DEGREE PROJECT IN COMPUTER SCIENCE AND ENGINEERING,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2016

Improving performance on base stations by improving spatial locality in caches

JONAS CARLSSON

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF COMPUTER SCIENCE AND COMMUNICATION

KTH Computer Science
and Communication

Improving performance on base stations by improving spatial locality in caches

JONAS CARLSSON
JCARLSSO@KTH.SE

School of Computer Science and Communication (CSC)
Master Thesis in Computer Science
Supervisor: Per Austrin
Examiner: Johan Håstad
Done at Ericsson DURA

Abstract

For real-time systems like base stations there are time constraints for them to operate smoothly. This means that things like caches which brings stochastic variables will most likely not be able to be added. Ericsson however want to add caches both for the possible performance gains but also for the automatic loading of functions. As it stands, Ericsson can only use direct mapped caches and the chance for cache misses on the base stations is large. We have tried to see if randomness can be decreased by placing code in the common memory. The new placement is based on logs from earlier runs. There are two different heuristic approaches to do this. The first was developed by Pettis & Hansen and the second was developed by Gloy & Smith. We also discuss a third alternative by Hashemi, Kaeli & Calder (HKC) which was not tested. However the results show there are no practical improvements by using code placement strategies.

Referat

Förbättra prestanda på basstationer genom att öka rumslokaliteten i cachen

Realtidssystem som basstationer är väldigt tidskritiska, vilket betyder att exempelvis cacher, som har en hög varierbarhet, undviks. Ericsson vill trots det lägga till cache, delvis för den prestanda den ger, men också för att automatisera laddning av kod. Dock så kan de endast utnyttja en direkt mappad cache, vilken har en otroligt hög risk för cachemissar. Vi försöker minska risken för cachemissar genom att placera kod i huvudminnet. Den nya placeringen baseras på loggar från tidigare körningar. För att uppnå detta har vi använt två olika algoritmer. Den första algoritmen är utvecklad av Pettis och Hansen och den andra av Gloy och Smith. Vi beskriver en tredje algoritm utvecklad av Hashemi, Kaeli och Calder (HKC) men den implementerades aldrig. Resultaten från våra tester visar att det inte finns några praktiska fördelar att använda kodplacering.

Contents

Chapter 1

Introduction

1.1 Caches

The algorithms discussed in this report are based on the problems found with caches. A cache is a volatile data storage often separated from the main storage unit with a faster access time, for example the RAM in home computers. The cache can always be used where some data is to be used multiple times. For example say there are two parties coming up which both require the use of a suit. Your nicest suit is stored in a special bag in your attic and retrieving and storing it takes a considerable amount of time. However, you know you will have to use it again soon after the first party. So instead of putting it back in the attic you store it temporarily in your closet which is a lot easier and faster to access. Talking about computers, the attic is the main or common memory which is slower but larger and the closet is the cache which is smaller but has a significantly faster access time. However, a difference between a computer and the suit allegory is that the data is never physically moved just copied from the common memory to the cache. For simplicity, we will always measure the cache size in number of cache lines instead of bytes of memory.

1.1.1 Associativity

Whenever an item cannot be found in the cache we must go to the common memory to get it. This is called a cache miss and these are the largest contributors to the randomness and slowdowns of the caches. For large programs, when and where one of these cache misses will occur is incredibly hard to determine, as the size of the cache is much smaller than the common memory, so there will be some kind of swapping procedure to remove unused items. There are a few algorithms used for the swap procedure. They go by the name N-associative where N is a number between one and the number of cache lines in the cache. The value of N represents how many cache lines will map against a chunk in the common memory. Even though we are talking about a theoretical infinite amount, there are in general 3 different forms. These are the direct mapped, fully associative and the N-way associative.

The direct mapped means that for each block in the common memory there is one line in the cache. This can also be called the one-way associative cache. This means there are many common memory lines which will be mapped toward the same cache line. If two or more of those lines are used very close to each other then there will be conflicts. If they are used one after another they will overwrite the previous information. This behaviour is called thrashing. Thrashing is exceptionally bad as for each time the cache is asked it will generate a cache miss.

The fully associative cache means that each block of common memory can be mapped to any cache line. If the cache runs a Least Recently Used strategy (LRU), which is a queue based system that can be implemented by using a priority queue. Whenever a cache line is used it will moved to the front of the queue and when the max amount of cache lines is reached then the lines at the end will be removed.

The last form is the N-way associative and this makes up the rest of the swapping algorithms. As the N in the name might suggest, this is not a single algorithm but all of them are very similar to each other. Like the fully associative cache this maps each block of common memory to N lines in the cache memory. For a two-way associative cache, each common memory block will be mapped to two lines in the cache. This reduces the chance of cache lines used close to each other of overwriting each other. Higher associativity leads to decreased chance of conflict. However, it is incredibly hard to predict how much the chance of conflict decreases as the size of the program and the actual displacement in the cache and common memory all affect the effectiveness of the cache. The rule of thumb however, is that doubling the associativity from direct mapped to two-way has about the same effect on the amount of cache misses as doubling the cache size. The same goes for going from two-way to four-way but increasing the associativity further generally gives smaller effects [19].

1.1.2 Spatial/Temporal Locality

When discussing cache efficiency there are two very important concepts, temporal locality and spatial locality. These are the fundamental ideas behind how the cache operates and how it is built. Performance can be greatly increased if the programs are structured to use these localities efficiently. Temporal locality says that if one memory area is used it is likely that it will be used again soon. This is helped by the associativity of the cache, making the data stay for longer periods of time.

Spatial locality is if one memory area is used then an area close by is going to be used in the near future. How much area that should be prepared is always a bit of a guess as this depends on several factors like the average size of functions. Spatial locality can also be improved by increasing the likelihood that one of the areas around the called memory area contain useful data.

1.2 Motivation

All mobile infrastructures rely on base stations to receive any form of data. These base stations contain different subsystems like analog radio processing, baseband processing and data packet processing. The baseband is a real-time system with very high processing intensity and has a direct impact on the overall product performance.

In baseband development there has been a ban on cache memories for a very long time. The reasoning behind it is the source of randomness it adds to execution times. For example, whenever a function is designed that functions time budget is set. A function might have a budget of $10ns$, which the developer can achieve during testing. When testing a function it is usually run in a mock environment which is smaller than the real application. However when the function gets patched in to the full application it runs in $30ns$. This happens as when we are working in the mock environment each called function is exchanged by a dummy function. Given that these functions only return a preselected value they are very small. This makes the test application very small and the effect it has on the cache is minimal, which means there will rarely be cache misses. While the whole application is large enough to take up the whole cache several times over. This makes cache misses a lot more common. This Master thesis aims to increase the predictability of execution time by placing code in the common memory to increase the cache locality. With common memory we mean both the code and data memory.

1.3 Problem Statement

Caches have an inherent stochastic element because the state of the cache can be widely different for individual procedure calls. Given that the cache state can differ by large amounts there can be huge variances in execution times. For all projects with real time constraints, like the mobile phone base stations, these variations can make projects abandon the use of caches. These variations occur because of the order in which the functions are run and the placement of said functions in common memory. In this project we aim to see if using heuristics for placing code in memory can decrease the randomness of the cache while still keeping the benefits. This will be done by decreasing the amount of cache misses. The main concept we are trying to improve upon is the spatial locality (Section 1.1.2). There are many different heuristics that aim to improve spatial locality and we will investigate two of them in this thesis and discuss a third. The ones we will investigate is created by Pettis & Hansen (Section 3.2) and Gloy & Smith (Section 3.3). The third algorithm which we discuss is created by Hashemi, Kaeli & Calder (Section 3.4). We will be working with a direct mapped cache since if improvements can be made then the direct mapped cache will show the largest results, see Section 1.1 for a short discussion on cache models. This thesis only aims to improve the use of the instruction cache in this specific application.

1.4 What to Measure

Finding the number of cache misses would be the perfect way to measure the efficiency of the chosen heuristics, since this is the measure the heuristics mentioned above try to optimize. This however can not be done on the machine we are running the tests on, as it does not count the amount of cache misses. Instead we will compare the histograms over execution times for each function and also compare the arithmetic mean for the same histograms. Given that we are trying to decrease randomness and not only the speed we also want to see what effects our heuristics have on the variance, so we also calculate the standard deviation. It is in the very nature of our problem that there will be multiple peaks also called bimodal distributions. One of these peaks will be the calls to the function when there was a cache hit, while the other peak will be the function calls to when it is a cache miss. However, for bimodal distributions the standard deviation will not give an accurate picture on the changes.

1.4.1 Bimodal Distribution and Standard Deviation

When looking at unimodal distribution it is known that if some changes cause the standard deviation to increase the peak becomes larger and vice versa. This cannot be said for bimodal distribution. Say that we have a distribution where the data is located at two distinct x-values in other words we have two high peaks in the distribution curve. If most of the mass from the first peak is moved to the second peak the standard deviation would become smaller and be more focused on that peak. For our purposes even though the standard deviation becomes a lot smaller it is a worse result than the equally large as this would mean the runtime has increased. We can however see if the results are actually better or worse by also investigating the arithmetic mean. If the mass moves, so does the arithmetic mean, which means that if the standard deviation decreases and the mean increases we know that that function is now slower. However, if the mean decreases we know that this function is faster.

1.5 Method

This project starts with taking a log (Section 2.1) from an earlier run, which will be the basis for the heuristics. The heuristics build a Call Graph (Section 2.2) from said log. The Pettis & Hansen and Gloy & Smith have their own strategy for generating a call graph and the third uses the same as Pettis & Hansen. From the call graph we generate an ordered list which is used to generate the final ordering. Finally to test the new ordering we swap in the generated LCF (Linker Control File) and relink all files. We create a new log from this run which will be used for analyzing the data to see if any improvements have been found.

1.6 Overview

This section gives a quick overview of this Master thesis. The two main heuristics implemented here are Pettis & Hansen (Section 3.2) and Gloy & Smith (Section 3.3). There is also a third alternative described here called HKC (Section 3.4) developed by Hashemi, Kaeli & Calder. Due to time constraints the HKC heuristic was not tried in practice. Each algorithm is built up with first a general explanation of the algorithm followed by a time complexity analysis. This in turn is followed by an example run which uses the same base call graph for each algorithm which is detailed in Section 3.1. After this we discuss the methodology for our tests in Chapter 4. We end by presenting the results (Chapter 5) and our conclusions (Chapter 6).

Chapter 2

Concepts used in this thesis

2.1 The Call Log

The log is built up by generating a log event on both entering and exiting of functions. This log shows us how the call stack changed during the lifetime of that specific log. Whenever the number of function calls is discussed, it is only the number of enter statements we refer to. There is also a timestamp for each event which is our main way of estimating the time taken for each function. In Figure 2.1 there is an example of what a log might look like.

```
Example of a log:
enter:Main
enter:Calc
exit:Calc
enter:Print
exit:Print
exit:Main
```

Figure 2.1: Example of a simple log file showing what the enter and exit of different functions might look like.

2.2 Dependency Graph

All of our heuristics use, in some way or another, a dependency graph to represent how different functions affect each other. There are a few different varieties like Pettis & Hansen's Weighted Call Graph (WCG) and Gloy & Smiths Temporal Relations Graph (TRG). This graph is an undirected graph as we assume that it is of no interest which function calls which, as we can see both ends of the edge as equally important.

In Figure 2.1 there is an example of a log from a simple program which helps us show the steps necessary for generating a Weighted Call Graph (WCG), which is the simplest dependency graph. The first task to do is to check each line. The first line is a special case as there is no function calling that, so an edge cannot get created yet. After that it enters the Calc function and now can assign an edge between Main and Calc, the new edge is assigned the weight of one. Next it exits Calc and return control to Main. Next it reads the next line which tells it that Main calls Print and add an edge between Main and Print. The following undirected graph Figure 2.2 is the WCG for the previous log. If it would have called Calc or Print once more before it exited main the corresponding edge would have added one to its weight. For example if it adds enter:Calc and exit:Calc before it exits Main it would have changed the weight on the Main-Calc edge to 2.

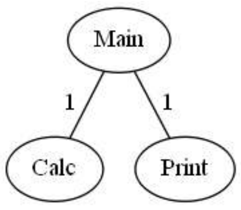


Figure 2.2: Example of a Call Graph given the log in Figure 2.1

2.3 Popular and Unpopular Procedures

When talking about both Gloy & Smith and HKC there exists the concept of popular and unpopular procedures. This was first proposed by HKC [5]. What this means is that all functions are divided into the two categories by how many times they are called. There have been no detailed discussions on good break off points for where the popular and unpopular divide should be. Although according to Calder et.al. [2], 90% of the execution time is accounted for by 10% to 30% of the program. It should be noted that the execution time of a function or procedure has no effect on the popularity. Even though a function may be running for a long time it might not get or do any calls to other functions, so it never alters the state of the cache. Instead what determines the popularity of a function is the total cost of all edges connected to that node in our dependency graph. If that cost is high then that function is either being called a lot or it calls a lot of other functions.

Chapter 3

Heuristics for improving spatial locality in caches

In this chapter we discuss the different heuristics in detail and an example which describes step by step how each heuristic finds an adequate solution to our problem. Like other heuristic and algorithmic problems the solutions can be split into smaller parts. This helps researchers focus on smaller parts of the problem and decrease the complexity of the problem. Our problem can be divided into two parts. The first part of the problem is converting the logs into a structure which allow us to make decisions around the placement. This is called a dependency graph where each node is a function and every edge shows how functions affect each other. Note that which function calls which is of no interest, only that one of the functions called the other. Pettis & Hansen generate the graph discussed in Section 2.2. Later this graph is exchanged for one that can represent more complex interactions of functions in the cache. This data is generated from profiled information from an earlier run.

The second part of the heuristic is the actual placement in the cache memory. This builds upon the information generated in the call graph. This information is used to place code so the spatial locality is maximized which is discussed in Section 1.1.2. That is whenever the cache pulls in a new function then the rest of the cache line should contain functions or procedures which have a relation to the first function. For example we know that A calls B very often so when A is called and subsequently read into the cache then B should also be read as it will most likely be called shortly after. All of the heuristics are trying to improve the spatial locality and they are trying to find a placement so functions close to each other during runtime is also close together in the common memory.

3.1 Example

To better show what differences exist between the heuristics discussed in this thesis, one call graph will be used to explain the placement, the example is shown in Figure 3.1. This is only done for the placement of functions as the differences between generating the call graphs are easier to see without an overarching example.

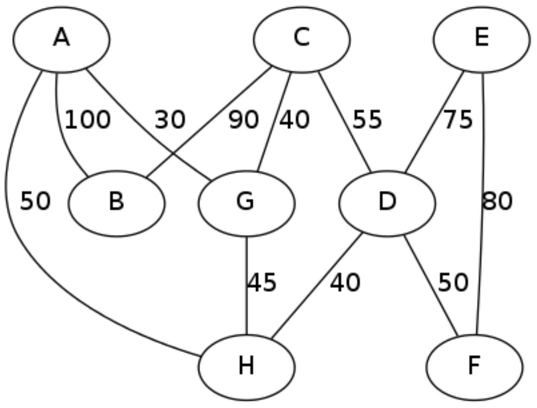
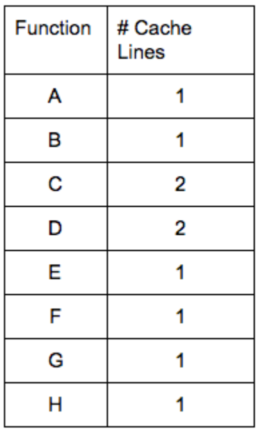


Figure 3.1: Base call graph example

For Pettis & Hansen this is sufficient information but both Gloy & Smith and HKC need more information. They need to know how large the functions are. These sizes are presented in Figure 3.2. For simplicity we only measure the size in number of cache lines. In this example the cache size is set to four cache lines.



Function	# Cache Lines
A	1
B	1
C	2
D	2
E	1
F	1
G	1
H	1

Figure 3.2: Sizes for functions in base example expressed in cache lines

3.2 Pettis & Hansen

Pettis & Hansen [14] were not the first to examine this problem, but they were the first to create a heuristic that went beyond simply grouping functions together.

The earlier attempts made by Hwu & Chang [8] and McFarling [12] both try to improve the cache performance by inlining functions into parent functions. The biggest difference from earlier work is the fact that Pettis & Hansen create a basis for their future choices by looking at a log from an earlier run. This means the heuristic by Pettis & Hansen is split up into two parts. The first step is creating a Weighted Call Graph (see Section 2.2) by analyzing the log which is described in Section 2.2. This graph is later used for the second part to create the linear placement.

3.2.1 Placement of Procedures

The placement procedure in Pettis & Hansen begins by picking the edge with the highest cost. This is used to combine the nodes of the edge into a compound node. This is continued by picking a new edge until all edges have been merged away. The final ordering can be found in the compound node left after all merging is completed.

To find the placement we start by finding the edge with the highest cost. The ends of this edge is used to find which nodes to merge into a compound node. All edges that where connected to the merged nodes must be redirected to the new compound node. If one node has edges to both of the merged nodes like in Figure 3.3a then the cost of both edges is added into the single edge of that compound node like in Figure 3.3b. An alternative way of looking at this is if there is no edge between two nodes, it only signifies an edge with cost zero. Then whenever we merge nodes the costs of all edges are added together.

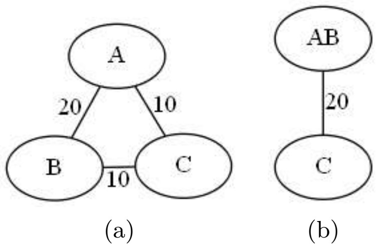


Figure 3.3: Figure 3.3a shows the how a graph can look before the merging phase. Figure 3.3b shows how that graph will have changed after the merging phase.

This is repeated until there are no edges left in the call graph. When going through all edges there will eventually be a case where two compound nodes have to be added together. For example if a compound node AB needs to be added to a compound node CD. What should the ordering of the original nodes be in the new compound node? There exists four different mappings being ABCD, ABDC, BACD and BADC. These possibilities are the only ones since the internal order is never changed. So AB will never be split to be able to insert CD in between like ACDB but the internals can be reversed. The original graph is used to find on which side the nodes should be placed. This is found by looking at the distance between the ending nodes. The distance is defined as the number of nodes on the shortest path. So the placement is found by looking at the shortest path between the ends of our compound nodes. This is best found by doing a Breadth First Search (BFS) from each end of the starting compound node and break as soon as it finds an end of the other node. Given how BFS goes through the graph it will find the shortest path.

When all merging is done there will be a list in the final compound node. This list shows us in which order all functions should be laid out. Do note that if there are several connected components in the call graph there would exist more than one node in the final graph. This is quite rare and all nodes are simply arbitrarily merged together when creating the final placement.

3.2.2 Time Complexity Analysis

Building the Weighted Call Graph

See Appendix A for the pseudo code. The time complexity for building our WCG is $O(L)$ where L is the number of function calls. This comes from the fact that there is only one loop and the most taxing operation in that loop is creating a node. This node can be created in constant time.

Placing Procedures

We define the boundaries for the number of nodes N as the number of unique functions in the log, and number of edges E as all entry lines in the log which is roughly equal to $L/2$.

The placement is built up by three main parts done in a loop. This main loop will at most run number of nodes (N) times as as in every step a node is merged. The first main operation of this loop is finding the edge with the highest cost, this will run in $O(E)$.

The second step is finding how the nodes should be placed together. This has some differences depending on if one or both nodes are compound. The heaviest case is if both nodes are compound. Here our heuristic must find which ends of the nodes are closest to each other, this is done by finding the distance between the ends. As discussed above this is done with a BFS and it has a worst time complexity of $O(E + N)$. This has to be done twice in the worst case so this step has a time complexity of $O(2(E + N)) = O(E + N)$.

The third step is merging two nodes. The first step of this is creating the new compound node. Then it adds all edges from the nodes removing them as it goes. E_i is the number of edges in node i. When merging the nodes i and j it takes $O(E_i + E_j)$. E_i can never be larger than the number of nodes in the graph. So this step can be simplified to $O(2N) = O(N)$.

To summarize, there is the main loop doing $O(N)$ turns and each loop will take $O(E+(E+N)+N)$. Which leads to $O(EN+N(E+N)+N^2) = O(EN+N^2+NE)$. Which gives us that the time complexity grows asymptotically to $O(EN + N^2)$.

3.2.3 Review of Example

We begin with the example detailed in Section 3.1. The absolutely first task to be done is to create a copy of the original graph. This will be used to answer placement questions later down the line. After the copy has been created it begins by finding the largest edge which is the first two nodes to be merged. In our example there is an edge between A and B with a cost of 100. After the first merging this edge is removed, and again look for the largest edge. This time it is the edge between AB and C with its cost of 90. Node G has an edge to both the compound node AB (created in the first step) and an edge towards C which is being merged into AB. When merging AB and C there is the question on which side node C should be put. In this case B is the closest node as it has the shortest path in our original graph so C is put at the side closest to B which creates the compound node ABC which is shown in Figure 3.4b. We continue merging the nodes with the largest

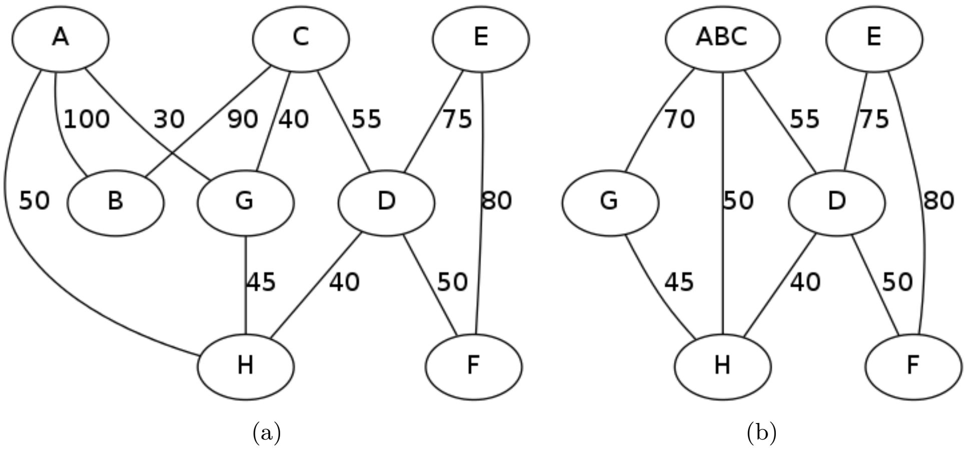


Figure 3.4: 3.4a shows the starting graph. 3.4b shows the graph after two steps into the merging process where the nodes A,B and C have been merged

edge together like with the ABC node. This means merging E and F which has no special requirements. After E and F is merged the edges E-D and F-D have also been merged together which means the largest edge is EF-D. This time we are left with a decision, on which side should D be placed? Should D be added next to E like DEF or next to F like EFD? This is determined by calculating the distance as discussed above. However in this case the distances from E to D and F to D is both one so the algorithm makes an arbitrary choice. This leaves us at Figure 3.5.

We continue investigating the largest edge which is now the edge ABC-G. Once again the original graph must be used to see on which side it should be placed. Here G is equally close to both A and C so it can be arbitrarily placed. In this example it is placed next to C. The next edge is between ABCG and H, where it is again equally close to both edges and it is placed next to A. This leaves us with one more problem before this example is done. When merging HABCG and DEF the same

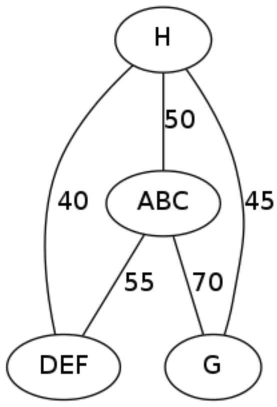


Figure 3.5: D,E and F nodes have been merged similarly to earlier

problem occurs again, on which side should the nodes be placed? With the two compound nodes, there are four possible orderings, ABCGH-DEF, ABCGH-FED, HGCBA-FED or HGCBA-DEF where H-D has a distance of 1, H-F has a distance of 2, A-F has a distance of 4 and A-D has a distance of 3. Again, the one with the shortest distance is chosen which is H-D. The final placement can be seen in Figure 3.6:

Figure 3.6: The final result, the order of this list will determine the ordering of the functions

3.3 Gloy & Smith

In a paper by N. Gloy and M.D. Smith [4], they try to improve upon the work done by previous researchers and most notably Pettis & Hansen [14]. This is done by simulating a fully associative cache while building the call graph. The resulting graph is called a Temporal Relations Graph (TRG), which conveys a better view of how that run utilizes the cache.

To further improve the results the researchers split all functions into basic blocks whose sizes have been statically determined. These blocks are called procedure chunks or chunks for short. These chunks are used to generate the TRG. This is one of the improvements the researchers use to generate a better placement, as each chunk will give more information on how two functions affect each other. For example, suppose that there are two functions split up into three chunks each, in the first function the third chunk collide with the second chunk in the second function. If we take the approach of Pettis & Hansen these function will collide but if the chunks are investigated with Gloy & Smith instead we see that they do not collide as different parts of the function are being used close to each other.

It is important to note that while we always talk about the chunks as a full cache line in our examples, this will not always be the case as functions vary in size. This means we will have to insert a gap so each function starts at a cache line.

Gloy & Smith also try to improve upon the placement of procedures. Although they are using the same kind of greedy merging like Pettis & Hansen, they add a step where they try to place the functions at different offsets so they have as few collisions as possible.

3.3.1 Building the TRG

As with the Pettis & Hansen heuristic this heuristic still starts with logged information from an earlier run. An important difference is that instead of using enter and exits of a function it uses the chunks. As mentioned in the previous section, a simulated fully associative cache is used during the graph building phase and as discussed in Section 1.1, a fully associative cache can be simulated with a priority queue.

Whenever a new log event is investigated the node connected to the function is added to the top of the cache, but this is not enough to update any of the edge weights connected to that node. If the new node already existed in the cache then the cost will be increased on all nodes between this node and the previous top node then the new node will be moved to the top. If the new node does not exist in the cache then no costs will be updated.

An example of this can be seen in Figure 3.7. When adding A to the cache shown in Figure 3.7a it will increase the edges A-B, A-C and A-D but not A-E because E occurs after the last A. After A is added to the cache the state will be updated to what is shown in 3.7b.

As can be seen in the example, we do not only get information on how function A will interact with the functions directly called by or those that call A. It also produces information on the other functions that are in the cache, which also can produce collisions.

Due to the fact that cache lines are mapped to different lines in the common memory they will be overwritten after some amount of time. Therefore there is no need to keep track of the whole call history in our simulated cache. The researchers have found that the simulated cache should have a fixed size, and they found from empirical testing that a size of double the actual cache size gives good results with not too much resource overhead.

3.3.2 Placement of Procedures

Part 1 - Popular, Unpopular Procedures

As was proposed by HKC [5] it starts by finding all unpopular and popular edges discussed in 2.3. This is mostly done for efficiency reasons. This is never discussed in any detail by the authors.

Simulated Cache
B
C
D
A
E

(a)

Simulated Cache
A
B
C
D
E

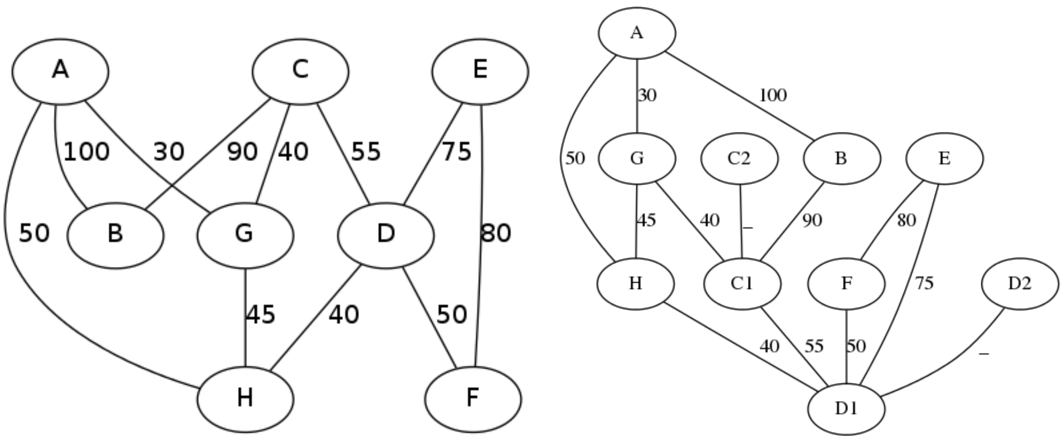
(b)

Figure 3.7: Example of the cache state before (a) and after (b) investigating a call to A.

Part 2 - Finding Nodes to Merge

For the actual placement of procedures there are two different call graphs. The first call graph is the same as the one discussed in the previous section. This graph is used to help find the specific placement for the procedures and it is called TRG_{place}, Fig 3.8b shows the base example version. However, the procedures cannot be split into smaller chunks in actual memory, so there need to be a graph that holds all temporal information on a procedure level instead of on chunks. This graph will help select the order in which the edges are investigated, this graph is called TRG_{select} which can be seen in Fig 3.8a. Note that this graph is the same as Fig 3.1.

In Fig 3.8b both chunks of C and D are connected to the same nodes but those edges have been removed so the figure is not too cluttered. Moreover, TRG_{select} will only be filled with the popular procedures.



(a) (b)

Figure 3.8: 3.8a shows the trg_{select} as it is in the start. 3.8b shows how the trg_{place} is built up when the algorithm starts, note that $C1$ and $C2$, $D1$ and $D2$ have the same edges but they are not shown to remove clutter

Noted the previously mentioned decisions about which edge will be merged is taken from TRG_{select}. The actual merging of the graph is the same as in Pettis & Hansen but the granularity has become finer as chunks are under investigation instead of functions. These relations between chunks is found by investigating TRG_{place}. The functions are never split, so when merging nodes the heuristic must find the position where each of the chunks contained in the two nodes is the smallest.

Part 3 - Combining Nodes

The placement is found by lining both nodes up so they start at the first line. Then one of the nodes is shifted down one cache line at a time, and the last cache line is moved back to the top.F aor each possible offset the total cost of that placement is calculated. The total cost is calculated by adding all weights between the chunks present in one cache line, and then summing over all cache lines. We are looking for a placement where the total cost is as small as possible. The offset discussed earlier is the most important metric from this part as it is used when generating the final ordering.

In Figure 3.9 is an example of how node A and compound node BC would be combined. For this example A will have an edge both to B and C. Figure 3.9a illustrates the first step of the comparison. To find if this position is actually good, the total weight is calculated for each cache line and then summed together. To be more specific, the cost for cache line one is A+B and cache line two and tree are ignored as there are no conflicts. The same is done in Figure 3.9b where a total cost of zero is found as there are no conflicts. In step three (Figure 3.9c) the end of the cache is hit so the node is wrapped around and begin again from the first cache line. This means in step three $A + C$ is calculated for the first line. The final placement for these two nodes is shown in Figure 3.9d. The final placement is the same as Figure 3.9b because it had the lowest total weight.

Cl	1	2
1	A	B
2		C
3		

(a)

Cl	1	2
1	A	
2		B
3		C

(b)

Cl	1	2
1	A	C
2		
3		B

(c)

Cl	1	2
1	A	
2	B	
3	C	

(d)

Figure 3.9: Example of how the merging process works in Gloy & Smith. Here Cl refers to which cache line is being investigated, and the columns show which node is being investigated.

Part 4 - Aligning Nodes for the Common Memory

Now there are one list for each cache line storing which functions that can be placed in that specific cache line. However, computer memory is linear so our data must be converted to one list. This is done by trying to place functions that are both the closest and have strong relations. In our model each function have an offset modulo the number of cache lines. Formula 3.1 is our heuristic to find the resulting list. This formula will try to find an ordering so that the number of empty areas in the final linear array is minimized while still maintaining the closeness found from the earlier parts. Formula 3.1 uses the starting point of the first function which is called p and the beginning of the second procedure which is called q and N is the total number of cache lines then we search for a placement minimizing LG which is the number of empty lines between functions.

When placing all functions it is preferable if the amount of functions that start in the middle of a cache line is minimized. As if they start at the end of a line and continue on over the next one both will have to be read which creates a larger time penalty. This can be avoided for most functions if all functions start at a new cache line. This is achieved by using Formula 3.2 which will calculate how big of a gap is needed between two functions so each function begins at a cache line. This step can be skipped if we only care about first level cache lines, if storage space is limited we can pick functions so this gap is minimized. Here CLS is the cache line size (in bytes), and S is the previous size (in bytes).

$$LG = \begin{cases} q - p & if\ q > p \\ q - (p - N) & Otherwise \end{cases} \tag{3.1}$$

$$G' = \begin{cases} CLS - (S \bmod CLS) & \text{if } (S \bmod CLS \neq 0) \\ 0 & Otherwise \end{cases} \tag{3.2}$$

Whenever a gap is created, the heuristic searches through the unpopular procedures for a match to fill the gap. This helps us utilize the space as efficiently as possible. However, this is not implemented in our version as we were afraid it would create too many gaps and that it would impact performance by increasing the number of cache misses due to the less efficient use of memory space.

3.3.3 Time Complexity Analysis

Building the TRG

The pseudo code for Gloy & Smith can be found in Appendix B. The time complexity of building the TRG is built up by a main for loop and an inner for loop with some constant operations. The inner for loop will be bounded by the size of Q, as all other occurrences of a procedure is removed whenever it is pushed to the top, it shows us that Q will never grow larger than N where N is the number of nodes. The position of the node in Q must be found, and this can be done in $O(N)$. The inner loop will take $O(2N) = O(N)$. For the whole function the inner loop will run whenever the function is not in Q. This means it will run a total of $L - N$ times, where L is the number of function calls in the log. We can assume that L will be much larger than N so we approximate the number of times to L. Which means the time complexity will be $O(LN)$.

Placing Procedures

Our implementation for the placement is almost the same as the original in [4] except we do not implement their suggestions to always merge into one node. This means there is a difference in the merging of nodes but the rest is the same. The merging process used is the same as the one discussed in Pettis & Hansen, which gives us that this part of the heuristic has a time complexity of $O(N)$. According to Gloy & Smith [4] the merging process is also $O(N)$, which means both our version and the version of Gloy & Smith is the same. This gives us a time complexity of $O(N^2 + SP + C)$ where N is the number of nodes (N is called P in Gloy & Smith), SP is the average size of the function and C is the number of cache lines. See part 4.2.5 in [4] for complete discussion.

3.3.4 Review of Example

The example used is the same as the Pettis & Hansen example, which is discussed in Section 3.1. A figure of the trg_{select} which is the same as the base example, which can be found again in Fig 3.8a and in Fig 3.8b is a figure of TRG_{place}. Whenever an edge is selected it is selected by TRG_{select} and whenever we are comparing two nodes the new placement is decided by TRG_{place}. The first task to do is pick out the popular and unpopular nodes. As this has no exact method and it is not discussed in this thesis it is skipped. Instead all functions are treated as popular procedures. Initially, the different graphs must be created.

The merging process is the same as in Pettis & Hansen, similar to before it begins by finding the edge with the highest value. The first step is merging A and B which is shown in Figure 3.10, this is done in such a way so they do not collide in the cache. The nodes are placed at different positions in the cache shown in Figure 3.9.

Cache line	Functions
1	A
2	
3	
4	

(a)

Cache line	Functions
1	B
2	
3	
4	

(b)

Cache line	Functions
1	A
2	B
3	
4	

(c)

Figure 3.10: Shows the tables for merging the A and B node

In the case shown in Figure 3.10 there are no other functions in the cache so the nodes are placed after each other. This gives B an offset of one e.g. it is at the second line in our simulated cache. C is subsequently merged into the compound node AB. C takes up two cache lines which means it will have to be split into two parts. C1 and C2.

Cache line	Functions
1	A
2	B
3	
4	

(a)

Cache line	Functions
1	C1
2	C2
3	
4	

(b)

Cache line	Functions
1	A
2	B
3	C1
4	C2

(c)

Figure 3.11: Shows how the compound nodes will look when merging AB with C. Notice that C has a size of two cache lines so it actually uses two cache lines. 3.11a shows the AB compound node, 3.11b shows the C node and 3.11c shows the merged ABC compound node

Figure 3.11 illustrates the merging of C (Fig 3.11b) with the compound node AB (Fig 3.11a), when the mergin is done the state of the cache is like in Figure 3.11c. It can be seen that the functions in A, B and C fit in one cache. Now the largest edge is between E and F. When merging E and F the heuristic will ignore how the ABC node is laid out, as there is no connection with that node, so the process for E and F will be the same as for A and B. However, when merging E and F there is an edge between E,D with a cost of 75 and F,D with a cost of 50. As in the Pettis & Hansen example, those edges will be merged together so the edge D,EF is created with a new cost of 125. This new edge is the largest, so the next step is to merge D to EF. Again this is the same procedure as with the ABC node.

Cache line	Functions
1	A
2	B
3	C1
4	C2

(a)

Cache line	Functions
1	G
2	
3	
4	

(b)

Cache line	Functions
1	A
2	B,G
3	C1
4	C2

(c)

Figure 3.12: Merging G into the ABC node. Notice that because G has a collision with A and C they are given an offset of 1

We continue by merging the edge ABC, G (Figure 3.12). When merging these nodes they will actually get a gap in our layout, this gap would be populated with one of the unpopular nodes that fit, but that would occur during the final placement. Continuing the edge ABCG, H are merged and finally the last edge is also merged into the last two nodes (Figure 3.13).

Now all procedures have a starting cache line in our layout, but this is only a relative starting address. To generate a complete layout, the gap between each function must be calculated and it is also necessary to add a small gap so all functions will start at a new cache line. This is done via calculating the gap between all nodes with Formula 3.1 and always minimizing the gap with Formula 3.2. However,

Cache line	Functions
1	A
2	B,G
3	C1,H
4	C2

(a)

Cache line	Functions
1	D1
2	D2
3	E
4	F

(b)

Cache line	Functions
1	A,D1
2	B,G,D2
3	C1,E,H
4	C2,F

(c)

Figure 3.13: The final merging. This will be used to find a good placement by minimizing the gaps

this will not be discussed as it is trivial.

3.4 Hashemi, Kaeli & Calder (HKC)

Instead of going for the full approach of both generating a new call graph and reordering the code, Hashemi, Kaeli & Calder, hereafter referred to as HKC, have instead focused more on the actual reordering [5]. Like Gloy & Smith, they keep track of how much space each procedure occupies and subsequently how many cache lines it will reside in. HKC use colors instead of numbers to signify where nodes are placed. The number of cache lines used by this function determines the number of colors it will be assigned when they are doing the placements. These colors will also be used for the so called unavailable set, which will consist of the colors that the neighbouring nodes currently use.

The main idea is to find a placement where each procedure is placed so it does not collide with any of the colors in its unavailable set. One of the things HKC discuss is how most of the runtime is spent in a few number of functions [2]. Instead of placing all functions they remove some of the functions called unpopular functions (See Section 2.3) and use them as glue or padding whenever the heuristic generates a gap.

In this part we will instead of discussing about compound and non-compound nodes discuss mapped and unmapped nodes, where a mapped node is similar to a compound node and unmapped is similar to a non-compound node. This change is done to stay consistent with the original paper and note that there is a subtle change in how they are treated. While a compound node is always 2 nodes merged together a mapped might consist of a single function. The position of a node inside a mapped node is not set in stone until all of its edges have been investigated. This means that as long as a node has an uninvestigated neighbour it can be moved.

3.4.1 How to Place Procedures

When doing HKC's placement strategy they begin with a WCG from Pettis & Hansen. From this WCG they pick out which edges are popular and unpopular edges (see Section 2.3). The popular edges are after this sorted into descending order just like the previous heuristics. The next step is where placement of procedures actually starts. When going through all the edges there can occur four different cases, these will be discussed in the comming sections. After each of these cases all colors of the nodes that have been moved as well as the unavailable set must be updated. Which will be discussed in Section 3.4.1.

Case 1 - Two unmapped functions

The first case is the simplest case where the nodes are two unmapped functions. Whenever this occurs the nodes are assigned a new color each which is arbitrarily chosen as they have no other colors set yet. This also generates an unavailable set for each node which consists of all used colors except the one given to them. Then the nodes are merged into a mapped node so the unavailable set must also be updated. For our example, node A and B is merged and has been given the colors yellow and green. A has the unavailable set of green, while B has yellow.

Case 2 - Two mapped functions

The second case occurs when the two functions are already mapped to locations in the cache, i.e. when both nodes are mapped nodes. When merging the nodes there will be multiple possible orderings which they can be put in, we must first discuss which ordering is preferred. As an example we look at the nodes B, C, D and E from our example discussed in Section 3.1. Given that nodes B-C and D-E already have been mapped and the edge C,D is to be investigated there are four possibilities: B-C-E-D, C-B-E-D, E-D-B-C, and E-D-C-B. First the heuristic need to find which mapped node is smaller. Both mapped nodes are equally large at three cache lines each so the heuristic picks B-C to be the smaller node. So the heuristic must find on which side to place B-C on E-D. Hashemi et al. suggest that you use a heuristic where you calculate the amount of cache lines between the middle of the function in question, in this case D. More specifically, distance to the procedure modulo the cache size. This heuristic is in place to minimize the probability of the two nodes conflicting with each other. For our example D is a two cache line function and E is a one line. E is to the left of D and the heuristic looks at the middle of D. Then it finds that to the right there is one cache line to the edge and to the left there is two (one from E and the leftmost line from D). Therefore the heuristic places B-C to the right of E-D.

The next step is to find which order E-D and C-B should be combined in. There are two possibilities left: E-D-B-C or E-D-C-B. This is solved by making sure the nodes in the edge are as close as possible. Once again the edges closeness is the same as distance in Section 3.2.1. This means the final position is E-D-C-B. To finalize this step, each node may not overlap each others unavailable sets. If there is a collision in the unavailable sets the smaller node is moved one step. If a position without a collision cannot be found the first position is picked and if a position without conflicts can be found then the gap is filled by unpopular procedures. Whenever nodes are moved around in this case, the node to be moved is always the one with the least amount of nodes in its mapping if the nodes are equal one is arbitrarily chosen. This examples unavailable set and color is discussed further in Section 3.4.1.

Case 3 - One mapped function and one unmapped function

The third case handles the event when an unmapped node is going to be merged with a mapped node. The biggest difference from when there were two mapped nodes in the previous case is that the unmapped node will always be the smallest node. The first step is to decide on which side it should be placed. Here the same heuristic discussed above is used, where the shortest distance between functions is found and measured in cache lines. When the side to place the unmapped node have been found it is checked against the colors in the corresponding unavailable set. As long as there is a conflict, more spaces are added between the two nodes. If a placement with no conflicts cannot be found the first placement is taken.

Case 4 - Two functions inside the same mapped node

The last case is when there is an edge between two nodes in the same mapped node. This case is one of the most interesting as here the unavailable set can be used to avoid cache conflicts. This case is only relevant if the two nodes connected by the edge have the same colorset. As whenever there are conflicting colors, then those nodes will collide in the cache, so when there are conflicting nodes they start by finding the node closest to one of the edges. This node will now be moved to the edge of the mapped node, the gap created from moving this node can be filled by unpopular procedures. The unavailable set is checked for a new placement similar to the other cases. If a new conflict is found the nodes continue to be shifted away from the node until a placement is found without conflicts. If a placement without conflicts cannot be found the node is placed at its original position which is inside the mapped node.

Colors and The Unavailable Set

When the placement have been found after one of these cases, the color distribution must be updated and the unavailable sets as well. We will use the example in case two (Section 3.4.1) to investigate the changes that will happen. The first table in Figure 3.14 shows which color is assigned to which cache line (red, green, blue and yellow). The two tables below are how the mapped nodes are built up before hand and the last one is the final placement.

The example is the two mapped nodes D-E and B-C are combined and the B-C node is added onto the D-E mapped node. As D and E have not been moved then both of their colors will not change, but nodes B and C are moved so they will be assigned the colors next to the D-E mapped. One observation that can be seen is that the color from the previous mapping does not matter, the heuristic only care for the colors after the merging checks for conflicts are done. The heuristic places

r	g	b	y
E	D1	D2	

r	g	b	y
B	C1	C2	

r	g	b	y
E	D1	D2	C1
C2	B		

Figure 3.14: The different forms of merging the mapped nodes D-E and C-B. The first table shows which color is assigned to which cache line, being red, green, blue and yellow. The two tables below are how the mapped nodes are built up before hand and the last one is the final placement.

C and D next to each other which means it changes C1,C2 and B to its new colors which can be seen in the figure to be yellow, red and green in respective order. The unavailable set changes for each node to contain its new neighbours. There are no changes to nodes B and E while C and D add each others color so D will have the unavailable set of red and yellow and C will have the unavailable set of green and blue.

3.4.2 Time Complexity Analysis

This heuristic uses the same dependency graph as Pettis & Hansen 3.2.2, which means N and E is the same as was proposed by Pettis & Hansen, this also gives us that the time complexity for building the graph is $O(L)$. The time complexity for the placement begins by sorting all edges which can be done in $O(Elog(E))$ where E is the number of edges. For each edge it is checked if they are mapped or not which is done in $O(1)$. From here, the nodes are checked to decide on which case should be run.

In case 1, it is only concatenated to the two nodes and map them. This takes the most time during the end as all positions in our mapped nodes must be updated. This takes $O(N)$ time where N is the number of functions.

In case 2, the closest edge of the mapped nodes must be found so that the length in functions between the nodes can be minimized. In the worst case, the proposed way to find the distance to an edge is done in $O(N)$, as all nodes must be checked. After all nodes have been checked, one of the nodes must be shifted around in the color space if a conflict exists. All possible colors must be checked, which is the same as the number of cache lines L, which gives a time complexity of $O(L)$. So the final complexity is $O(L + N)$ for case 2. Case 3 is the same as case 2.

Case 4 is constant if there are no collisions. If a collision is found the node closest to an edge of the mapped node, which in the worst case can be at most $1/2N$. Then all possibilities in the color space must be checked which as noted earlier is equal to the number of cache lines L. This gives us a time complexity that is the same as in case 2 and 3, $O(L + N)$. In total, for all edges in the graph is either a constant operation, or $O(L + N)$ if a collision occurs. This gives us the final complexity of $O(E * (L + N))$.

3.4.3 Review of Example

We begin with the example discussed in Section 3.1 for this review. The first step in this heuristic is to sort all edges so the heaviest edge can be found. We begin by picking the edge with the highest edge cost, which in this case is A-B. Both nodes are unmapped, so case 1 is applied. The nodes are placed in the cache like in Figure 3.15. From the figure it can be seen that A have been assigned the color red and B the color green. While their unavailable set is green for A and red for B.

r	g	b	y
A	B		

Figure 3.15: The first two nodes have been merged into a mapped node

The placement is arbitrary but placing new functions at the beginning of a cache seem like a good starting point. The next node in our sorted list is picked as it has the highest cost, which is B-C. Now there is an unmapped node and one that is mapped. This translates to case 3 and the new node is put at the edge closest to the node in the mapped, which is shown in Figure 3.16. Here it can be seen that C will be assigned two colors as the function occupy two cache lines. Of course the unavailable sets will have to be updated after this change. B will add the colors of C to its unavailable set i.e. blue and yellow. A will be unchanged and C will add green to its unavailable set. Going forth the next edge to be inspected is E-F. Again

r	g	b	y
A	B	C1	C2

Figure 3.16: C has been merged with the AB mapped node

there are two unmapped functions to be merged, which means case 1 is repeated. It should be noted that this mapped node is separate from the first one as there is no edge connecting them. Then edge E-D is added with case 3. Again the functions are combined in such a way so that they are as close as possible. The results of these changes can be seen in the cache in Figure 3.17. Note that we show them as two separate nodes as they have both been mapped but not linked together yet. This is also shown in how the unavailable sets are changed. All nodes in the ABC node have no changes and F will have the unavailable set of green, E will have red, blue, yellow and D will have green. Going further with the edges the edge C-D is

r	g	b	y	r	g	b	y
A	B	C1	C2	F	E	D1	D2

Figure 3.17: We have merged the FED node but note that it is not connected to the other as they are not in the same mapped node

next. Both C and D have been mapped but in different already mapped nodes.

This means case 2 is to be used. First the FED node is placed to the right of ABC. The placement does not really matter as the distance between D and C is equal for all mappings. Remember it is the distance in cache lines that is important. However, their colors collide so the FED node have to be moved further away from the ABC node. If they are moved one step to the right they are still in conflict, now between D1 and C2. It is moved one step further and there are no conflicts between the nodes. This means a better placement have been found. The next nodes are F-D and both nodes exist in the same mapped node. case 4 is used to determine their placement, but the current placement has no conflicts so they are left as they are. Next are the edges H-A and G-H, which both have one unmapped node and a mapped node, this means case 3 can be applied to both of them as done earlier. We begin by placing H to the left of A which creates no conflicts with our current data. The same is done with G-H. In Figure 3.18 the results of these last steps can be seen. Note that the lines should be seen as one continues line instead of a separate, they are shown as separate to visualize possible conflicts.

r	g	b	y
		G	H
A	B	C1	C2
F	E	D1	D2

Figure 3.18: Merged together ABC and FED. Nodes G and H have also been merged

When the next edge is placed which is H-D it can be seen that the unavailable set of H is updated with yellow and the same for D. Remember that the unavailable sets does not contain its own colors and only its closest neighbours. This means there is conflict and not only that but both nodes are in the same mapped node so case 4 is used to hopefully find a better placement. This is done by moving H. However, depending on how you calculate distance you might move edge D instead. We move H one step to the left, away from the main mapped node over G where no conflicts arise. The final position of H is shown in Figure 3.19. We find the same

r	g	b	y
	H	G	
A	B	C1	C2
F	E	D1	D2

Figure 3.19: Moving H because it collides with D

problem with the G-C edge and it is moved over the H node to find that there are no conflicts. To find a placement without conflicts the G-C edge only needs to be moved one space. Now there is only one edge left: A-G. However, it falls under case 4 and in the current placement there are no conflicts. And with this, the final placement is found in Figure 3.20. It is worth noting that if there where unpopular procedures the holes could potentially be filled.

r	g	b	y
G	H		
A	B	C1	C2
F	E	D1	D2

Figure 3.20: The final placement for HKC

3.5 Other Work

Bershad et al. propose to use a hardware device called a cache lookaside buffer which detects cache misses and records the history which is used to dynamically remap pages [3]. McFarling proposes a heuristic using Directed Acyclic Graphs (DAGs) which gives a direct mapped cache decrease in cache misses being the same as increasing the cache by a factor of three [12]. McFarling also suggests creating test cases derived from real world applications as it is hard to get good coverage on very large applications using the current test module. Because of the mentioned difficulty in getting good coverage on very large applications, McFarling suggests using data from a small numbers of users based on how they use it [13]. As is discussed in [20], static analysis can be used to generate our call graph to possibly get a better view of the whole program instead of just for one example.

Kalamationos and Kaeli [10] also create another way to generate a call graph. This has some similarities to Gloy & Smith like the fully associative cache. The biggest difference towards the other versions is that they add a weight dependent on how many rows are affected in between the different calls, and they add this weight to all functions between the different calls.

Chapter 4

Methodology

4.1 Implementation Details

The software we ran these heuristics on is a prototype to be used in Ericssons 5G base station. The base station handles the air interface between a mobile and the telecom network. It sorts all incoming and outgoing transmissions in order so that each user receives the correct data and sends data to the correct recipient. More specifically the tests were run on the scheduler module, which makes sure that all incoming signals are calculated in the correct order. This module is built up in total of 198 different functions where most of the functions have a small size, between 10-100 bytes and a few as large as up to 1000 bytes, where the size of the function is the sum of the size of all instructions for that function. In the module we are looking at about 20% of the code stands for 80% of the execution which follow the results published by Calder et.al. [2]. It should also be noted that this kind of application is static in its execution. This means that whenever the program is run there are few to no changes in the order which all functions are executed in.

The smallest of the functions all had a mean runtime in the interval of 200 ns to 600 and the largest functions go up to 54000 ns. The runtime data seem to follow a similar distribution as the sizes. The standard deviation is harder to sum up but as has been discussed this software is very static and so the standard deviation is in general very low. All times for the functions where calculated inclusively which means we only look at when every functions starts and when it returns.

The generated call graph for the standard layout showed a graph with many edges or in other words calls between functions. In this graph there are a few groups of functions which call each other frequently and there are also many functions that are called very rarely. Of all the functions that are called rarely, there are some functions that should not have any calls in-between. This can be attributed to the fact that the log is not reliable in how calls are ordered. We will return to this discussion later in Section 4.3. However, this should not affect our results as the heuristics go for the edges with the highest weight or most calls first and only change it if a better solution is found.

The tests where run on Ericssons own hardware which is a multicore architecture where each core has its own L1 instruction cache and there is a common memory which is shared between all cores, which holds both the program and data. The data is vectorised, which means all data is split up into smaller chunks and sent to each core, so each core runs the same functions but on different data. The size of the instruction cache follows the industry standard which is 32KB. The data cache is not relevant and it is also counted as proprietary information. In theory the whole application should fit in the cache but earlier practical tests at Ericsson DURA show that this is not the case. This comes from how the cache is being used. But how the cache is used is proprietary information.

There was also limitations in the trace framework, we did not implement any of the parts where the granularity for placement has been smaller than procedures. In the end we decided to implement three heuristics: using Pettis & Hansen [14], Gloy & Smith [4] and random placement. A fourth heuristic was discussed but due to implementation of the integration systems for EMCA (Ericsson Multi-Core Architecture) taking longer than expected it was never implemented. The heuristic is still discussed here for the possibility of future implementations. The kernel functions could also not be moved, which in theory should generate the largest increases as they are the most time critical and called more often than other functions.

4.2 Testing

As this project aims to find if code reordering is viable for EMCA Systems, a few different reordering approaches are tested. To get the results, each heuristic will run and collect data on runtimes for all functions. To complement the findings of the heuristics, randomized orderings are made which will give us some idea of how much the placement can change the runtimes. Each of the logs used are two million lines long. Which gives that there around a million function calls being investigated.

From this data we will examine a top level function and one of the smaller functions which is called more often. Then these functions will be compared by generating histograms for each run. We will also do a run where only the top level functions are logged. A top level function is defined as a function that calls a lot of smaller functions.

The experiment with the random placement was first done 10 times. However, the results were extremely stable between the different runs. The variation between all those runs where all in the same span as the other functions during the earliest test. Based on this experience we decided that a single run of each later experiment was sufficient to evaluate each situation.

When looking for interesting functions we will be looking for functions where the histograms have more than one peak. If the function is not dependent on an external source (for example functions that use memcpy) we can assume that that function has a collision with some other functions. Whenever there is only one peak we cannot say if that peak signifies either many cache misses or cache hits. If many cache hits are seen there should be a decrease in execution time.

4.3 Problems During Measuring

It should be noted that these tests are not perfect and most of the imperfections come from the variations in the logs. The first problem came from having large jumps occurring in the logs. These are created by the chips syncing to an external source like an external card or that a trace buffer becomes full and a new trace buffer is allocated. The log information is also sent via UDP, so chunks of the log can disappear. These problems create large outliers and negative execution times which disrupts our calculations. However, they were easily filtered.

Chapter 5

Results

Several different types of tests have been run on Ericsson's baseband prototype. More specifically all test have been run on the channel resource scheduler, which manages the time and frequency resources for the base station. Whenever there is a new test it is the layout in the .map file that has been changed in some way. The first test is the compiler generated layout. This is the standard case and it is the base case which all comparisons to other tests are done with. There are also generated logs for two runs with the same layout based on the standard layout, which gives us a picture of how much two runs can vary without changing the layout. Additionally, there are two random runs to get a measure on how large the total variations can be.

5.1 Differences for Base Case

There are differences between two runs with the same layout and this test tries to quantify that. The standard layout is used to show these differences. In Figure 5.1a and 5.1b it is shown that even though they are quite similar there are some differences, especially how the lower part tapers off faster in Figure 5.1b. We can assume that the larger the high execution time boxes are, the more cache misses occurred during that execution.

5.2 Looking at Specific Functions

The function under investigation in Figure 5.2 is a top level function where the time from the lower level functions are included. From this an aggregated view of the effects of our placement in the memory with these histograms can be found.

The histograms in Figure 5.2 show that there are very small differences between the functions. To be more specific, it can be seen that for this specific function the one that performed the best is Gloy & Smith in Figure 5.2c. Gloy & Smith is considered the best performing due to the taller boxes at the lowest times and also the lower boxes at the highest times. These changes are very small and very

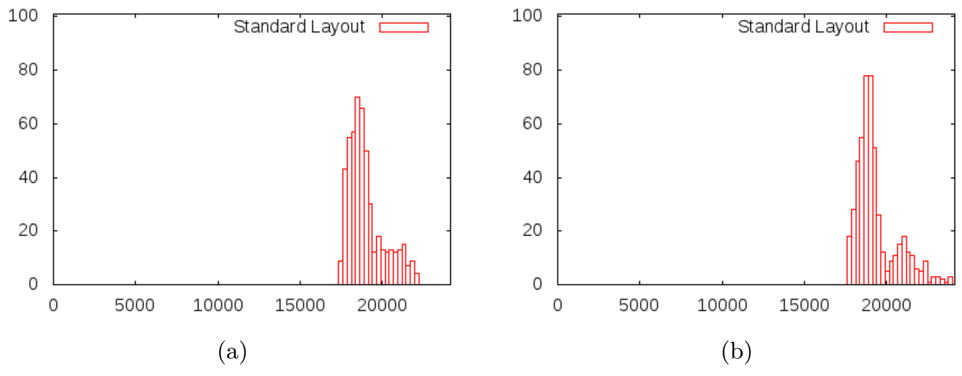


Figure 5.1: Two separate runs with the unmodified standard layout. The X-axis shows the time in nanoseconds and the Y-axis shows how many times that interval occurred during the log.

hard to see. When looking at other functions we still find the same small differences between the layouts. In Figure 5.2 one of the functions with the largest differences is shown. We have looked at how the top level functions differ between different layouts. We will now look to Figure 5.3 and see how smaller functions were affected. In these pictures two boxes with about 20 ns between them can be seen. The first box represents all of the times a cache hit was found and the second box is a cache miss. This can not be stated for certain as the value for the stall we calculated at Ericsson should be larger. Going on the assumption that the difference is when a cache miss or cache hit was found, we can clearly see that the assumed amount of cache misses does not change in a significant way between the different layouts. We can see some changes between them, but they are so small we cannot say it is from our layouts.

5.3 The Bigger Picture

We can also look at how the whole application differs by looking at scatter graphs where the standard deviations for each function is shown. In Figure 5.4 each graph represent a run compared to the standard output. Each dot represents a function and whenever it is close to the black line in the middle there is little to no difference for the standard deviation. When the dot is below the line the function has performed better than the standard case while it is above it is performing worse. While there are some functions performing better, a few are performing worse and the most keeping close to the middle line. This illustrates what have been discussed earlier, that there is hardly any difference between the different layouts.

It should be noted that the standard deviation calculation is done on bimodal distribution, which makes the results a bit unreliable as discussed in Section 1.4.1. When calculating the standard deviation it cannot be said if a decrease comes from

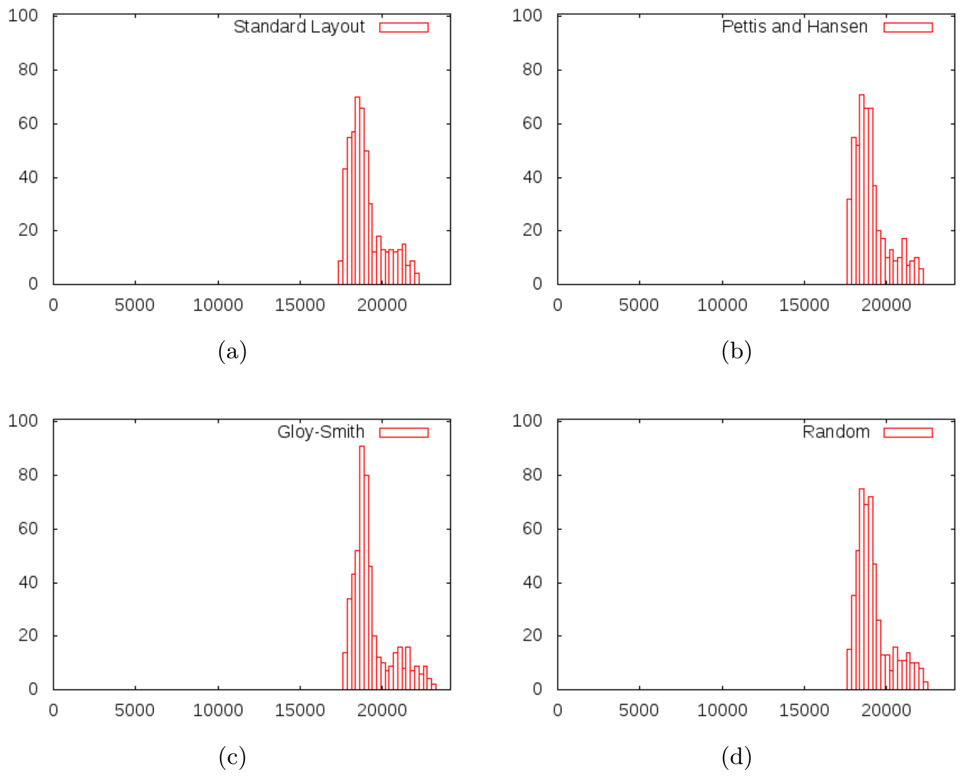


Figure 5.2: Results for a top level function where we are logging all functions showing different layouts. The X-axis shows the time in nanoseconds and the Y-axis shows how many times that interval occurred during the log. (a) shows execution times with the standard layout, (b) shows the times with the layout from Pettis & Hansen, (c) shows the times with the layout from Gloy & Smith and finally (d) shows the execution times for a random layout

less or more cache misses. Either case can decrease the spread of the data and that is what is being investigated when calculating the standard deviation. However, when looking at the data presented in the earlier figures and compare it to the data in our scatter graphs we see that the results are very similar. The relative changes of the standard deviation (Figure 5.5) can be investigated to see further details. Everything is compared to the same base case as before. The most important fact to note is the general form of this histogram. The form tells us that most functions' differences are very small with a few being better but many being even worse.

5.4 Only Logging Top Level Functions

We also tried what effects the logging function had on our data. Whenever the logging function is run, those functions are loaded and disrupt the cache. This made us test our different layout heuristics when only logging the largest top level

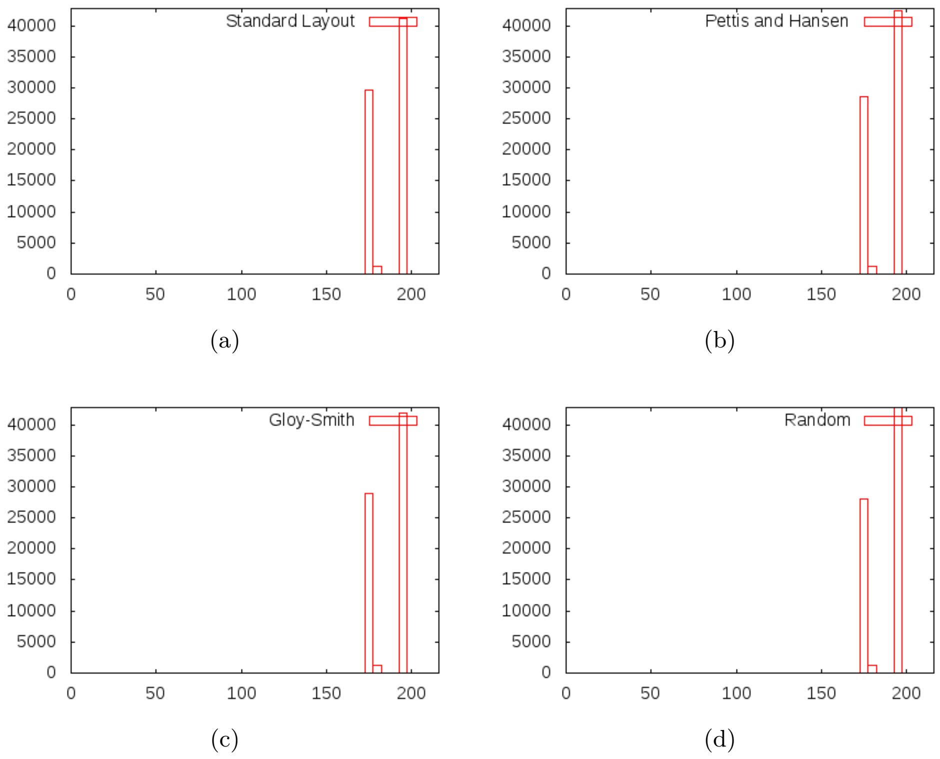
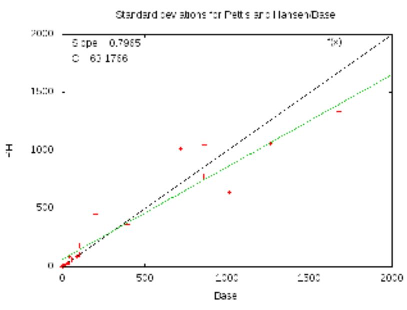


Figure 5.3: Results for a low level function with different layouts. The X-axis shows the time in nanoseconds and the Y-axis shows how many times that interval occurred during the log. (a) shows execution times with the standard layout. (b) shows the times with the layout from Pettis & Hansen, (c) shows the times with the layout from Gloy & Smith and finally (d) shows the execution times for a random layout

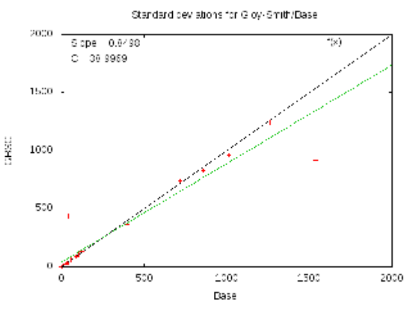
functions, the results can be seen in Figure 5.6. The function you see is the same as in Figure 5.2. We can directly compare these figures which show us that it has almost halved the total execution time and that the variance has gone from ±2500 to about ±500. This means the logging is a huge culprit, but it does not give any definite results while changing layouts. However, it shows promise that better results can be obtained if the kernel functions can be placed.

5.5 The Numbers

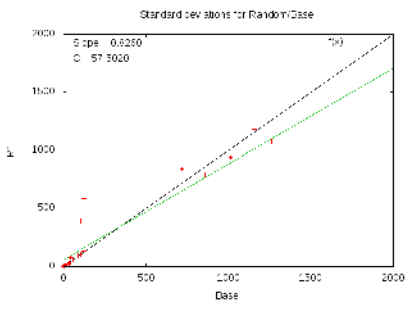
This chapter ends with the standard deviations and arithmetic means used in the previous graphs. This data can be seen in Table 5.1. The first observation to note is that Function A and C are the functions discussed earlier. A is the large function and C is the small function. We can see in Function A that both standard deviation and mean decreases which as discussed in 1.4.1 shows that the results are only positive. However, this is not the norm, the normal function is like Function C



(a)



(b)



(c)

Figure 5.4: Scatter graphs comparing the standard deviation between the standard case and different placement heuristics. Each dot represents a function where its x coordinate is the standard deviation for the so called base case. The y coordinate represents the standard deviation for the run with a generated layout. A linear regression is also done on the data to get an easier comparison which is the green line. The black line in the middle functions as a reference line. With Pettis & Hansen in (a), Gloy & Smith in (b) and Random in (c)

where the differences are insignificant. The final case is shown by Function B which have become worse for all. The changes found in functions like Function B eats up any improvements found in the functions like Function A, which supports the data found in our scatter graphs in Section 5.3.

Another observation that can be seen from the data is that in most cases the standard deviation and mean change simultaneously. If one of them rises the other rises as well and conversely if they decrease both decreases. This strengthens our measuring method as if there are less cache misses it should be both less variance and faster functions (smaller means). The number of runs is also an interesting variable to investigate but none of the functions have larger then 0,4% difference.

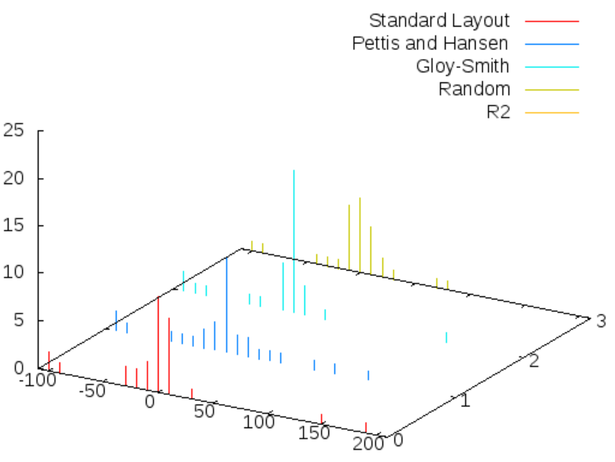


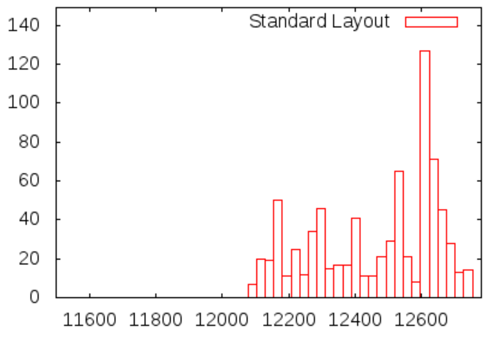
Figure 5.5: Relative changes between the different layouts and standard layout. Each color shows a different layout. On the x-axis the changes is shown in percent and the y-axis shows how many fall in that interval. If you cannot see the colors the 0 on the z-axis is the standard layout, the 1 is Pettis & Hansen, the 2 is Gloy & Smith and the 3 is random layout

	Standard, First Run		Standard, Second Run	
	Std Dev	Mean	Std Dev	Mean
Function A	1261,3	19482,71	1097,63	19133,1
Function B	716,95	21927,14	799,52	21683,39
Function C	8,1	187.79	8,04	188,31

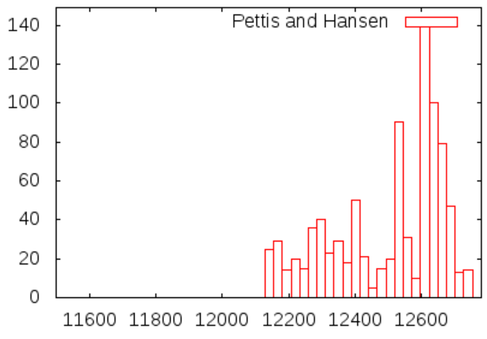
	PH		GBSC	
	Std Dev	Mean	Std Dev	mean
Function A	1059,71	19186,86	1240,56	19480.99
Function B	1017,26	22021,1	742,74	21932.66
Function C	8	188.57	8.02	188.45

	Random	
	Std Dev	Mean
Function A	1075	19329.61
Function B	842,37	22050,79
Function C	7,98	188,67

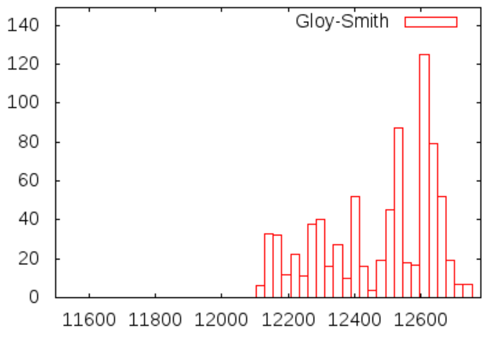
Table 5.1: Tables with standard deviations and arithmetic means for the different layouts.



(a)



(b)



(c)

Figure 5.6: Results for the same function as in Figure 5.2 but now only the large top level functions of the code are logged. Once again the X-axis shows the time in nanoseconds and the Y-axis shows how many times that interval occurred during the log. (a) shows execution times with the standard layout. (b) shows the times with the layout from Pettis & Hansen and (c) shows the times with the layout from Gloy & Smith

Chapter 6

Discussion

6.1 Conclusions

We are certain that there is no practical benefit of adding code placement into the compiler tool chain on an EMCA (Ericsson Multi-Core Architecture) System. Even if we cannot say anything on how the number of cache misses are affected, there have been no general increase in performance or reliability given our results in Chapter 5. There is a possibility that better results can be achieved if assembler and kernel functions can be placed.

6.2 Reasoning

There are a multitude of possible reasons for the results we have found. The biggest reason is that the logs which have been used for time measurements have as discussed in 4.3 some uncertainties. For example there were large gaps for a few different reasons which created odd calculations

We have also not been able to measure the number of actual cache misses. If the number of cache misses was available to us we could know when a cache miss was found and when it is not. Scrutinizing our runs, the data suggests that there were not many cache misses. Furthermore, all functions cannot be traced as there is no information from kernel functions, which means these functions are excluded from the new placement. These functions are some of the most called and most time critical functions so if they could be placed we should be able to get much more reliable results.

We did think that the actual logging was a culprit as when a function starts it calls the generate trace information function. When that function is called it most likely generates more cache misses. The logging function uses its own internal write queue and it was never really built for producing log events for all functions, so this could produce considerable slowdowns. One of the tests tried to remove almost all log events to see how much the logging function affected the system. This was done by only creating log events at the top level functions. The expected speedup was seen but no changes in any of the patterns were found.

If we look at the different heuristics we see that Gloy & Smith can generate gaps in the placement. These were removed as the limitations of space was considered more important. By adding these actual gaps it could have attained the intended layout and possibly seen improvements.

According to [15] there are small to no improvements found for placement of procedures in operating system code. Our application is more like an operating system rather than an ordinary application.

6.3 Future Work

Ignoring the measurement problems we should look to find different ways of generating the WCG. Static analysis could generate a better WCG according to [20]. If the granularity of placements could be increased in the EMCA Systems, it would be possible to make one of the largest improvements according to most researchers like [14] [4] [15] [10]. You could also let a part of the cache be statically allocated and let kernel functions reside there, as of right now kernel functions are hidden from us. During our implementation of Gloy & Smith we could not get temporal information on chunks and all our data came from how procedures interacted. This suggests that we should get better results from again being able to get log events on smaller chunks or use statistical analysis to get these smaller chunks like noted earlier.

In this report we have discussed the work of HKC [5]. It has not been implemented but this heuristic should give better results for this specific platform. In our version we try our best to create a small placement to make sure the effects on the binary size is mitigated. Even though size will play a large role in execution speed, the effects of those holes in the placement will most probably outweigh the effects on the speed. This is a delicate balance, making it hard to say anything for certain. We should also investigate how the size of the logs might affect the data. A few MB:s of data should be sufficient.

Bibliography

[1] A. Ayers, R. Schooler, and R. Gottlieb. Aggressive inlining. In *ACM SIGPLAN Notices*, volume 32, pages 134–145. ACM, 1997.

[2] and B.Zorn.Quantify B.Calder, D.Grunwald. Static branch frequency and program profile analysis. In *Journal of Programming Languages, 2(4)*, pages 1–11, 1994.

[3] B.N. Bershad, D. Lee, T.H. Romer, and J.B. Chen. Avoiding conflict misses dynamically in large direct-mapped caches. In *ACM SIGPLAN Notices*, volume 29, pages 158–170. ACM, 1994.

[4] N. Gloy and M.D. Smith. Procedure placement using temporal-ordering information. *ACM Transactions on Programming Languages and Systems*, 21:977–1027, 1999.

[5] A. Hashemi, D. Kaeli, and B. Calder. Efficient procedure mapping using cache line coloring. In *ACM SIGPLAN Notices*, volume 32, pages 171–182. ACM, 1997.

[6] A. Hashemi, D. Kaeli, and B. Calder. Procedure mapping using static call graph estimation. In *Workshop on Interaction between Compiler and Computer Architecture, San Antonio, TX*, 1997.

[7] Y. He, K. Chen, J. Gu, H. Deng, A. Liang, and H. Guan. A New Approach to Reorganize Code Layout of Software Cache in Dynamic Binary Translator. In *Parallel Architectures, Algorithms and Programming, 2010 3rd International Symposium on*, pages 175–182. Ieee, December 2010.

[8] W. Hwu and P. Chang. Achieving high instruction cache performance with an optimizing compiler. In *ACM SIGARCH Computer Architecture News*, volume 17, pages 242–251. ACM, 1989.

[9] C. Jang, J. Lee, B. Egger, and S. Ryu. Automatic code overlay generation and partially redundant code fetch elimination. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9:10, 2012.

[10] J. Kalamationos and D. Kaeli. Temporal-based procedure reordering for improved instruction cache performance. In *High-Performance Computer Architecture, 1998. Proceedings., 1998 Fourth International Symposium on*, pages 244–253. IEEE, 1998.

[11] P. Lokuciejewski, H. Falk, and P. Marwedel. Wcet-driven cache-based procedure positioning optimizations. In *Real-Time Systems, 2008. ECRTS'08. Euromicro Conference on*, pages 321–330. IEEE, 2008.

[12] S. McFarling. Program optimization for instruction caches. *ACM SIGARCH Computer Architecture News*, 17(2):183–191, April 1989.

[13] S. McFarling. Reality-based optimization. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003.*, 2003.

[14] K. Pettis and R.C. Hansen. Profile guided code positioning. In *ACM SIGPLAN Notices*, volume 25, pages 16–27. ACM, 1990.

[15] A. Ramírez and L.A. Barroso. Code layout optimizations for transaction processing workloads. In *Proceedings. 28th Annual International Symposium on Computer Architecture, 2001.*, pages 155–164, 2001.

[16] A. Ramírez, J.L. Larriba-Pey, C. Navarro, J. Torrellas, and M. Valero. Software trace cache. In *Proceedings of the 13th international conference on Supercomputing*, pages 119–126. ACM, 1999.

[17] A. Ramírez, J.L. Larriba-Pey, and M. Valero. The effect of code reordering on branch prediction. In *Proceedings 2000 International Conference on Parallel Architectures and Compilation Techniques*, pages 189–198. IEEE Comput. Soc, 2000.

[18] W.J. Schmidt and R.R. Roediger. Profile-directed restructuring of operating system code. *IBM Systems Journal*, 37(2):270–297, 1998.

[19] André Seznec. A case for two-way skewed-associative caches. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, pages 169–178. ACM, 1993.

[20] Y. Wu and J.R. Larus. Static branch frequency and program profile analysis. In *Proceedings of the 27th annual international symposium on Microarchitecture*, pages 1–11, 1994.

Acronyms

BFS	Breadth First Search.
DAG	Directed Acyclic Graph.
EMCA	Ericsson Multi-Core Architecture.
HKC	Hashemi, Kaeli and Calder.
LCF	Linker Control File.
LRU	Least Recently Used.
RAM	Random Access Memory.
TRG	Temporal Relations Graph.
UDP	User Datagram Protocoll.
WCG	Weighted Call Graph.

Appendix A

Pseudo Code - Pettis & Hansen

Algorithm 1 Building the WCG

```
for all Reference R in Trace do
    if R is a function enter then
        Add R to Callstack
        if R not in Graph then
            Add R to Graph
        end if
    else
        Pop R from Graph
        Add 1 to edge between R and Top of Callstack
    end if
end for
```

Algorithm 2 Generating the Placement

```
function MAIN(WCG)
    while Edges exists do
        TopEdge = Edge with highest cost
        (LeftNode,RightNode) = GETPOSITIONING(TopEdge.left, TopEdge.right)
        MERGENODES(LeftNode,RightNode)
    end while
end function

function GETPOSITIONING(LeftNode, RightNode)
    if LeftNode and RightNode isCompound then
        return GetClosestNodes(LeftNode, RightNode)
    else if One Node isCompound then
        return GetClosestNodes(CompNode, NonCompNode)
    else No node isCompound
        return (LeftNode, RightNode)
    end if
end function

function MERGENODES(LeftNode, RightNode)
    remove edge between leftNode and rightNode
    for all Edges in both leftNode and rightNode do
        Merge leftNode.edges with rightNode.edges
    end for
end function

function GETCLOSESTNODES(LeftCompoundNode,RightNode)
    pathRight = BFS(LeftNode.RightMostNode, RightNode)
    pathLeft = BFS(LeftNode.LeftMostNode, RightNode)
    if pathRight.length < pathLeft.length then
        if RightNode.isCompound() and pathRight.Last == RightNode.Left then
            reverse(LeftCompoundNode)
        end if
        LeftCompoundNode.append(RightNode)
    else
        if RightNode.isCompound() and pathLeft.Last == RightNode.Left then
            reverse(leftCompoundNode)
        end if
        LeftCompoundNode.append(RIghtNode)
    end if
end function
```

Appendix B

Pseudo Code - Gloy & Smith

The pseudo code is cited from [4] with a few exceptions. Everything is the same except for the removal of cache objects when the cache would become too large. Removed after reading in Kalamationos & Kaeli that they only removed when the function would be called again. In the original they also implement their suggestion to decide on one node to be the compound node and merge everything into that node.

Algorithm 3 Building the WCG

```
Q = Empty;
for all R doeferences R in Trace
    if R in Q then
        k = position of R in Q
        for  i=k-1 downto 0 do
            weight(Q[i],R) += 1
        end for
        delete Q[k]
    end if
    add R to front of Q
end for
```

Algorithm 4 Generating the Placement

```
function PlaceProcedures(procedureTRG PTRG)
    derive working graph G from PTRG;
    while G has edges do
        find e(NY,NX) := maximum weight edge in G
        d := FindBestDisplacement(NY,NX);
        merge NY and NX
        update G to reflect merging of NX into NY;
    end while
    LayoutProcedures(N1);
end function

function FindBestDisplacement(node N1, node NX)
    chunks1 := BuildChunkSetArray(C, N1);
    chunks2 := BuildChunkSetArray(C, NX);
    best_d := 0; best_cost := INFINITY;
    for all d do in [0..C-1]
        cost := ComputeCost(C, d, chunks1, chunks2);
        if ( thencost < best_cost) then
            best_cost := cost; best_d := d;
        end if
    end for
    return best_d;
end function

function BuildChunkSetArray(int cachesize, node N) // builds an array of
sets of chunks from N, one set for each cache line
    for all i in [0..cachesize-1] do
        chunks[i] := ;
        for all (proc,offs) in N do
            for all j in [0..Size(proc)-1] do
                proc_chunk := j / chunk_size_in_cache_lines;
                chunk_offs := (j + offs) MOD cachesize;
                chunks[chunk_offs] := chunks[chunk_offs] UNION proc_chunk;
            end for
        end for
    end for
    return chunks;
end function

function ComputeCost(int cachesize, int d, chunkSetArray a1, chunkSetArray
a2)
    cost := 0;
    for all k in [0..cachesize-1] do
        for all ch1 in a1[k] do
            for all ch2 in a2[(k+d) MOD cachesize] do
                cost += weight from e(ch1,ch2) in CTRG;
            end for
        end for
    end for
    return cost;
end function
```

Appendix C

Pseudo Code - HKC

Although this is not implemented we present the pseudocode here.

Algorithm 5 Generating the Placement

```
Sort all edges into descending order
for all Edges (going from highest to lowest weight) do
    if Case 1 then (Both nodes are unmapped)
        map A and B so they form a compound node
    else if Case 2 then (Both nodes are mapped but to different compound nodes)
        Concatenate C1 and C2 so distance between A and B is minimized
        if Color conflict is found then
            Shift C1 in the color space so no conflict exists
            if conflicts are unavoidable return C1 to starting position
        end if
    else if Case 3 then (A or B is in a compound node but not the other)
        Same as case 2
    else if Case 4 then (Both nodes are mapped to the same compound Node)
        if No color conflicts exist then
            Do nothing
        else
            Move the node closest to an edge of the compound node outside of the compound node
            If there is still a conflict move it in the color space If conflict is unavoidable move the node into its starting position
        end if
    end if
    Update unavailable set
end for
Fill in possible holes with unpopular edges
```