Universiteit Leiden
Opleiding Informatica

Exploring scheduling alternatives for a
Computer Vision application on embedded MPSoCs

Name: Bart van Strien Studentnr: 1110527

Date: 26/01/2015

1st supervisor: T.P. Stefanov
2nd supervisor: J. Spasic

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Abstract

The automated design and development tools for MPSoCs are used to speed up the design process of MPSoCs and often they are able to provide guarantees on performance metrics like throughput and latency prior to actual execution on a real platform. We study a computer vision application that places heavy demands on the hardware, and attempt to discover, and analyse the effects of different scheduling algorithms on the performance.

Using measurements from the statically and dynamically scheduled application, and a mathematical approximation of the hard-real time scheduled application we find dynamic scheduling to provide the best performance, and real-time scheduling to provide the best development workflow.

1 Introduction

Due to the increased complexity of working with modern MPSoCs, and the programs run on them, it no longer suffices to run a single task on a processor. However, running multiple tasks on a processor brings its own set of problems, like how to allocate tasks on these multiple processors. Another problem is how to schedule these tasks so they run efficiently, and to maximise the benefit obtained from running on a multi-processor system. Because there are many possible combinations for allocating an application's tasks on an MPSoC, the design space to explore is huge. To this end, the use of automated design tools is a practical necessity, as they not only allocate and schedule for their users, but they also allow for auto-parallelization of the sequential application into a set of concurrent tasks.

In this thesis we will be focusing on different state-of-the-art scheduling algorithms used for scheduling a computer vision application, namely the Disparity Map Algorithm (see Section 2), on embedded MPSoCs and their performance characteristics. Specifically, we will be using the Daedalus framework and its implementations of static, hard real-time and round-robin (dynamic) scheduling. To accomplish this, we will first develop an application requiring a lot of computational power, and measure its execution time in different circumstances to discern the effect the scheduling algorithms have on its performance.

We will first look at the computer vision application from the title, the Disparity Map Algorithm. We will explain how it works, and how it is ap-

plied. Then we will discuss our hardware platform, and how this ties in with the Daedalus and Deadalus^{RT} toolchain. Furthermore, we explain what what the evaluated scheduling algorithms are, how they work, and what their advantages and disadvantages are. Then we will measure the execution time of our application on the specified hardware for multiple inputs and evaluate what impact the scheduling algorithms have. We predict differences we may observe, and explain them. Lastly, we will draw our conclusion and outline possible future research.

2 Disparity Map Algorithm

A "Disparity Map Algorithm" is an algorithm used to determine depth in stereographic images based on so-called disparity. This algorithm has a lot of uses, for example collision control, object recognition and pedestrian tracking, oftentimes on embedded systems. There are strict limitations on what images this works on. It is important, for instance, that both the left and right images are shot at exactly the same orientation, with the exact same zoom levels, white balance, and so on. Disparity is a simple measure of the relative movement of a point shared by two images. As an example, if one such a point is located 17 pixels right of the left edge in the left image, and only 14 pixels in the right image, the disparity of that point is 3 pixels. As is common knowledge, the closer-by an object is, the further it will move if the observer moves, so it follows that the disparity for such an object must be higher. Therefore, the disparity is an inverse measure of relative distance.

Several implementations of this algorithm were evaluated, using existing code, for the quality of their output, their source quality and its control flow. The source quality was an important element because it became immediately clear that no implementation evaluated would work unmodified and significant effort would have to be spent reimplementing the algorithm. Similarly, the control flow had to be obvious and natural, such that the application could be split into multiple tasks, that could then be run on multiple processors.

2.1 How the algorithm works

The implementation of this algorithm used in this thesis has been based in most part on the earlier work presented in [1]. For reasons outlined later,

this code has been ported to the C language, and the missing image reading and writing functionality was added. This implementation works in two main tasks, an initial guess is made of the disparity map, in the code and hereafter referred to as "$l_0values$". Then it is refined repeatedly to increase the quality of the disparity map, before it is finalised. After i refinement steps we refer to the corresponding values as "$l_nvalues$".

2.2 Initial guess

First, the two greyscale images are divided into a left and right image, and a new image is created to store the $l_0values$. This image has as many "colour channels" as the disparity range, and every one of those values is used to store a how well this pixel matches at that specific disparity value. Then for every pixel in the right image, and for every Δx between the minimum and maximum disparity, the pixel $\begin{pmatrix} x \\ y \end{pmatrix}$ in the right image is compared with the pixel $\begin{pmatrix} x + \Delta x \\ y \end{pmatrix}$ in the left image. The stored result is the normalised "likeliness" from 0 to 1, as calculated in Equation 1.

$$\begin{aligned} disp(x, y, i) &= 1 - |Image_r(x, y) - Image_l(x + i + Disparity_{\min}, y)| \\ disp(x, y) &= \sum_{i=Disparity_{\min}}^{Disparity_{\max}} disp(x, y, i) \end{aligned} \tag{1}$$

After all likeliness values have been calculated for a single disparity value, a blur with the l_0 radius is applied, to make the results more uniform. Then another normalisation step is applied, this time modifying the likeliness values of a single pixel, magnifying peaks and shrinking valleys. Of course not all information is present in both images, and it is hard to compare the edges, so the algorithm also greys out the edges, the size of the border radius.

2.3 Refinement

The refinement step starts by applying another blur, followed by a magnification of likeliness values. This magnification is achieved by taking the sum of neighbouring values and assigning that as the new value, with the size of the neighbourhood being determined by the disparity radius parameter, as

seen in Equation 2. These values are then normalised and multiplied with the original values ($l_0 values$) and themselves.

$$disp'(x, y, i) = \sum_{j=i-D_{radius}}^{i+D_{radius}} disp(x, y, j) \tag{2}$$

Finally, like the initial guess, the borders are cleared of all data, since they can't be accurately determined.

Figure 1 shows the left and right images of a stereo pair, followed by the calculated disparity map as produced by this algorithm.

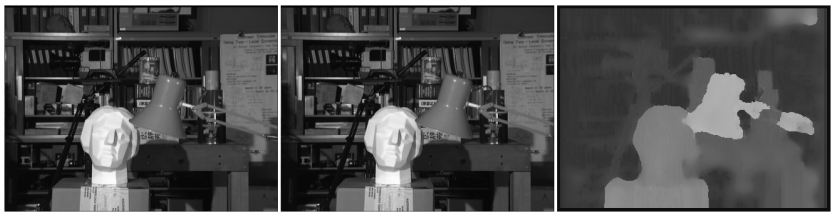

Figure 1: Example disparity map

2.4 Determining the final disparity map

If we consider the disparity values associated with a single pixel, we get a curve where a higher value indicates a better match between the input images. We can determine the highest value in the curve, and we can then assume that is the disparity belonging to that pixel. Once we've found the disparity value where the corresponding likeliness is the greatest, we can then determine the difference between its neighbouring values, and use that to guess if the true peak might be slightly above or below this value, to achieve sub-pixel accuracy. Lastly, we store the disparity value found, and a confidence value determined by the sum of this disparity value's likeliness and its neighbour's likeliness.

Having obtained a disparity map, scaling it becomes trivial, as it is simply a matter of mapping the disparity values back onto a greyscale image.

2.5 Analysis and parameters

After porting this code from C++ to C to prepare for the next step (see Section 3.1), we measured the performance of the application and its component

parts with the intention of finding balanced subtasks. Because it was impossible to completely balance these tasks out, and still have them be sensible, some time was also spent tuning the performance. Not only the code and task selection were optimised, the parameters were similarly treated.

As is to be expected, the algorithm also has multiple parameters that need to be set, for the tested input images a single set of parameters was determined that proved sufficient for all cases. These parameters are listed below.

- Disparity range: 12
- L_0 radius: 1
- Row-column radius: 4
- Disparity radius: 1
- Iterations: 7
- Maximum initial scale: 96%

Though many of these parameters have been discussed above, we shall quickly discuss each of them. The disparity range is measured in pixels, and represents the range of disparity values that can be observed within these images. The various radii are used in the "blurring" mentioned above, and the amount of iterations determines how often the disparity map is refined. Lastly, the "maximum initial scale" determines what the maximum likeliness value is that the calculation of $l_0values$ may contain.

3 Hardware platform

An FPGA, or Field-Programmable Gate Array, is a device that contains a large amount of logic gates, or slightly higher-level parts, and that can connect these together by virtue of an uploaded "program". A complete explanation of the functionality of an FPGA is outside of the scope of this thesis, but the application of it will be discussed further.

For this thesis, the FPGA implements a multi-processor device based on the proprietary MicroBlaze processor architecture by Xilinx. This specific platform was chosen by the Deadalus$^{\mathrm{RT}}$ framework's real-time scheduling

tool (see Section 3.1), that given task timings can determine the necessary amount of processors, and the way tasks are mapped onto them for maximal, theoretical performance. These timings were obtained by running every task on its own processor on a previous platform. Of course, the only way different schedules can be compared in a fair way is with the same available hardware, so the same platform is used for every tested configuration.

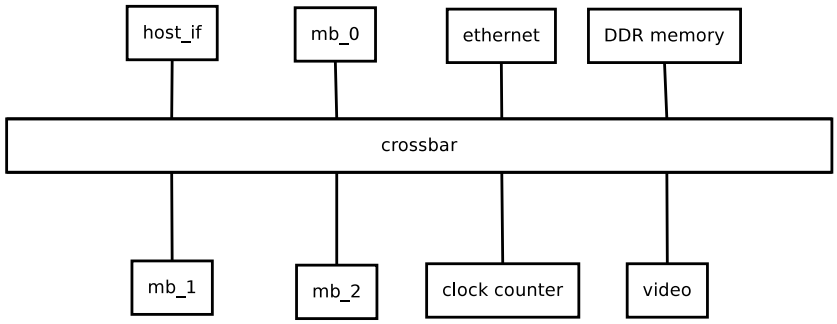

Figure 2: A simplified platform overview

As we can see in Figure 2, our platform consists of 4 microblaze processors, one of them indicated as `host_if`. This host interface is only used to facilitate communication between the board and the computer it is attached to. It is also responsible for measuring the duration of the program. Since it starts the other cores' programs, it simply stores the value of the clock counter when it does so, then determines the difference when all processors have signaled end of execution.

Because of the high memory usage of our program, we store the images (all stages in-between represent their status as images) on DDR memory, but because this travels over the crossbar, and is obviously non-local, this incurs heavy cost. To counter this cost, as little as possible is stored on the DDR memory, and as much as possible is stored on the on-processor memory.

3.1 Daedalus

The Daedalus framework [2] is a set of tools that assists programmers running their applications on MPSoC systems. Traditionally, porting applications to MPSoC platforms involves a manual process. The Daedalus framework, and its accompanying real-time variant Deadalus^{RT} attempt to change this by allowing users to automatically convert their code to MPSoC-ready code.

Given a subset of C code known as SANLP it can automatically determine boundaries between tasks, and the relations between these tasks. Using this information it can derive a process net which is then used for analysis and code generation. The rest of this section will describe our experience porting, explaining the relevant stages along the way.

3.1.1 Porting

For this thesis we started with C++ code, that was first ported to a restricted subset of C code, before it was handed to the first part of the Daedalus framework, PNgen [3]. This subset is also known as Static Affine Nested Loop Programs, or SANLP, and places severe restrictions on what the main function of code can do. Two such restrictions are that no dynamic allocation can occur, and all loops have iteration counts independent of the input. PNgen parses this code and creates a Polyhedral Process Network, a graph that encodes the dependencies between parts of the application. This also intuitively explains why this subset is used, namely because it is easier to reason about for this software. The PPN generated for this application consist of 5 processes (tasks) and has 7 channels, as shown in Figure 3. These processes are:

1. Read: reads the left and right images from the input stream
2. Initial guess: as in Section 2.2
3. Refinement: as in Section 2.3
4. Scale: scales the resulting disparity map down to a single image representing it
5. Write: writes the output image to the output stream

The Deadalus^{RT} [4] framework converts that to a CSDF model, such that the darts tool can then be used for static real-time analysis. The output of the analysis is a platform specification and a mapping specification which give the maximal performance. The number of processing elements in the platform specification is calculated by darts as the minimum number which guarantees the hard real-time schedulability and the maximum throughput of the application. If instead of using a hard real-time scheduler, the static or dynamic schedulers are used, then the mapping and platform specifications

are derived from design space exploration. Together with the PPN providing the application specification, the platform, and the the mapping, we can then use ESPAM [5], which builds an environment ready for use with the official Xilinx tools.

Figure 3: The process net for the application

To assist with developing this code, not just a hardware target can be selected, but also a target called SystemC (an IEEE standardised language [6]). Since SystemC code can be run on a development machine, it is much easier to reason about and to verify. It turned out the porting of the application to SANLP form was very much an iterative one. This also shows that tools such as the Daedalus framework still aren't flawless, and in this case often lacked error messages when generation failed.

3.2 Modifications

The Daedalus framework, at the time of writing, expects all communication to happen within the on-chip memory associated with the interconnect. However, this application has considerable memory usage, due to the large amount of "colour channels" within the image structure, and therefore the communication channels do not fit in the on-chip communication memory. The specific FPGA used did contain a sufficient amount of off-chip DDR memory, and all communication channels have been relocated to that memory. This required quite a few modifications to get working, and still requires manual intervention whenever code is generated.

3.3 The final platform

For this specific project, we initially built a 5-processor platform (excluding host interface) and used a 1-to-1 mapping. The intention of this process was to measure the execution times of the subtasks identified in Section 4, and to make sure the code actually worked on the platform. Using the measurements acquired using this platform, PNtools could then generate a CSDF model, that was then used to generate an optimal platform for the project when using

real-time scheduling. This platform turned out to be a 3-processor platform, which we then used for all further measurements, as explained in Section 3. Darts uses these measurements to create analytical estimates of the execution times based on communication overhead. It then converts the CSDF tasks to periodic hard real-time tasks and applies hard real-time scheduling theory to determine the number of processors in the platform and the task allocation.

4 Scheduling

4.1 Scheduling algorithms

The algorithm as described above was then turned into five independent tasks. These tasks are run on the hardware platform (see Section 3) independently, and possibly in parallel.

Even though the platform used is a multi-processor platform, we still have more tasks than processors, so we need to distribute our tasks over these processors. With the distribution as determined by the Deadalus^{RT} framework (see Section 3.3), we still need to determine how to run multiple tasks on a single processor. We have looked at three different methods: static, dynamic and hard real-time scheduling. As the name might imply, static scheduling follows a pre-determined schedule, and has no work at runtime. The others determine their schedule at runtime.

The intention is to determine the effect that the different scheduling algorithms have on the execution time, and thus throughput, of our program. As all scheduling algorithms produce the same result, and there is no case where a slower program is better, we can then determine which algorithms performs best. Analysis could then show if there are certain code conditions that make certain algorithms perform better or worse. This could then be used to provide recommendations about which scheduling algorithm or code patterns to use.

4.2 Static scheduling

Static scheduling on a multi-processor platform, like used for this thesis, consists of assigning tasks of the algorithm to different processors, and running them in strict order. As an example, with one processor the only valid static schedule for this program would be to run the tasks in order, because each

task needs the output of the former. With multiple processors these tasks can be moved around, but it is important to make sure different tasks run at the right time, and often enough for the program not to stall.

As this is the simplest of all algorithms, namely it consists of just running tasks in order, there is little to no overhead associated with it. If the schedule used is suboptimal, or the application benefits from different schedules at different time, it is very difficult to have a static schedule that matches this behaviour, and the results will also be suboptimal. As especially this program does not suffer from such behaviour, it is likely, especially with a single input, that static scheduling outperforms the other algorithms, simply by lacking overhead. An obvious downside may be that it requires complex design space exploration to determine a task allocation which gives the maximum performance.

4.3 Round-robin scheduling

Dynamic scheduling is mostly similar to static scheduling, except on a single processor no ordering is enforced, and a task is simply run when able. This means that no other task must currently be running (unless a preemptive scheduler is used), and its inputs are ready. It is vitally important that the tasks do their communication in cooperation with the scheduler, so another task can be run when the input is found not to be ready. This roughly means that with dynamic scheduling a task is run on-demand.

In this case round-robin scheduling was used specifically, which simply means that if one task finishes or is preempted the next one in line is called, and after that the next one, and this one is only called again when all other tasks have finished. Yielding is also allowed on this platform, and that means that instead of having time wasted until a time slice has ended, a task can signal the scheduler when it is done or blocked, to allow the next task to run.

An advantage is that this is a highly flexible algorithm, and if tasks are expected to block relatively often, it can quickly switch to another tasks, allowing for better processor utilisation. Of course this does come at a cost, which is the overhead that comes with managing different tasks, preempting them and switching between them. If an application can only follow a simple schedule then it is almost always outperformed by static scheduling, lacking any overhead, but if there is room for variation, the dynamic schedule may utilise the processor better, decreasing the total execution time by decreasing wait times. Similar to the static scheduling, this requires design space

exploration, which is a large disadvantage in most cases.

4.4 Hard real-time scheduling

Real-time scheduling is largely based on dynamic scheduling, but the scheduler is much more rigid, allocating timeslots in which a task can run (when ready). Though this implies more overhead than either of the above methods, it does provide guarantees that neither can provide. Specifically, such an algorithm can specify a minimum throughput which are met as long as the computational demand of all tasks do not exceed what the hardware can provide.

In some problems having those guarantees is a requirement, for instance, in certain control systems code has to be run at certain times, and cannot be delayed. In other instances, like this one, having these guarantees is not required, but a nice thing to have. In this particular case it would give us a lower bound on throughput, so were it used for live video it could give a lower bound on the frame rate. Similarly, it would provide an upper bound for how many images can be sent through a single device, making planning easier, as well as any inter-device scheduling.

It also comes at cost, as mentioned there is a lot of overhead involved. Where statically scheduled programs determine when context switches occur themselves, real-time scheduling is inherently pre-emptive. Even the dynamic scheduling implementation used is based on co-operative concurrency, where each thread determines when to "yield" to another, thereby controlling exactly where this happens, and when this happens. This latter property is especially nice because a yield can be, and is inserted when there is no work to do, so the context switch overhead comes during a time where it is irrelevant.

In real-time scheduling though, to uphold the guarantees, the scheduler is actively interrupting tasks at fixed intervals, and at that point it can then decide, with fairly expensive calculations, what task gets to run next. This means that when no task switch happens, we still have the costs of having to switch from the task to the scheduler, doing the calculations, and switching back. Because these switches involve writing and reading state from memory, these are not cheap switches to make.

In contrast to the other scheduling algorithms discussed, hard real-time scheduling theory can be applied to provide us with the minimum number of processes needed to achieve the target performance, thus removing the need

for design space exploration.

5 Experiments

For these experiments images were used from the New Tsukuba Stereo Dataset, as created for [7] and [8]. Additionally, a selection of images from the Carnegie Mellon University Vision and Autonomous Systems Center's Image Database [9] were used.

The tasks were mapped onto those processors in the following configuration, with the task numbers corresponding to the list in Section 4.

Test case	Scheduling algorithm	Processor 1	Processor 2	Processor 3
1	Static	1, 2	3	4, 5
2	Dynamic	1, 2	3	4, 5
3	Real-time	1, 2, 5	3	4

Table 1: Task mappings on processors

To derive the static and dynamic mapping we used a combination between reasoning and exploration. We have observed that the application executes in linear fashion, and therefore parallelization could best be applied in a pipeline-like fashion. This means that instead of focusing on reducing the execution time for a single image, we increase the amount of images processing simultaneously, thereby reducing the execution time for a set of images. To induce a pipeline-like architecture, we want to make sure our tasks run "in order" for the static schedule, and the dynamic schedule should automatically do the same. To then increase the utilisation of each processor we would like to spread the workload such that every processor takes roughly as long as the others to run all tasks once. If that condition is satisfied, the blocking in communication is minimal.

Due to the measured execution times for each task, this left us with two main candidates to test, corresponding to the two different configurations in Table 1. Experimentally, we found the chosen mapping to be slightly more efficient in the dynamically scheduled case, though the resulting speed difference did not appear significant. In the statically scheduled case we observed that the first processor blocked until an image was completed, to execute task 5, and therefore could not start preparing a new image for the pipeline.

This meant that every disparity map was calculated one after the other, and no pipelining occurred. Therefore the mapping that allowed pipelining, was chosen instead.

If our application had not shown such linear behaviour, the design space would have been much larger, roughly on the order of $5! = 120$ different mappings on three processors. Although this may not seem like a large number, it is still prohibitively large to explore manually, as testing the performance application is a relatively slow exercise. Similarly, when exploring how many processors to use, here given by the amount calculated for hard real-time scheduling, the number of tests quickly grows.

5.1 Expectations

Because of the overhead involved in dynamic and real-time scheduling algorithms, which is completely absent with a static scheduling, it is likely that if the same schedule is achieved, static scheduling outperforms both. Similarly, the overhead associated with real-time scheduling algorithm is generally higher than that of dynamic scheduling algorithms, because there is a lot of algorithmic complexity needed to provide the real-time guarantees.

With regards to the possible gains in finding more efficient schedules, we can distinguish multiple cases, and two important such cases are the processing of single versus multiple inputs. This is mostly interesting because of the highly linear nature of this program. If only one image is processed, then there is no schedule more efficient that linearly running the tasks. Considering this is exactly how the static schedule was constructed, it is with extreme likelihood the fastest algorithm for this case.

Using multiple inputs, however, we can explore the program's pipeline-like behaviour, where tasks can be running simultaneously by processing different images. In this case, all tasks would still run linearly for every image, but we can process one image per processor, each in different stages of the process. It is trivial to construct an optimal mapping for this case, as long as we ignore the possibility of a task stalling a processor because it can't acquire an output buffer, but another task on that processor would be eligible to run. Indeed, the static schedule used is one such optimal mapping, and the other scheduling algorithms can also produce it, but their overhead would slow the application down. That said, if we do concern ourselves about that possibility, there is an opportunity for a more dynamic scheduling algorithm to be faster than the static schedule we use.

As an example, if task 2 completes before task 3 does, and the buffer from task 2 to task 3 is full, processor 1 will stall, even if there is still space in the buffer from task 1 to task 2. Unfortunately, measurements indicate task 2 being the slowest, followed by task 3, meaning we can expect full processor utilisation on processors 1 and 2, and making it unlikely this will occur between task 3 and task 4, the only other place this could happen. Consequently, it is unlikely dynamic scheduling can beat static scheduling.

5.2 Results

We have measured the execution time of the program for different source images, three of which have their results shown below. For every image we have run the program on it one, two, four and eight times, treating the image as consecutive frames in a stream. Note that the application calculates every disparity map anew, and therefore does not benefit from the frames being equivalent. Each of these measurements have been done 5 times, and the average was then used.

Input image	Static schedule	Dynamic schedule	Real-time schedule
Tsukuba (1)	5,367,414,631	3,643,503,098	33,715,138,240
Tsukuba (2)	8,274,774,864	5,838,259,599	40,458,165,888
Tsukuba (4)	14,083,463,254	10,033,093,726	53,944,221,184
Tsukuba (8)	26,841,066,482	18,425,867,237	80,916,331,776
Cube (1)	4,009,036,043	2,622,098,508	33,715,138,240
Cube (2)	6,698,294,268	4,174,675,542	40,458,165,888
Cube (4)	11,197,985,495	7,151,535,871	53,944,221,184
Cube (8)	21,193,963,457	13,094,576,846	80,916,331,776
Pentagon (1)	4,197,345,552	2,811,664,125	33,715,138,240
Pentagon (2)	6,775,930,632	4,537,172,095	40,458,165,888
Pentagon (4)	11,341,453,989	7,841,951,368	53,944,221,184
Pentagon (8)	20,369,076,223	14,425,084,576	80,916,331,776

Table 2: Average execution times in cycles with n input images

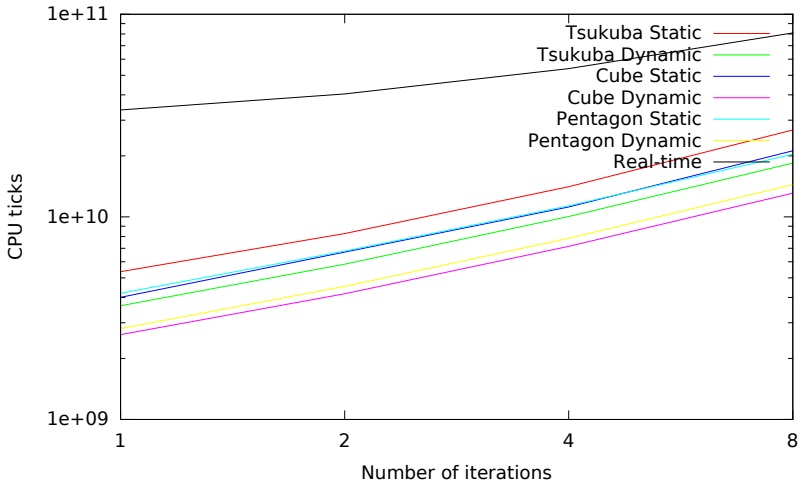

Figure 4: Measurements

5.3 Observations

After executing just the first round of measurements, corresponding to `Tsukuba (1)` in table 2, it was quite a surprise to see the dynamic schedule having a much higher performance than the static schedule. We propose a theory about the inter-processor communication code in particular that could explain this in Section 5.3.1.

5.3.1 Memory saturation

Aside from the obvious differences between the scheduling algorithms themselves a lot of code is shared between the different versions. One of the few places this is not the case, is in the code for inter-processor communication. In particular, the static scheduled version simply waits for a message to appear on a channel, where the dynamic scheduled version explicitly reschedules when no message is available. Even though no thread may have work ready and this would still effectively be a loop checking for availability, the overhead involved in switching threads would mean a wildly different memory access profile.

If we assume the DDR memory module, used for all image calculations, is saturated, that is, either it is handling as many requests as it can, or the bus connecting it to all processors is, something as simple as a different memory

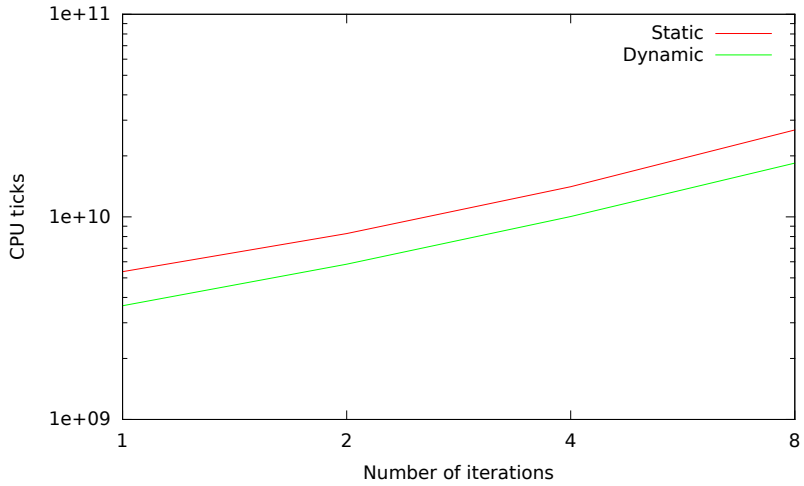

Figure 5: Tsukuba measurements

access profile can severely impact memory. Because the static version issues a lot more "useless accesses", by virtue of it being a faster loop, this would mean that a larger portion of memory accesses are useless. If the memory is also saturated, this means that a backlog of accesses is created, that contains a lot of useless requests, preventing "useful" requests from reaching the memory in a timely manner.

There are several things we could then expect in further measurements, with the assumption the memory is saturated with the statically scheduled `Tsukuba (1)`. If these useless requests play a large role, we would expect the performance of the statically scheduled to increase, because of a pipelining causing the processors to have higher utilisation, and therefore send less of these requests. It is interesting to note that this would start playing a role with more than two images, since the first and last don't benefit from pipelining as much in our three-stage pipeline. Unfortunately it is hard to say if this will be measurable against the performance gains involved in pipelining itself.

Then we can also distinguish two cases for the dynamic schedule. If the memory was not saturated in these measurements, we would likely find a case with pipelining where it would be saturated, causing a drop in performance at that point. In the other case, we would expect to see performance gains much like in the static version, but to a lesser extent, since it also impacts this version less.

The further measurements seem to match these expectations, specifically, there is a slight change in the incline of the lines, where the dynamic schedule is worse off. It is hard to confirm whether this is the cause, because the change is extremely small, and likely hard to see on paper.

5.3.2 Linearity

The graphed results clearly show a largely linear nature (on the log-log scale), indicating that, in general, doing twice the iterations also implies twice the duration. This is exactly what is expected to happen, and it is also interesting to note that the linear and dynamic schedules seem to roughly grow equally. This equal growth would imply that the performance difference between the two schedules is not only based on constant costs, like set-up, but runtime costs also scale up.

5.3.3 Bottleneck

Although not present in the recorded data, and not accurately measured, the execution time of the second processor, closely followed by the third, dwarfed that of the first. In particular, the third processor would finish slightly before the second, for every iteration (with each processor working on a different iteration), and the first processor would finish long before both. Seeing as this would have negative impact on the effect of pipelining, which benefits from having each stage/processor be of equal duration, it is likely that this causes the minimal performance gains observed. This might be a subject for further research.

5.3.4 Image size

As seen in Figure 4, even with a single scheduling algorithm there are performance differences. The largest factor in this seems to be resolution, all images were scaled to fit in 192×144 pixels, but did not have the same aspect ratio. Of the three shown, only the `Tsukuba` image has the full 192×144 pixel resolution, and the others were 144×144. Because the same parameters were used, an image of the same size should yield roughly the same performance, and that is indeed what we see.

6 Conclusion

The results indicate that dynamic scheduling outperforms static scheduling for our application. We have also observed that design space exploration is easier for dynamic scheduling than it is for static scheduling, because the order of tasks on a processor is of lesser concern. Although, because of reasons outlined above, these results can not easily be generalised to all applications, it can most likely be applied to applications with similar memory requirements, which extends to most image-based algorithms.

Unfortunately, we have not been able to get actual measurements of hard real-time performance, and only have a worst-case approximation. Our observations regarding design space exploration, or lack thereof, still hold, indicating that in any case the development time of an application with hard real-time scheduling can be drastically shorter than for any of the others. Since we applied the platform as designed for hard real-time scheduling to our project as a whole, this benefited our exploration process even for the other scheduling algorithms.

The calculated worst-case performance of the real-time schedule is about 10x worse than the dynamic schedule for a single image (see Table 2). Even though we expect high overhead, and this is an over-approximation, that is still much worse than the alternatives. Another factor that comes into play is image size. Since our analytical performance measure never dealt with the actual input images, it could not determine the actual runtime of a step. The allocated image is larger than any of the measured images, so we would expect worse performance when full-size images were used. Given the performance decrease we have observed due to image size, we assume real-time performance to still be worse than any of the others, but significantly less so.

We have therefore seen that performance-wise, using a round-robin, non-preemptive dynamic scheduler provides the best results for applications with high memory usage, such as computer vision applications. In the program design and porting stage, however, we have found that using the Deadalus^{RT} toolchain provides us with a much faster workflow, even assisting our development of ports for other scheduling algorithms. Given these factors, we advise anyone developing similar applications to generate a hard real-time scheduled version and base a dynamically scheduled version on the results obtained from it if extra performance is necessary.

6.1 Further research

Due to runtime issues with the generated real-time code, it was sadly impossible to get measurements for it, and only analytical approximations were used. Analytical calculations use the worst case execution times estimates which are not very likely to happen during the real execution of an application on a hardware platform. The overestimation of the total execution time of an application comes from these worst case estimates. However, analytical estimates are a very useful tool for giving the early information on the lower bound of the throughput of an application, and to determine the number of processors in the platform without doing time-consuming design space exploration. Validating these approximations would be beneficial for the validation of any conclusions drawn. Similarly, these results were only found using one specific application, so further analysis is in order on other applications.

This particular implementation also had a few design choices that could drastically impact performance. It could, for instance, be beneficial to duplicate heavy tasks onto multiple processors, to increase the throughput. As is, the parallel execution, specifically of the first processor, was severely limited.

References

[1] C. Lawrence Zitnick and Takeo Kanade. A cooperative algorithm for stereo matching and occlusion detection. *IEEE Trans. Pattern Anal. Mach. Intell.*, 22(7):675–684, July 2000.

[2] Mark Thompson, Hristo Nikolov, Todor Stefanov, Andy D. Pimentel, Cagkan Erbas, Simon Polstra, and Ed F. Deprettere. A framework for rapid system-level exploration, synthesis, and programming of multimedia mp-socs. In *Proceedings of the 5th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '07, pages 9–14, New York, NY, USA, 2007. ACM.

[3] Sven Verdoolaege, Hristo Nikolov, and Todor Stefanov. pn: A tool for improved derivation of process networks. *EURASIP Journal on Embedded Systems*, 2007(1):075947, 2007.

[4] M.A Bamakhrama, J.T. Zhai, H. Nikolov, and T. Stefanov. A methodology for automated design of hard-real-time embedded streaming systems.

In *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, pages 941–946, March 2012.

[5] H. Nikolov, T. Stefanov, and E. Deprettere. Systematic and automated multiprocessor system design, programming, and implementation. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 27(3):542–555, March 2008.

[6] IEEE. Ieee standard system c language reference manual. *IEEE Std 1666-2005*, pages 1–423, 2006.

[7] Sarah Martull, Martin Peris, and Kazuhiro Fukui. Realistic cg stereo image dataset with ground truth disparity maps. In *ICPR workshop TrakMark2012*, volume 111, pages 117–118, 2012.

[8] Martin Peris, Atsuto Maki, Sara Martull, Yasuhiro Ohkawa, and Kazuhiro Fukui. Towards a simulation driven stereo vision system. In *Pattern Recognition (ICPR), 2012 21st International Conference on*, pages 1038–1042. IEEE, 2012.

[9] Carnegie Mellon University. Cmu vasc image database. `http://www.ius.cs.cmu.edu/idb/`, 2003.