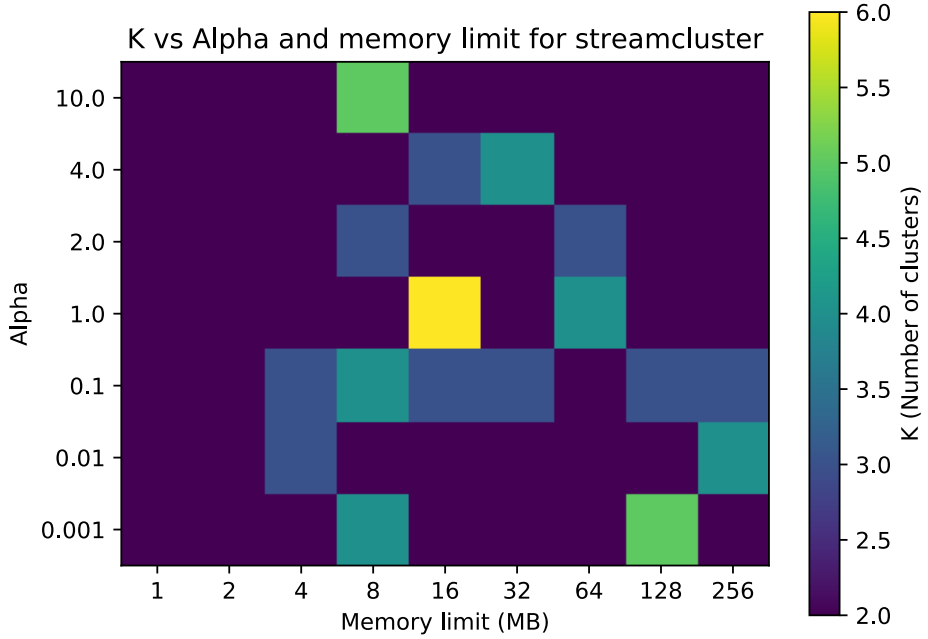

Figure 6.11: Hardware design space for the `streamcluster` benchmark.

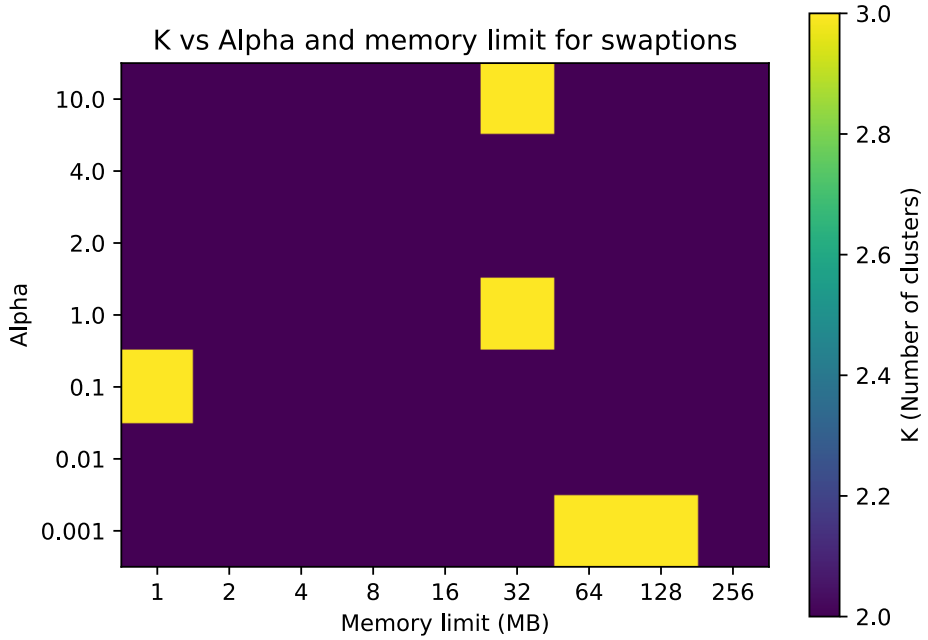

Figure 6.12: Hardware design space for the `swaptions` benchmark.

Academic Vita

Jack Kinsey

Education

The Pennsylvania State University Class of 2019
- The Schreyer Honors College
- Bachelor's in Computer Science
- The College of Engineering
- Dean's List *(Fall 2016–Present)*

Conestoga Valley High School Class of 2016

Experience

Summer Games intern at Booz Allen Hamilton in Washington, D.C. Summer 2018
- Worked alongside a team of 4 to deliver a functional software product for use internally
- Led development of a JavaScript/HTML/CSS user interface from scratch
- Integrated with a custom Python backnd built to do graph analytics

Web intern at Fox Chapel Publishing in East Petersburg, PA Summer 2016–Summer 2017
- Migrated 3+ WordPress sites and 1 forum from one hosting provider to another
- Customized WordPress CSS for 4+ blogs
- Presented Google Analytics data on 3+ sites to superiors
- Prepared over 40 articles worth of backlog content for web publishing
- Managed DNS records for 10+ sites
- Provided technical support to over 400 active users across 2 forum sites
- Helped manage Magento web commerce backend

Independent web developer, programmer, and sysadmin 2013–Present
- Host and maintain a professional and personal website
- Develop sites and web applications for friends, fun, and other commitments
- Build and maintain an Arch Linux desktop

Activities

Panels Coordinator, Setsucon 2018 Fall 2017–Winter 2018
- Curate panel submissions from nearly 40 panelists
- Design the schedule for a 2 day convention of over 1000 people
- Coordinate with a team of 20+ fellow volunteers

Webmaster, Penn State Anime Organization Fall 2016–Present
- Provide weekly schedule updates to the club WordPress site
- Attend regular meetings to contribute to the administration of the club

Penn State International Collegiate Programming Contest Fall 2016–Spring 2017
- Exercised programming skills as part of a team of 3
- Solved problems in a competitive context at the Ohio State ICPC

Skills

Programming
- C, C++, Python, JavaScript, HTML, CSS, Bash

Tools
- GNU text utilities, Git, LaTeX, SQL, Wordpress

Systems
- Arch Linux, Debian Linux, Windows

Writing
- Author papers for classes and write blogs on games, hardware, software