Санкт-Петербургский государственный университет

Математическое обеспечение и администрирование информационных систем
Кафедра системного программирования

Малютин Данила Павлович

Выпуская квалификационная работа

Распределение данных программ для многобанковых гетерогенных архитектур памяти

Научный руководитель:
д. ф.-м. н., профессор Терехов А. Н.

Консультант:
технический руководитель., Huawei Якушкин С. И

Рецензент:
инженер-программист, ООО ”Синопсис СПб” Антрушин Д. Ю

Санкт-Петербург
2019

SAINT-PETERSBURG STATE UNIVERSITY

Information Systems Administration and Mathematical Support
Software Engineering

Malyutin Danila

Data assignment for multibank heterogeneous memory architectures

Graduation Project

Scientific supervisor:
professor Andrey Terekhov

Consultant:
technical lead, Huawei Sergey Yakushkin

Reviewer:
software engineer, Synopsys, Inc. Antrushin Denis

Saint-Petersburg
2019

Contents

Introduction

Many modern computing systems use highly specialized heterogeneous memory architectures to achieve greater performance or better power consumption. The heterogeneous memory architecture is the most relevant for embedded systems on a chip, DSP, microprocessors for the Internet of things [20], machine learning tasks and computer vision, as well as high-performance multiprocessor computing systems with shared memory. Data access is the usual bottleneck of such applications, and homogeneous architectures are less efficient than specialized solutions for applications with regular data flow [19]. One common solution is to use several independent physical memory banks, divided by address ranges. This architecture allows you to increase memory bandwidth due to the fact that requests for data from different memory banks can be carried out in parallel.

Heterogeneous memory architectures require special support in the software model and high-level programming languages. For example, C89 uses a flat memory model and the programmer has full control over the placement of objects in memory. However, for working with heterogeneous memory architectures, there are extensions of the C99 standard that add address space attributes [21], which allow you to specify memory regions for objects and pointers. In OpenCL, support for heterogeneous memory architectures is built into the standard. Data can be located in several types of memory and there is a host API that allows you to manage memory allocation and data transfers. However, OpenCL memory is divided into predefined types (local, global, private, constant, etc.), which corresponds to the GPU, but does not cover more specialized computing devices with multi-bank architecture. Virtual machine-based languages such as Java usually leave the programmer less control over memory allocation. The task of the distribution of objects and garbage collection in memory falls on the execution environment.

With the increasing complexity of modern programs and algorithms, increases the need for high-level programming languages to reduce system development time. On the other hand, the competition of device manufacturers is growing, and specialized architectural solutions are used increasingly more often.

Allocating program data for heterogeneous memory by annotating data types with address spaces is a time-consuming task that requires careful analysis of the program data flow and profiling of the critical sections of code. Manual data distribution can lead to errors and does not guarantee the final performance of the program, since it is necessary to take into account possible conflicts of access to memory banks and how data is used at the level of the entire program.

The variety of devices, tasks and applications of processors leads to the need to quickly and efficiently adapt existing applications to new configurations and memory structure.

In this work we try to solve this problem by providing a solution that automates data assignment between memory banks.

1 Problem statement

The primary aim of this work is to design and implement a solution that would reduce the amount of manual work needed to adapt a program to a new memory configuration. This solution should be capable of doing whole program data flow analysis of C and C++ code to automatically map the static data to the appropriate memory banks of multibank heterogeneous architecture in a way that maximizes parallel access.

To achieve this goal the following tasks have been formulated:

- review existing approaches and algorithms used in them
- design the solution
- implement the solution
- evaluate implementation

2 Related work

The problem of bank assignment is similar to the problem of register allocation. Indeed, in both cases we have a limited number of resources (registers and banks) which need to be assigned to some data in the most optimal way with the condition that different values can't occupy the same resource at once, though it is only a soft requirement for memory banks. As such, we reviewed the works related to both register and bank allocation.

2.1 Bank allocation

There have been several works that tackle the problem of data assignment for dual memory banks. An early approach for dealing with dual memory banks in C is presented in [13]. In this work, the solution is implemented by expanding the original Gepard C compiler with extra passes and running the backend twice. An initial run of the backend is used to determine the exact set of memory accesses, which is then used to back-annotate the IR for the second and final backend run. For finding the optimal assignment to the memory banks, they use an Integer Linear Programming (ILP) approach.

Solution proposed in [24] operates on IR generated by the compiler. It uses special independence graph and solves weighted Max-Cut problem. It takes execution frequency and access proximity into account but it is not clear how well it translates into the final machine code.

Another work that takes ILP approach is [16], which operates on the source level converting C code into DSP C. As a model for program data usage, it uses the Interference Graph. It also proposes two additional algorithms, one based on graph soft coloring and the other based on Genetic Programming, which both perform better than ILP approach.

The ILP is also used by the [11] work. They operate on the synchronous data flow [12] specifications and the simple conflict graphs that accompany such programs. However, they only evaluate their algorithm against benchmarks for which two-coloring exists, so their technique is not demonstrated to work on hard problems.

Finally, in [15] the problem of bank assignment is considered in the context of multi-core systems with shared memory. In such systems multiple banks are exploited by assigning data in a way that minimizes simultaneous access to some memory bank from multiple cores. This work proposes general models for multi-core system and executed parallel program. The algorithm uses graph coloring via collision graphs with multiple heuristics and relies on a run-time statistics of the problem tasks collected during the initial execution of the program. The assignment is when improved in multiple consecutive iterations with updated statistics on each run.

Below is a table with a short summary of some reviewed papers.

Paper	Algorithmic approach	Works on	Resulting tool
[16]	ILP, Stochastic Coloring, Genetic Programming	C source code	ISO C to DSP-C compiler
[24]	Max Cut problem solving	IR	IR optimization pass
[13]	ILP	ASM	Additional compilation phases
[11]	ILP	SDF representation	NA
[15]	Iterated Greedy Coloring based on profiling data	NA	NA

2.2 Register allocation

Until recently common approach to register allocation was to view the allocation to k registers as k-coloring of the graph problem and run several iteration of a greedy register allocation algorithm trying to improve the results on each run [2, 1].

More recent works rely on the fact that the chordal graphs can be optimally k-colored in polynomial (even linear) time and that in many languages interference graphs for most functions turn out to be chordal [18]. Furthermore, in [5] it is shown that interference graph for any strict program is chordal. This is exploited for fast k-coloring, however the real complexity comes from spilling and coalescing steps which are necessary when there is not enough registers. [22] explains that the spilling, coalescing and respective optimization problems are what makes register allocation NP-complete.

3 Design

In this section, we define memory model that we use to describe our target memory architecture, give the formal definition of the model used to represent program's data flow and of the problem we are going to solve, then we propose the top-level architecture for the solution and describe the general tool flow.

3.1 Memory model

Multibank heterogeneous memory architectures have many different applications and as such can vary greatly in their specifics. For example, 256 Andey processor [26] is a multicore system used in high performance computing which takes advantage of multibank heterogeneous memory architecture. This processor integrates 256 Processing Elements (PE), which are grouped into 16 compute clusters. Each compute cluster has a local on-chip memory with 2 MB capacity, which is organized in 16 independent SRAM banks. These are arranged in two sides (left and right). The PEs are organized in 8 pairs. Each pair has two memory buses (one for each side), which can be utilized in parallel by the two cores.

Other frequent users of multiple memory banks are systems on a chip and DSPs. DSP processor based embedded systems have an on-chip memory, which typically has a single cycle access time [17]. The on-chip memory, also referred to as scratch pad memory or closely coupled memory, is mapped into an address space disjoint from the off-chip memory but connected to the same address and data buses. Typically, the scratch-pad memory is organized into multiple memory banks to facilitate multiple simultaneous data accesses. DSP Processors typically have 2 or more address generation units and multiple on-chip buses to facilitate multiple memory accesses.

Further, each on-chip memory bank can be organized either as a single-access RAM (SARAM) or as a dual-access RAM (DARAM), to provide single or dual accesses to the same memory bank in a single cycle. For example, Texas Instruments TMS320C54X digital signal processor has two data read buses and one data write bus [8] and, Texas Instruments TMS320C55X processor has three data read buses and two data write buses, since concurrent access to the same array are common in DSP applications [9]. The DARAM and SARAM regions can be recognized using multiple memory banks to enable two concurrent accesses.

In this work our focus is on the problem of data mapping optimization for DSPs. As such, we aimed to create a memory model that would allow to sufficiently approximate the most common multibank memory configurations found in typical DSPs as well as being simple enough to define.

Definition 1 A memory configuration is a tuple (B, c, p), with $B = \{b_1, ..., b_{n_B}\}$ a set of disjoint memories (memory banks). $c : B \to \mathbb{N}$ gives the capacity of each memory and $p : B \to \mathbb{N}$ gives the penalty of the load/store operation to this memory.

3.2 Data usage model and formal problem definition

One of the common ways to represent program data flow in the presence of parallel access conflicts is an Interference Graph (IG) which was proposed in [23] and used with slight modifications in several related works [13, 15, 16, 24].

Definition 2 We will call a weighted graph $G = (V, E)$ an Interference Graph for some program P if each global variable p in P has a corresponding $v_p \in V$ and for each potentially parallel access between variables $p1$ and $p2$ there exists an edge $(v_{p1}, v_{p2}) = e \in E$. Weight function $w : E \to \mathbb{N}$ maps each interference edge with the number roughly equivalent to the number of parallel accesses represented by that edge. Because of their simplicity, popularity and the fact that graphs are easily representable both visually and textually we settled on Interference Graphs as our data usage model.

An example of a basic C program that calculates dot products of some arrays and its corresponding Interference Graph can be seen on fig. 1.

Now we have everything we need to formally define the task we are going to solve. For a given memory configuration (B, c, p) and an Interference

```
int a[100], b[100], c[100];

int dot_product(const int *x, const int *y, int n) {
  int acc = 0;

  for(int i = 0; i < n; i++) {
    acc += x[i]*y[i];
  }

  return acc;
}

int main() {
  return dot_product(a, b, 100) + dot_product(a, c, 50) + dot_product(b, c, 50);
}
```

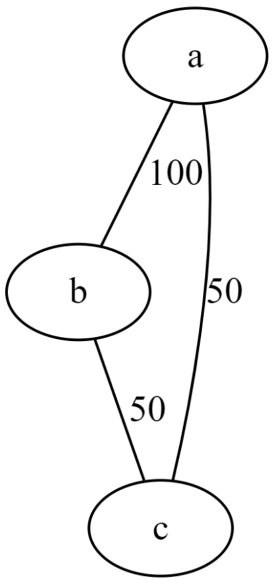

Figure 1: Program written in C and its corresponding Interference Graph

Graph $G = (V, E)$ we want find such data mapping $m' : V \to B$, that $m' = \arg\min_{m} \sum_{m(v_i) \neq m(v_j)} w((v_i, v_j))$

In other words, we want to find mapping of program's variables to memory banks that gives minimum total interference, which is the sum of weights of the edges between interfering (in the same memory) vertices.

3.3 Tools

The data assignment problem can be split into three major subtasks:

1. whole program analysis to build data usage model
2. identification of the optimal data mapping based on this analysis
3. application of the selected data mapping to the original program

The choice of Interference Graphs as a data usage model allowed us to separate the solution into two independent tools, because graphs can be easily represented in textual form which makes it easy to communicate the results of one tool to another.

The two tools were called Analyzer and Allocator.

Analyzer does the first subtask of analyzing program's data flow and produces the Interference Graph, which is then taken as an input to the

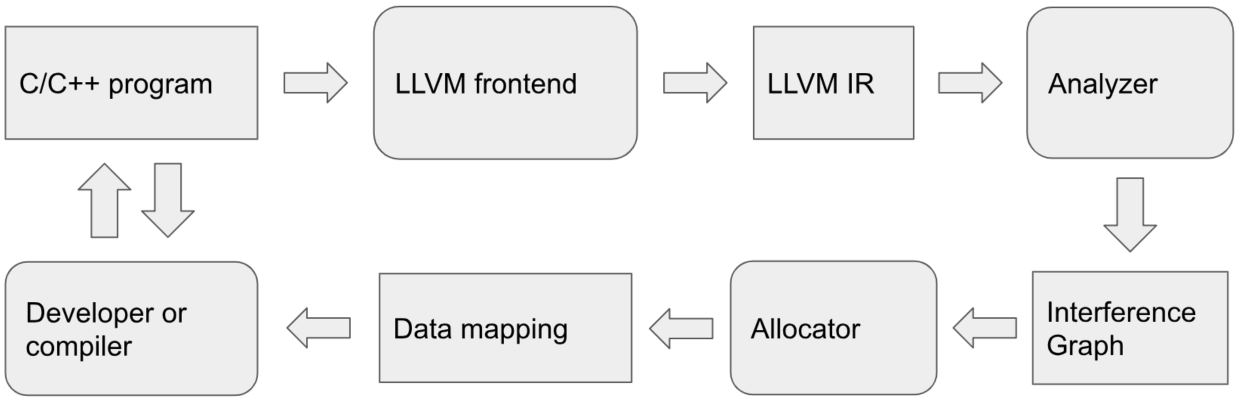

Figure 2: The structure of passes in the analyzer

Allocator that tries to come up with the optimal mapping of data to memory banks.

Such division makes it possible to test and develop both tools independently since Allocator can consume manually crafted Interference Graphs while Analyzer's output is useful for the developer by itself for manual optimizations.

3.3.1 Analyzer

Since we need to support programs written in C and C++ languages, we had to either implement whole C/C++ parsing and analysis ourselves or use an existing solution. Because parsing C/C++ is notoriously hard we chose to implement our tool using powerful cross-platform compiler framework LLVM.As a result, our Analyzer does not work on C/C++ code directly, but analyzes IR produced by LLVM using provided mechanism of code passes. The final tool flow is shown in fig. 2. First, C/C++ program is compiled to LLVM IR, for which then Analyzer produces the Interference Graph. Allocator takes this graph and produces the data mapping which is then used by either the compiler or developer to annotate the original program.

3.3.2 Allocator

Allocator takes the Interference Graph produced by Analyzer, memory configuration, such as the number of available memories and their size, and

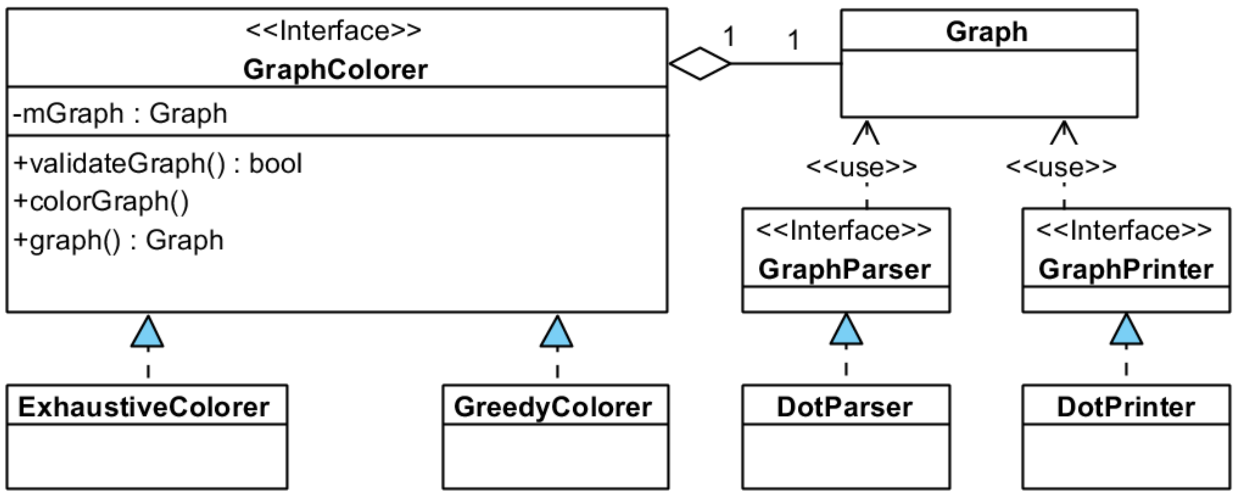

Figure 3: The simplified class diagram of the Allocator

produces a mapping of program variables to appropriate memory banks. Since each variable is represented by the vertex in the Interference Graph, this task can be reduced to the problem of coloring the input graph in a way that will minimize the interference, i.e. sum of edge weights between vertices of the same color, there each color corresponds to specific memory bank.

Main requirement for Allocator was the ability to easily add new algorithms to test and develop new coloring heuristics. Its simplified UML class diagram is shown in fig. 3. Each coloring algorithm implements the GraphColorer interface which holds Graph read from the input in the proper format and returns its colored version.

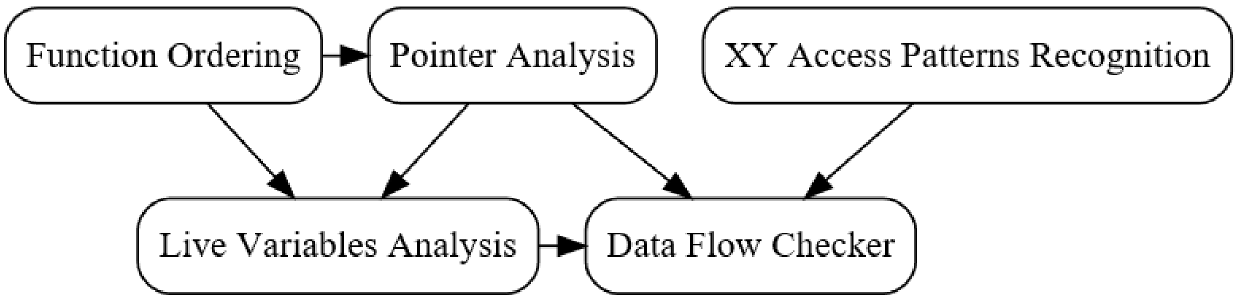

Figure 4: The structure of passes in the analyzer

4 Implementation

In this section, we describe the implementation details of the Analyzer and Allocator tools, such as the data format chosen for Interference Graph and constraints representations. We also explore different implemented coloring algorithms used by the Allocator.

4.1 Analyzer

Analyzer consists of several LLVM passes that analyze program's IR produced by the LLVM backend to build Interference Graph. The way those passes interact is show on fig. 4. Here, an arrow pointed from pass A to pass B means that pass B uses the result of pass A.

Function Ordering This pass analyzes program's call graph and produces two function orderings: top-down and bottom-up, i.e. from top (main) to bottom (leaf functions) and reverse. Currently this pass ignores recursion and indirect function calls, i.e. calls done through a pointer. However, this is was not have a significant impact on our use case since such constructs are rarely used in the typical DSP programs or at least in their performance-critical parts.

Pointer Analysis This pass takes orderings produced by the Function Ordering pass and tries to build a model of global data usage through pointer analysis, i.e. it attempts to answer for each pointer in the IR which data can it point to. It does so by walking functions first in a bottom to top ordering to build a template of pointer usage for each function parametrized by their pointer arguments. Then it walks from top to bottom substituting template parameters with concrete data known at the call site.

Live Variables Analysis This pass uses Pointer Analysis to determine when certain global variables are live and when they become dead.

XY Access Patterns Recognition This pass is used to mark "interesting instructions", that is IR instructions that can be compiled into machine instructions utilizing parallel bank access. This is mostly dependent on target architecture and should be implemented for each CPU.

Data Flow Checker This is the final pass that takes analysis results collected so far by the previous passes and uses them to construct the final Interference Graph.

4.1.1 Textual graph representation

For textual representation of the Interference Graph we chose dot format [3]. This format has the advantages of being human readable, versatile, and supported by many programming languages and tools which makes it easier to consume Analyzer results by Allocator, visualization tools and other programs. Edges weights and vertices properties (such as their size and number of uses) are encoded in their labels. Below is the dot representation of the Interference Graph of the program from fig. 1.

```
graph ''main'' {
   0 [shape=record,label=''{a|s:400 u:150}''];
   1 [shape=record,label=''{b|s:400 u:150}''];
   2 [shape=record,label=''{c|s:400 u:100}''];
   3 [shape=record,label=''{Other|s:0 u:0}''];

   0 -- 1 [label=''100'' style=''bold''];
   0 -- 2 [label=''50'' style=''bold''];
   1 -- 2 [label=''50'' style=''bold''];
}
```

While printing the Interference Graph in a dot format is easy due to the format's powerful syntax, parsing it reliably can be much trickier so we delegate that to the graphviz C library used by Allocator.

4.2 Coloring algorithms

We have implemented several graph coloring algorithms. In this section, we give their description.

Below we use the following notation:

G	Interference Graph
E	The set of G's edges
V	The set of G's vertices
n	The number of vertices $\|V\|$
k	The number of available colors (memories)
$\chi(G)$	The chromatic number of G, that is, the minimum number of colors required to color the vertices of G in such a way that no two adjacent vertices share the same color
$\omega(G)$	The clique number of G, that is, the number of vertices in a maximum clique in G

4.2.1 Branch and Bound

The first algorithm is a precise algorithm based on exhaustive search. We iterate over all k^n possible colorings and in $\Theta(m)$ time we check if a coloring is minimal and satisfies color size constraints. Overall running of the such algorithm is therefore $\Theta(m \cdot k^n)$.

This is a good starting point considering the problem of graph coloring is NP-complete [25], but it can be refined further. For example, in practice we can significantly decrease the number of colorings considered and improve the typical running time, if we apply a well-known technique called branch-and-bound. For each partial coloring of G (e.g., coloring, where only $n' < |V|$ vertices were colored) we estimate lower end bounds for cost function value. If the lower bound of some partial coloring is bigger than global minimal upper bound of some other partial coloring, we conclude than this partial coloring, however good we color the rest of the graph, cannot induce minimal coloring, so we ”cut” on current partial coloring and go on with the next one. In our implementation, lower bound is calculated by applying cost function to a sub-graph G' of G where vertices of G' are already colored vertices of G, and edges of G' are edges of G connecting already colored vertices. Upper bound is cost function value of current minimal coloring.

Intuition and benchmarking tells us that the running time of the algorithm can be greatly decreased, if we find a good ordering in which we consider vertices. Indeed, we want to ”cut” on ”bad” partial colorings as soon as possible, so that we could omit lots of unnecessary checks. The ordering must depend on some ”vertex importance” criterion, where most ”important” vertices come first in the ordering. In our implementation, we consider vertex a ”more important” than vertex b, if $deg(a) > deg(b)$ (e.g., if variable a interferences with more variables than b).

We can also omit checks of the colorings that differ only by the permutation of colors when all memories have identical load penalty since in this case the permutation of colors would not change the overall penalty. We say that colorings a and b are members of the same penalty equality class, if coloring a is the same as coloring b for some permutation of colors in a. for example, colorings RED-BLUE-BLUE and BLUE-GREEN-GREEN are in the same equality class, because if we apply [RED:=BLUE, BLUE:=GREEN, GREEN:=RED] substitution to the coloring RED-BLUE-BLUE, it becomes BLUE-GREEN-GREEN. The number of penalty equality classes is $\lfloor k^n/k! \rfloor$, so we can improve our algorithm further by checking just one member of the penalty equality class rather than checking them all, and then permute all the colors to fit into memory constraints.

After the aforementioned optimizations, the running time of exhaustive search becomes bound by $\mathcal{O}(m \cdot (k^n/k! + k!))$, in pathological cases and much better run time in the typical cases which can be reduced even more by implementing a more sophisticated ”vertex importance” criterion.

4.2.2 Greedy coloring

We have implemented the optimal solution that finds the best coloring possible, but since graph coloring is NP-complete the optimal algorithm is bound to be at least sub-exponential in the number of vertices of IG.This means that waiting for the optimal algorithm to finish on graphs for more sophisticated programs or on graphs with more complex data flow optimizations may take months, years and even centuries. However, we may obtain sub-optimal coloring in more decent amount of time by implementing heuristics algorithm.

The first heuristic algorithm uses simple greedy coloring. It goes from the first vertex to the last and tries to color it in the first available color. If there is no such color it adds vertex to the list for later processing. At the end, all uncolored vertices are colored in the color that would give the minimum interference (sum of edges weights) with its neighbors.

In this algorithm there are two main places that affect the quality of the final coloring. The first one is the initial order in which the vertices are considered by the greedy algorithm. The second one is the order in which remaining uncolored vertices are considered by the final coloring step.

We have tested several possible heuristics for determining these orders and in the end considering vertices with the highest sum of interferences first proved to give the best overall results. However, we were able to construct some pathological test cases where this heuristic performed suboptimally, even though they do not exhibit patterns usually found in practice.

4.2.3 Chordal coloring

Another heuristic graph coloring algorithm is Chordal coloring. Chordal coloring is optimal for a specific subset of all graphs called chordal graphs and sub-optimal for other graphs. The main idea of chordal coloring is to construct a special ordering of vertices known as perfect elimination ordering (PEO) and to run greedy coloring algorithm on that order.

A chordal graph is one in which all cycles of four or more vertices have a chord, which is an edge that is not part of the cycle but connects two vertices of the cycle. In other words, every induced cycle in the graph should have exactly three vertices. Chordal graphs are a subset of the perfect graphs.

This algorithm was chosen with false assumption that IGs for static data, like IGs of programs in strict SSA form, are chordal [4]. Later, however, it was proven to be able to find quite close to optimal colorings for graphs produced by some languages [18]. We have reasons to believe that IGs where for each state of a variable exists a separate vertex in the graph (like in IGs for strict SSA-formed programs) will be chordal.

In our implementation, the chordal coloring algorithm consists of the following parts:

1. (optional) graph ”chordalization”
2. clique elimination
3. coloring the graph
4. coloring the previously eliminated vertices
5. permuting the colors to fit into memory constraints

We will describe each part, estimate their running time and outline weak points if such are present.

Graph chordalization Also known as minimal triangulation of a graph, this step guarantees that other steps of algorithm will be dealing with chordal graph and thus coloring will be optimal on the new graph. It would also allow us to easily estimate penalty acquired due to clique elimination and heuristic coloring of the eliminated vertices, as these steps also assume that the input graph is chordal.

Algorithm used to find minimal triangulation is called MCS-M, its correctness was proven, and its running time was estimated in [14]. In our implementation the MCS-M algorithm has $\mathcal{O}(N \cdot M \cdot \log N)$ running time, using Dijkstra-like approach to find maximum weight of a vertex occurring along the way from u to v for all (u, v) in E, which is required in MCS-M to construct a minimal elimination ordering.

However, adding minimal triangulation step to chordal coloring algorithm improved the average quality of coloring very insignificantly, but in exchange it increased the average running time of an algorithm by more than 4 times, being the most time-consuming step in chordal coloring algorithm. Therefore, it was decided to leave this step optional until further investigation.

Clique elimination From this moment on, we will assume that the input graph is chordal for all chordal coloring steps, starting from this one. If it is ”more or less chordal”, then the results won't differ much from optimal; if it is ”very much not chordal”, we will apply chordalization beforehand.

Chromatic number of chordal graphs equals to the size of largest clique, since they are perfect. Therefore, one possible way to decrease chromatic number of chordal graph (thus allowing the data this graph represents to fit into existing memory banks) is to eliminate all cliques of size bigger than the number of available colors. We can find such cliques in linear time by the property of PEO, following directly from its definition: for all vertices v_i in PEO, a sub-graph formed from vertices $Neighborhood(v_i) \cap \{v_1, v_2, ..., v_i\}$ is complete.

The only question is: how do we select vertices to eliminate? Currently we use heuristic based on the sum of weights of edges incident to the vertex divided by number of colors. The higher it is, the higher the cost to eliminate it.

The next step, graph coloring, will be dealing with graph whose cliques have been reduced to have $\chi(G) = \omega(G)$. In following steps, however, the previously eliminated vertices will be returned to the graph.

Coloring the graph At this step we have a chordal graph, which chromatic number is guaranteed to be less than or equal to the number of available colors

By constructing PEO and passing it in reversed order to greedy coloring algorithm, we will obtain optimal coloring of a graph reduced on the previous step.

Coloring the previously eliminated vertices After we've optimally colored vertices we could color using no more than the number of colors available, we need to color the rest of the graph.

Like clique elimination, this step is done in heuristics fashion: for each yet uncolored vertex v, we explore its neighborhood, and for each color k we find out what penalty would v introduce if it were colored with k. We find such k so that v would introduce the minimum penalty possible; color v with k, and move to the next uncolored vertex.

Permuting the colors to fit into memory constraints The previous steps searched for the optimal graph coloring. Here we transform this coloring so that it would fit into the given memory constraints.

Currently, we do it by permuting colors: for each color i from 1 to k, we check if we would fit into memories if we substituted it with some other color. Thus, running time is $\mathcal{O}(k^k)$, which is tolerable as there are usually not that many independent memories present on a real hardware.

However, the permutation algorithm itself introduces a bottleneck for coloring, because of its last position in algorithms chain, where it accepts the result of previous steps unable to affect their output. This is a serious issue now, and we are willing to find another way to check memory constraints in chordal coloring. Constraints checking in exhaustive coloring is integrated with coloring search, which allows the colorer to skip the checks for colorings that does not fit into the memory. Ideally, we want this kind of integration for constraints checking in chordal coloring algorithm as well.

5 Evaluation

Implemented solution was evaluated on the number of sample tests as well as a real-life application that implements an algorithm for detecting objects in an image using a neural network trained using the CIFAR-10 data set.

The quality of the assignment was measured in several ways. When the source code for the original program was available, we used the number of CPU cycles spent executing it with specific data assignment as a metric. The tests were run on the processor in DesignWare ARC EM DSP Family. The number of CPU cycles was measured using supplied cycle-accurate simulator and the results were compared against some base data mapping (usually when all data resides in the single generic memory).

In other cases, such as randomly generated Interference Graphs, instead of comparing application performance we calculated the mapping's penalty metric and compared it between implemented algorithms. The penalty of the mapping is the cumulative cost of all edges between interfering vertices, that is vertices assigned to different memories.

The quality of the algorithms was tested on several memory configurations:

1. generic memory with small load penalty and one averagely sized memory bank
2. generic memory with big load penalty and three small memory banks
3. generic memory with small penalty and three memory banks of unlimited size
4. generic memory with small penalty and four memory banks of unlimited size
5. generic memory with average penalty and two big memory banks

For each memory configuration we collected the penalty difference of each algorithm with the optimal coloring found by the exhaustive search and

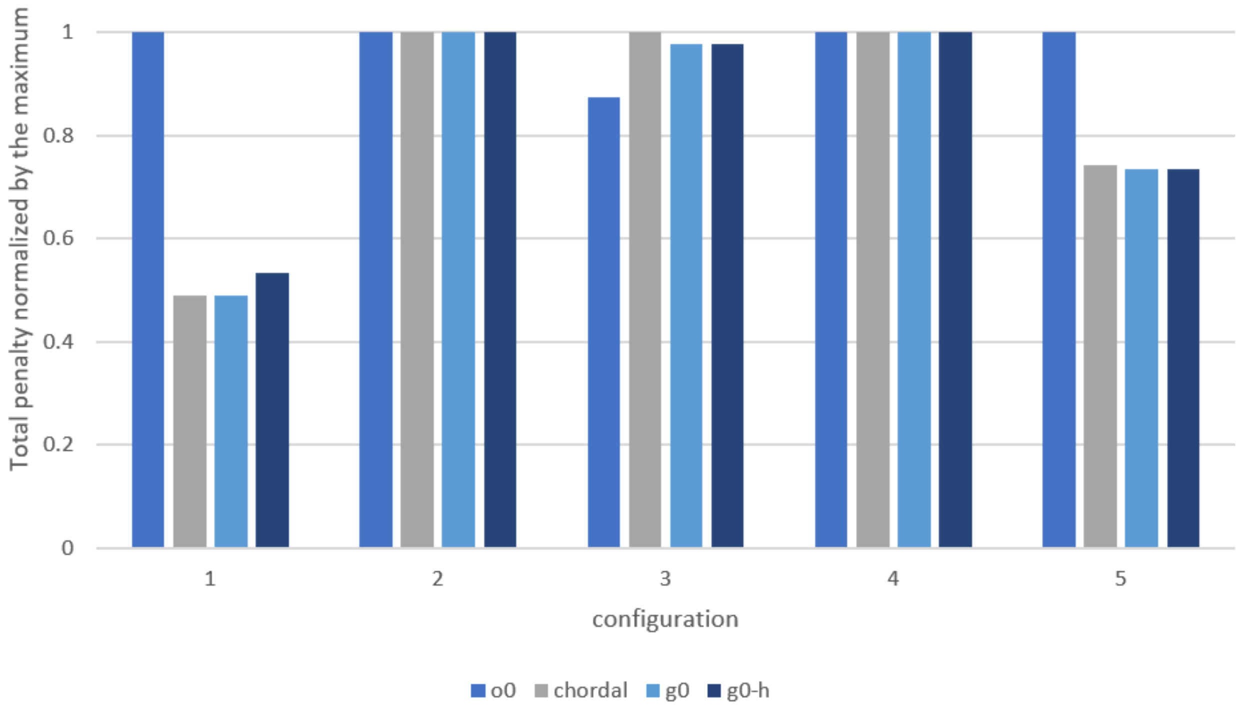

Figure 5: Total normalized penalty for different memory configurations (lower is better)

divided it by the maximum total penalty among tested algorithms. The summary results are shown on figure 5, where the lower the value is the better an algorithm performed for the given memory configuration. In this and the following figures exhaustive is an exhaustive branch and bound algorithm, o0 is a base greedy algorithm, and g0 is a greedy algorithm with applied heuristics.

g0-h is a special version of g0 coloring algorithm that we come up with after measuring coloring quality on randomly generated graphs. Regular coloring algorithms do not take into account the usage of the variable outside interference edges, so they might produce suboptimal coloring when there are vertices with high usage (weight) and memory configuration includes memory with a noticeable load penalty.

One such example is the fifth configuration. For this memory configuration figure 6 shows the penalty results of algorithms on randomly generated Interference Graphs while fig. 7 shows the results for graphs based on real-life programs. Figure 6 shows that g0 performed especially badly for randomly generated tests number 12 and 13. These were the tests with an

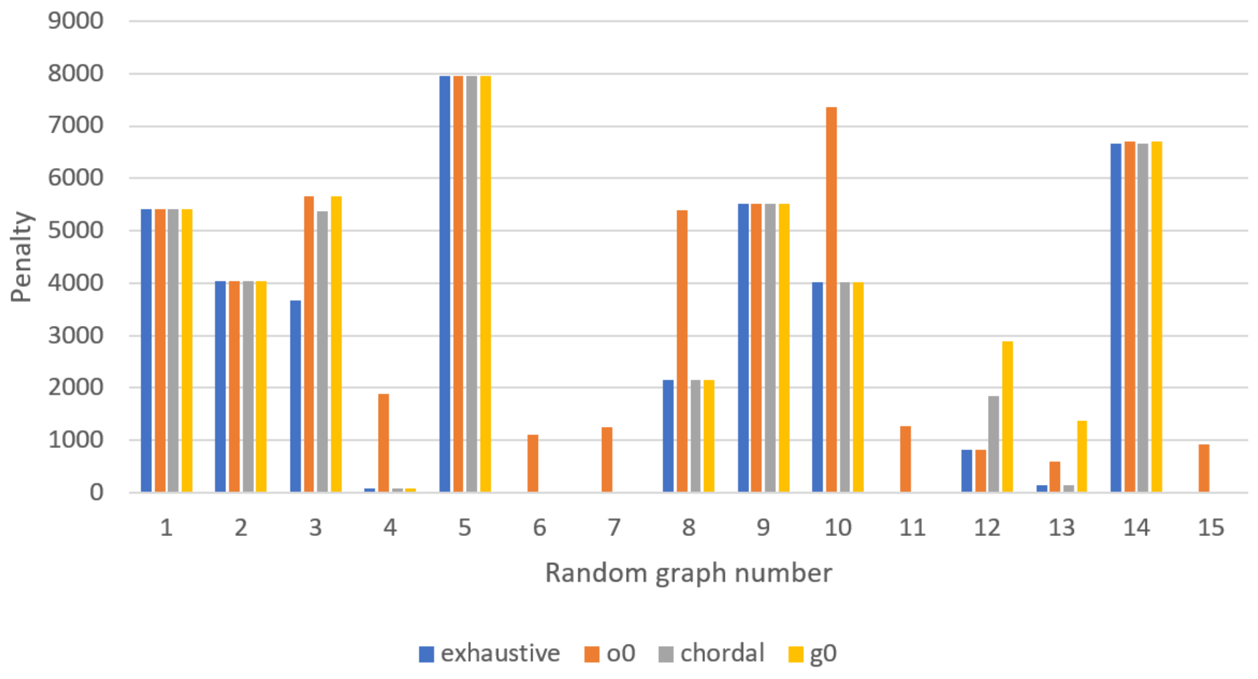

Figure 6: Total penalty on random graphs

unusual pattern of high usage of individual vertices. To better handle such graphs with modified g0 algorithm by adding special auxiliary vertex for each vertex with non-zero usage.

While addition of clever heuristics can noticeably improve the coloring quality, greedy algorithms still often lose to the optimal result of the exhaustive branch and bound algorithm. This is especially true for the cases where graph has a chromatic number significantly higher than the number of available memory banks. However, the time required to color the Interference Graph exhaustively grows quickly with the number of vertices and edges of the graph as shown on fig. 8 which makes it impractical for all but simple programs. Greedy algorithms, on the other hand, do not exhibit such problem and all execute in the fraction of the second. This allows to even apply all implemented greedy algorithms (like g0, g0-h and chordal) and choose the best result among them with no noticeable speed hit.

Finally, we took an real word application that implemented an algorithm for detecting objects in an image using a neural network trained using the CIFAR-10 data set and generated all possible data mappings for it, i.e. for each global variable used in the program we considered all possible memories available on the target ARC EM processor that are close to the

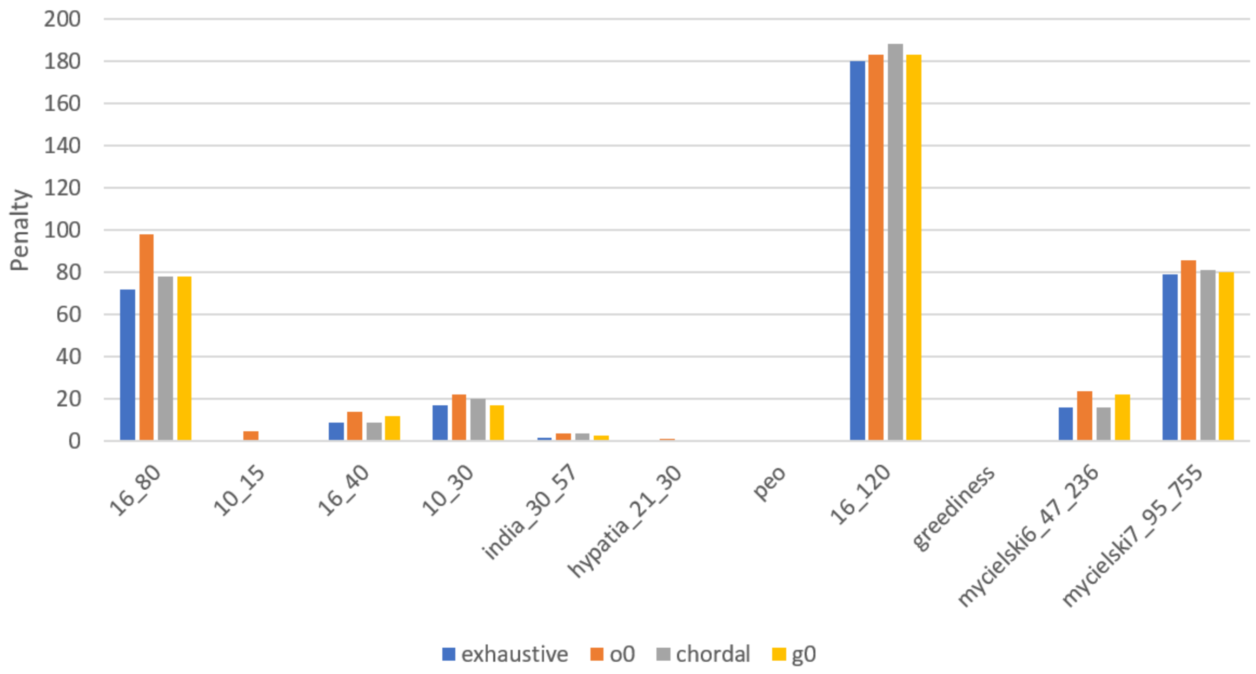

Figure 7: Total penalty on graphs based on real software

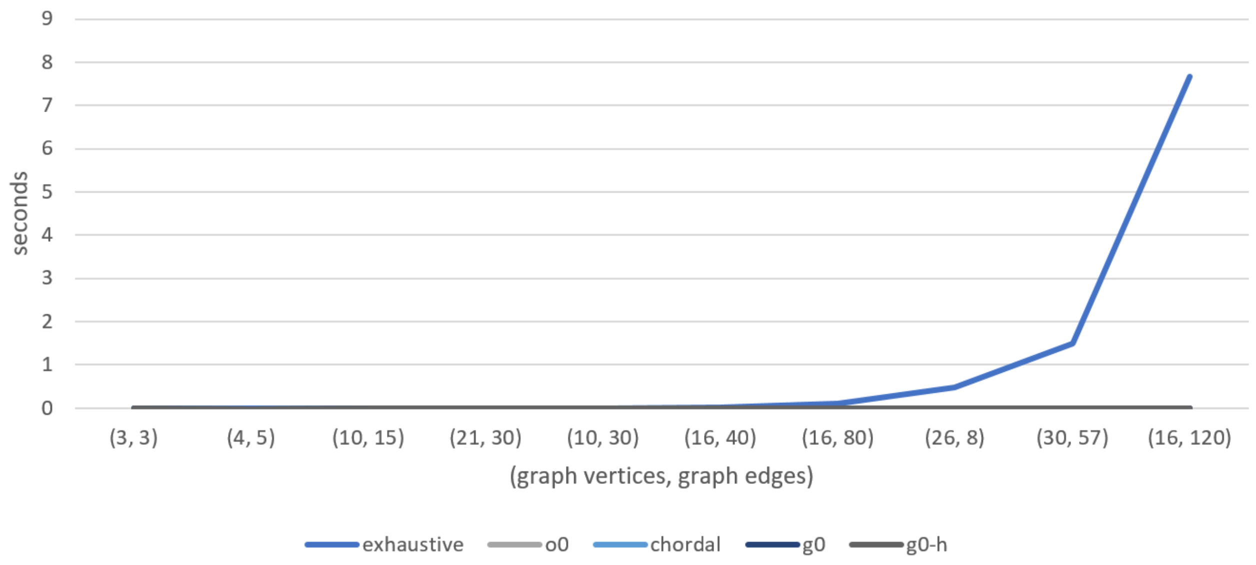

Figure 8: Algorithm scaling with the size of the graph

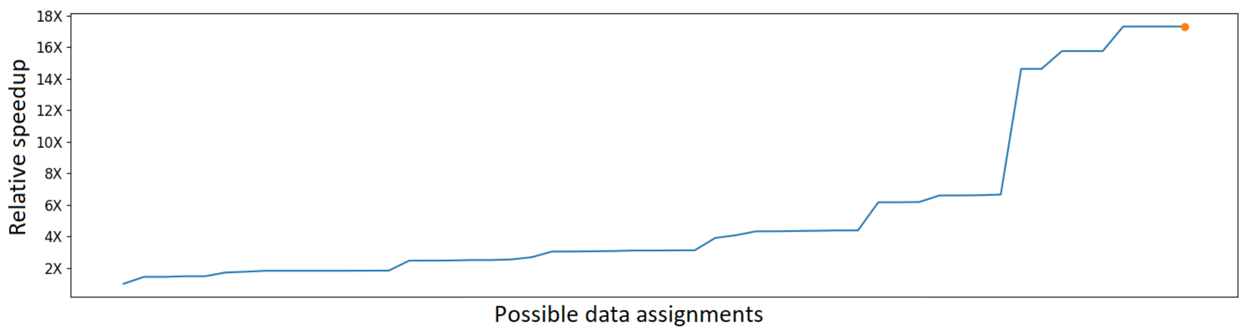

Figure 9: CIFAR10 performance characteristics based on chosen data mapping

fifth memory configuration described above. We then compiled the program with each produced data mapping and measured the cycle counts using the cycle-accurate simulator. The results can be seen on fig. 9, which shows the speed up of each possible data mapping relative to the slowest one ordered from slowest to the fastest. It demonstrates that choosing the right data mapping is very important as it can lead to the performance difference up to the factor of 16 or more. For this application both the chordal and g0 algorithms were able to find the optimal data mapping.

Conclusion

In this work we achieved the following results:

- existing approaches to similar problems were reviewed and compared as well as the algorithms used in them;
- a solution architecture that provides flexibility in the choice of data assignment algorithm was designed;
- a set of automation tools following the proposed design was implemented with the help of the LLVM framework;
- the developed solution was evaluated on a number of test programs, including those implementing an algorithm for detecting objects in an image using a neural network trained using the CIFAR-10 data set.

Using the developed solution, one can quickly find the optimal memory assignment for the given program and memory configuration as well as choose the best suited memory configuration for the given task from the set of available configurations.

Furthermore, the chosen design allows to easily implement and test new ideas for the data assignment algorithms. We have implemented and evaluated several such algorithms, including slow but precise exhaustive branch-and-bound algorithm, fast greedy coloring algorithm based on heuristics and fast coloring algorithm based on the idea of graph's chordality.

There are several directions for future research and development.

One area to improve is the quality of the generated Interference Graph. Currently, it depends on the quality of the pointer analysis which is rather basic. For example, it does not consider the probability of the branches taken and does not handle recursion. There are ways to improve on it by using more sophisticated pointer analysis algorithms such as those presented in the [7] and [10]. Another approach is to delegate program's data flow analysis to the appropriate libraries, such as [27].

Other direction for further improvement is the data mapping algorithms themselves. Apart from exhaustive branch-and-bound, the current algorithms are all based on the idea of coloring the graph in some specific order. However, it is possible to use different approaches to the problem of data assignment such as ILP, Genetic Programming and PBQP [6]. Due to the fact that in the end it all comes down to the mapping of data to memory banks, all these approaches can still be implemented in the terms of graph coloring interface used by Allocator.

References

[1] Briggs Preston, Cooper Keith D., Torczon Linda. Improvements to Graph Coloring Register Allocation // ACM Trans. Program. Lang. Syst.

[2] Chaitin G. J. Register Allocation & Spilling via Graph Coloring // Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction. — SIGPLAN '82. — New York, NY, USA : ACM, 1982. — P. 98–105. — URL: `http://doi.acm.org/10.1145/800230.806984`.

[3] Gansner Emden R., Koutsofios Eleftherios, North Stephen. Drawing Graphs with dot. — 2015. — 01. — URL: `https://www.graphviz.org/pdf/dotguide.pdf`.

[4] Interference graphs of programs in SSA-form : Technical Report : ISSN 1432-7864 / Universitat Karlsruhe ; Executor: Sebastian Hack : 2005.

[5] Hack Sebastian, Grund Daniel, Goos Gerhard. Register Allocation for Programs in SSA-Form // Compiler Construction / Ed. by Alan Mycroft, Andreas Zeller. — Berlin, Heidelberg : Springer Berlin Heidelberg, 2006. — P. 247–262.

[6] Hames Lang, Scholz Bernhard. Nearly Optimal Register Allocation with PBQP // Modular Programming Languages / Ed. by David E. Lightfoot, Clemens Szyperski. — Berlin, Heidelberg : Springer Berlin Heidelberg, 2006. — P. 346–361.

[7] Hardekopf Ben, Lin Calvin. The Ant and the Grasshopper: Fast and Accurate Pointer Analysis for Millions of Lines of Code // SIGPLAN Not.

[8] Instruments Texas. TMS320C54x DSP CPU and Peripherals Reference Set, volume 1 edition. — 2001.

[9] Instruments Texas. TMS320C55x DSP CPU Reference Guide. — 2001.

[10] Jia Chen. Andersen's pointer analysis // GitHub. — 2018. — URL: `https://github.com/grievejia/andersen` (online; accessed: 05.11.2018).

[11] Ko Ming-Yung, Bhattacharyya Shuvra S. Partitioning for DSP Software Synthesis // Software and Compilers for Embedded Systems / Ed. by Andreas Krall. — Berlin, Heidelberg : Springer Berlin Heidelberg, 2003. — P. 344–358.

[12] Lee E. A., Messerschmitt D. G. Synchronous data flow // Proceedings of the IEEE. — 1987. — Sep. — Vol. 75, no. 9. — P. 1235–1245.

[13] Leupers R., Kotte D. Variable partitioning for dual memory bank DSPs // 2001 IEEE International Conference on Acoustics, Speech, and Signal Processing. Proceedings (Cat. No.01CH37221). — Vol. 2. — 2001. — P. 1121–1124 vol.2.

[14] Maximum Cardinality Search for Computing Minimal Triangulations of Graphs / Anne Berry, Jean R. S. Blair, Pinar Heggernes, Barry W. Peyton // Algorithmica. — 2004. — Aug. — Vol. 39, no. 4. — P. 287–298. — URL: `https://doi.org/10.1007/s00453-004-1084-3`.

[15] Minimising Access Conflicts on Shared Multi-Bank Memory / Andreas Tretter, Georgia Giannopoulou, Matthias Baer, Lothar Thiele // ACM Transactions on Embedded Computing Systems. — 2017. — 09. — Vol. 16. — P. 1–20.

[16] Murray Alastair, Franke Björn. Adaptive Source-Level Data Assignment to Dual Memory Banks // ACM Transactions in Embedded Computing Systems - TECS. — 2012. — 06. — Vol. 11. — P. 1–22.

[17] Oshana R. DSP Software Development Techniques for Embedded and Real-Time Systems // Embedded computer systems. — 2006. — 01.

[18] Pereira Fernando Magno Quintão, Palsberg Jens. Register Allocation Via Coloring of Chordal Graphs // Programming Languages and Sys-

tems / Ed. by Kwangkeun Yi. — Berlin, Heidelberg : Springer Berlin Heidelberg, 2005. — P. 315–329.

[19] Performance Evaluation of Homogeneous and Heterogeneous Architectures in Real-Time Signal Processing and Control / M.O. Tokhi, M.A. Hossain, M.J. Baxter, P.J. Fleming // IFAC Proceedings Volumes. — 1995. — Vol. 28, no. 5. — P. 537 – 542. — 3rd IFAC/IFIP Workshop on Algorithms and Architectures for Real-Time Control 1995 (AARTC'95), Ostend, Belgium, 31 May-2 June 1995. URL: `http://www.sciencedirect.com/science/article/pii/S1474667017472776`.

[20] Pieter van der Wolf. Processor Solutions for Energy-Efficient IoT Applications // MPSoC Conference. — 2017. — URL: `http://www.mpsoc-forum.org/previous/2017/files/proceedings/Pieter_VanDerWolf.pdf` (online; accessed: 05.02.2019).

[21] Programming languages – C – Extensions to support embedded processors : ISO/IEC TR 18037 : 2008.

[22] Register Allocation: What Does the NP-Completeness Proof of Chaitin et al. Really Prove? Or Revisiting Register Allocation: Why and How / Florent Bouchez, Alain Darte, Christophe Guillon, Fabrice Rastello // Languages and Compilers for Parallel Computing / Ed. by George Almási, Călin Caşcaval, Peng Wu. — Berlin, Heidelberg : Springer Berlin Heidelberg, 2007. — P. 283–298.

[23] Saghir Mazen A. R., Chow Paul, Lee Corinna G. Exploiting Dual Data-memory Banks in Digital Signal Processors // Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems. — ASPLOS VII. — New York, NY, USA : ACM, 1996. — P. 234–243. — URL: `http://doi.acm.org/10.1145/237090.237193`.

[24] Sipkova Viera. Efficient Variable Allocation to Dual Memory Banks of DSPs // Software and Compilers for Embedded Systems: 7th Inter-

national Workshop, SCOPES 2003, Vienna, Austria, September 24-26, 2003. Proceedings / Ed. by Andreas Krall. — Berlin, Heidelberg : Springer Berlin Heidelberg, 2003. — P. 359–372. — ISBN: 978-3-540-39920-9. — URL: `https://doi.org/10.1007/978-3-540-39920-9_25`.

[25] Stockmeyer Larry. Planar 3-colorability is NP-complete // ACM SIGACT News. — 1973. — 07. — Vol. 5. — P. 19–25.

[26] Time-critical computing on a single-chip massively parallel processor / Benoît Dinechin, Duco van Amstel, Marc Poulhies, Guillaume Lager. — 2014. — 03. — P. 1–6.

[27] Yulei Sui. SVF:Pointer Analysis for C and C++ // GitHub. — 2018. — URL: `http://svf-tools.github.io/SVF/` (online; accessed: 05.05.2019).