A parallelization model for embedded applications

Master student: Jerome Reybert*†
Supervisors: Jean-François Mehaut*, Miguel Santana† and Carlos Prada*†
* Mescal, Avenue Jean Kuntzmann, 38330 Montbonnot-Saint-Martin
† STMicroelectronics, 850 r Jean Monnet 38920 CROLLES

***Abstract*—The previous years have seen an important evolution in the embedded system domain, with the apparition of multi processors and multi cores architectures. For the moment, these architectures are embedded specific, built with one processor and several accelerators around it.**

The next ones will be based on many processors and core/accelerators. The software development should change as developing for a parallel system is completely different from a mono processor one. Furthermore, the software directory of industries such as STMicroelectronics must be adapted, in order for the applications to benefit all the available computing power of multi-core systems.

We propose to define a simple methodology to manage the parallelization process of sequential and embedded applications.

I. Introduction

Embedded systems manufacturers must now evolve to multi-processor and multi-core architectures. The whole catalog of their software should be able to exploit these parallel architectures. It is impossible to rewrite all of them from scratch. The held approach is to begin from the sequential source code, and to transform it into a parallel version: to parallelize it.

Parallel development needs important and specialized development skills. For instance, developers specialized in image processing, should not have enough expertise to parallelize efficiently themselves their algorithms. Two alternatives exist:

- to provide to image processing experts some tools to give them a simple parallelization method.
- to give the opportunity to a parallel expert an opportunity to parallelize an algorithm in a domain he is not comfortable with.

Moreover, embedded system domain is confronted to many specialties: signal processing for communication applications and video processing for compression / decompression are common examples. The upcoming computation resources provided by the parallel processing appearing technologies open the opportunity to new domains such as security : finger print recognition, face recognition...

It seems difficult to hire one expert for each cited previous domains. Moreover, even if we can provide them efficient parallelizing tools, they should as well be aware of specific parallel programming characteristics. For this work, we are interested in the other solution: a parallel system expert, working on all these various domains. He can not understand all the specialties he will face. We must give him the opportunity to parallelize the code without having a deep understanding of this one.

This article is structured as follows. In section II we briefly present existing tools to observe sequential code and section III shows techniques to implement parallel code. Then in section IV we propose our methodology to manage the parallelization. Section V presents an example of parallelization following our methodology. The section VI gives a conclusion.

II. Understanding and optimizing performance

A. The existing observation techniques

There are several techniques and tools to gather information about the execution of a program. The different techniques can be gathered in four categories: *1*) source code *2*) compiler *3*) binary translation *4*) sampling techniques .

Intrusion: This point must be taken into account to compare the analysis tools. The resulting data is important but we must also consider the intrusion aspect of the observation. Too much intrusion can disturb the program execution and produce erroneous observation. This intrusion must be carefully thought, especially when the source code is instrumented manually. If we use existing tools, the intrusion should have been already thought. In this case the intrusion should be predictable, and in some cases, corrected.

We are now going to compare the different techniques.

1) Source code: Gathering information in order to understand the program behavior is a common task that every programmer has already done since he is a student. The common error is trying to gather all the information manually. It will seem the easiest way, using well known primitives as “gettimeofday” and “printf”. However, the developer is confronted to several problems:

heterogeneous hardware From one specific platform to another, counters accesses can be completely different. The access method to the counters should be rewritten for every platform.

data representation The way to save and present the gathered data is important. Maybe the programmer will be satisfied by the form of its traces, but it will be difficult for the other to read it. A specific format implies to develop the visualization tools for this given format.

As we can see, reinventing the wheel each time we want to trace a program execution can have important consequences. There are some tools which can help the developer to gather information:

PAPI : Performance Application Programming Interface, *PAPI* provides predefined high level hardware events summarized from popular processors and direct access to low level native events of one particular processor. PAPI is fully portable to a large number of architectures. But it does not solve the trace format problem.

TAU : Tuning and Analysis Utilities[1] is a set of tools providing easy ways to settle timers. TAU provides simple accesses to PAPI events. TAU is a good choice for manually tracing an event, as we don't have to worry about memory and files. It also provides a standard format for the traces. The resulted traces can be used in some visualization tools. TAU also provides some trace format converters, in order for the traces to work with several visualization tools.

OTF : Open Trace Format is is an XML standard format for traces. Developed for tracing parallel programs, it fits the sequential requirements, and is still in active development. To use this format will make the traces portable among several visualization tools.

Some events are too specifics to be automated, that's why manual tracing is still important. But for most of the events, tracing becomes quickly a repetitive task. We will see that an implicit tracing can be used for a majority of primitive events. Furthermore, getting a global point of view of the application seems unrealistic, tracing this one manually.

2) Compiler: As a program can be very large or very complex, to understand the program reading the source code might not be suitable. It is even more difficult if you are not familiar with the project. The goal here is to dissect the total execution time among all the functions. Locating the parts where the program spends most of its execution is a good start. The idea is simple, to start a timer at the beginning of each function and to stop this timer when the function returns.

Bell laboratories early began to work in this direction, adding a tool named prof[2] in UNIX system. Rather than recreating a tool to parse and analyze the source code, this tool is located in the compiler. During the compilation process, timers are placed at the beginning of each functions, and before each return statement. This work is made easier thanks to the parsing already made by the compiler. During the execution, each time the program steps into a function, the relative counter is incremented and the timer starts. When the execution flow exits this function, the time spent in this function is added to the total execution time of this function.

prof has then been improved into *gprof*[3]. In early 80's, a team working on Unix had some problem understanding the time spent in some functions, as these functions where primitive ones, and called from different places in the program. The global time spent in a function was not relevant enough. Instead they preferred to split this time among the different places from where the function was called. A more complete description of gprof is done in a following chapter.

TAU, used with the framework Program Database Toolkit (PDT[4]), is able to analyze the source code. Using compiler information, it adds some profiling routines. TAU and PDT are more complex than gprof to settle, but is more tunable in return.

This kind of trace is not so intrusive. The memory accesses are limited, as the profiler does not collect all the times spent into the functions, but just a sum of each. Furthermore, these times are stored into hashtables. The access time to a particular counter function does not consume too much.

Some can say that adding some instrumentation code could produce observation bias. As it adds function calls, it could change the way the compiler applies its optimizations. In our case, profiling for parallelizing, we are not expecting so strict analysis. On the contrary, we would rather compile without optimizations.

Furthermore, to produce profile like this is very simple. Most of the time, simply add an option into the compiling command line is enough (the option for gprof in gcc is `-pg`). gprof is often a good start to discover a program.

3) Binary translation: The program can also be profiled in its binary form, without having to compile it with profiling methods. At a first glance, it seems a valuable method only for programs which we do not have the source code. But as it would be useless, we will see that this technique brings other opportunities. To profile an application in a binary form can be done in two ways: statically or dynamically.

a) Static: There exists some tools as MAQAO[5] which are able to extract knowledge from a compiled program. The idea here is to parse the assembly code. This restrict the tool to analyze code for a specific platform, and even compiled by a given compiler. In the case of MAQAO, it works with Intel platform binary code, compiled with ICC. Static analysis can begin to extract call graph, control flow graph, list the loops and so on. With all that information, some analyzes can already be done.

But static analysis often provides irrelevant results. e.g. statistics about branches are only statistics. Caches effects are not predictable. MAQAO sticks up to fill the gaps with dynamic analysis. It will insert some instrumentation code directly into assembly code. To profile at this level allows to access to interesting values: function parameters, addresses used in load/store instructions…

In binary translation, and in all other form of profiling, static analysis can not be enough. The good aspect here is to use results of both static and dynamic analyzes. Moreover, there are good ideas, such as observing the distribution of parameters values for each function. However, this tool is clearly dedicated to profile an application in order to to optimize the sequential version. Furthermore, we are restricted to a given platform and compiler.

b) Dynamic: Intrusion during execution can modify the results: the trace functions added, the resulted disturbation during the optimization phase of the compilation, the access to the memory to store the trace results, … As we can try to minimize this as much as possible, it will never be possible to reduce the intrusion to zero. Another considered solution is to simulate the execution and do all the trace work into the

simulation.

Here, the tool is a kind of virtual machine dedicated to tracing, using just-in-time compilation techniques. As the execution is emulated, the resulted overhead does not disturb the execution itself. The most famous tool is *Valgrind*[6]. The conception of this tool allows the creation of external ones:

- memcheck for memory allocation profiling (in order to spot invalid allocation or deallocation)
- helgrind, tool capable of detecting race conditions in multithreaded code
- cachegrind[7], and its evolution callgrind, able to produce impressive call graph, to profile instruction cost and cache misses.

Virtual machine trace tools are very powerful. Thanks to the simulation side, it is possible to get very precise information, for instance about caches, number of primitive instructions per line. These informations are difficult to gather when executing on a real platform, as all these counters can be accessed in different ways depending on the platforms, in the case they exist. The simulation of an execution will be exactly the same, no matter the platform. Furthermore, the traced execution is not disturbed at all, as the tracing instructions are embedded in the simulator. The downside is that there is a huge overhead for the real execution time. Depending on the options passed to the simulator, the execution time can be 20 times to 200 times longer.

4) Sampling technique: All the following techniques alter the source code, or the binary code in order to extract some information. They insert some special code at he beginning and the end of the methods, with timing information, to profile it. Another way to profile an execution is to sample it. A sampling profiler observes the program counter at regular intervals. The results are not numerically exact, but a statistical approximation. It could seem to produce less interesting results. But in practice, these results are more accurate:

- this profiling method is not intrusive to the target program, and do not produce side effects as memory cache misses or breaking code pipeline
- it does not over-evaluate small and often called functions
- the program can run at nearly full speed

As this kind of profiler works at a close level with the hardware, the tools are often hardware related:*VTune* for Intel, *CodeAnalyst* for AMD, *Shark* for Apple. Other tools can be used without platform concern, as *Oprofile*[8].

5) Discussion: Two aspects must be considered to choose the appropriate analysis tool: *1*) the efficiency of its analysis, i.e. the trace tool should not be too intrusive and in the same time provide accurate analysis; *2*) the utilization should not require too much effort, in order to not to spend too much time in the analysis phase. Valgrind is a good candidate, as it fills the two preceding points. Even if the simulation can take a long time, the execution is not disturbed by the tracing tool. Furthermore, the program can easily be traced without any modification.

B. Parallel programming model

We have seen in the previous section how to analyze a sequential program in preparation for its parallelization. There are lots of techniques to implement a parallel program. In this part we will discuss some of the existing solutions.

1) How to express parallelism?: A parallel programming model is a set of software technologies to express parallel algorithms and match applications with the underlying parallel systems. It encloses the areas of applications, languages, compilers, libraries, communication systems, and parallel I/O. The targeted architecture is important, as the tools, and even the programing design depends on this.

It must be noted that the final implementation will be done using specific embedded system API. For our approach, we prefer to firstly implement on general purpose architecture, in order to simply implement, test and debug them. Once the parallel algorithm is approved, it is transformed in order to work with embedded system API.

The targeted architecture here is SMP (symmetric multiprocessing); this architecture involves a multi-processor or a multi-core computer where two or more identical processors can connect to a single shared main memory through a bus. In this part we will only consider SMP parallelization solutions.

a) Language specific: Parallelism can be expressed in several ways. Specific languages have been designed for parallel programing such as Ada or Cilk[9]. But language specific can not be considered, as most of the embedded systems software source code is written in C.

b) Auto-parallelization: There exists a large literature about auto-parallelization. Both popular compilers GCC[10] and Intel compiler try to implement auto-parallelism. The idea here is, with a simple compilation flag, detect parallelism opportunities in sequential source code and provide a multi threaded application. However, the analysis provided by these parallelizer is only static. It has been shown[11] that, in a general manner, static data dependence analyzers are not powerful enough to prove that some loops are data dependences free on large programs.

Other works such as Thread Level Speculation[12] or Automatic Profile-Driven Parallelization[12] provide a dynamic data dependence analysis. TLS has an interesting approach. The source code is instrumented during the compilation. It introduces an “executable intermediate representation”: that means the profiling is more aware high level code (like loop definition) and data structures. The program is executed several times, with multiple input datasets. For each execution, the profiler generate a trace file, which is analyzed to detect data dependences. If there are code regions which can not be disambiguated by this analysis, the program is recompiled. The instrumentation is done more finely on the disambiguated areas. Once the program has been fully inspected, OpenMP directives are added for areas which have been approved to be parallelized.

But this kind of tools are still in research area. Their efficiency is not proved, and it is hard to find implementations. However, this lead should give results in the next years.

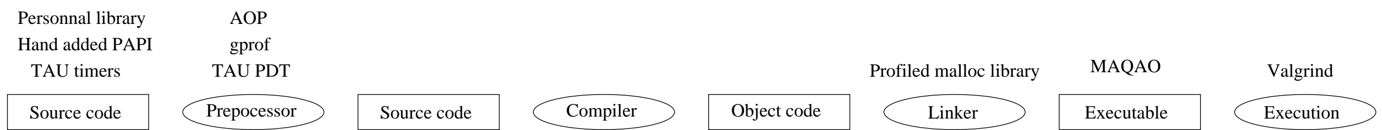

Fig. 1. An overview of the existing possibilities to add profiling and tracing methods during the program life.

2) Software programming language: In the prospect of C programming, we will present existing APIs. The most famous is certainly *pthreads*[13], the POSIX thread standard for threads. It is available for Unix systems, but also for Microsoft Windows with *pthreads-w32*. It is a low-level interface that allows users to create and manipulate threads (lightweight processes). All the threads of a single process share the same address space which does not need any explicit communication. On the other hand, it is the user responsibility to protect variables from concurrent accesses with synchronization mechanisms (monitors and conditions). Programing with pthreads is not very difficult. However, even to achieve simple tasks, coding turns out to be difficult and repetitive.

a) OpenMP: OpenMP[14] is an easy-to-use parallelization portable interface which is based on a shared memory approach. Programming with OpenMP consists in inserting directives to the sequential code. Theses directives are used by the compiler to generate the parallel code for the processors and cores. Its closeness to sequential programming makes it wildly used for parallelization of existing sequential algorithms.

The section of code that is meant to run in parallel is marked accordingly with a preprocessing directive. At this point, the main thread forks into a defined number of threads. Each thread run concurrently. After the execution of the parallelized code, the threads ”join” back into the master thread, which continues onward to the end of the program. The number of threads can be defined before the execution or specified for a specific parallel region in the source code.

Two kind of parallelism can be defined with OpenMP.

- Data parallelism: this kind of parallelism often appears into loops. A simple example is a for loop doing calculation on a matrix. The matrix is split into several memory areas, and each thread will execute the same code on its own memory area.
- Task parallelism: in a parallel region, we define several tasks to run jointly. Each thread will execute independent code.

For these parallel regions, it is possible to define more precisely the OpenMP compiling behavior. We can specified which variable are *private* and *shared* among the threads. A data parallelism region can be scheduled in a *a*) *static* way, which means the memory is split into *chunks* among the threads before the loop execution. There are as many chunks as threads *b*) *dynamic* way, here the chunks are smaller. There are more chunks than threads. When a thread finish to compute a chunk, it gets another one. It is designed to minimize load balancing issues *c*) *guided* way, which looks like to the dynamic scheduling. The difference is that the first allocated chunks are bigger, and the next are smaller to smaller. This scheduling can be even more efficient for load balancing issues.

OpenMP also offers some directives to easily specify parallel behaviors like *critical section* or *barrier*.

It can be noticed that OpenMP is implemented in popular compilers such as GCC or Intel compiler. It allows to simply compile OpenMP directive adding a flag during the compilation.

3) How to check parallelism?: Data races and dead locks are notoriously hard-to-find threading errors. Such unsynchronized memory reference causes non-deterministic behaviors. Usual debugging tools are inefficient to detect such problems.

Such tools are today available. Helgrind[15] is a thread debugger which finds data races in multithreaded programs. It looks for memory locations which are accessed by more than one thread, but for which no consistently used mutex lock can be found. Such locations are indicative of missing synchronization between threads.

Another one is Intel Thread Checker[16]. Thread Checker is involved at different phases. Compiling a source code with *icc* and the flag “`-tcheck`”, Thread Checker instruments the code during the compilation. Then, the application is executed with `tcheck_cl` command. This program firstly instrument the binary code, then supervise the execution. It instruments every memory reference instruction and every thread synchronization primitive in the program. When the instrumented program is executed, the runtime analysis engine monitors every memory reference event and every thread synchronization event and analyzes if there is a data races. Thread Checker supports Posix Threads and OpenMP. The drawback is that to benefit all the Thread Checker features, the source code must be compiled with *icc*. Some modification for the compilation may be needed in order to be compatible with *icc*.

4) Discussion: We have seen that automatic tools are still not mature enough to provide industrial results. OpenMP seems to be a good compromise: it is simple to express parallelism, widely available, and the OpenMP behavior is simple to reproduce with native thread API. An interesting lead may be to extract the intermediate code produced by the OpenMP. It could make the translation work to native thread easier. If some problems are encountered during the parallel execution, the use of tools like Intel Thread Checker is really encouraged.

III. Definition of a parallelization methodology

We are now able to trace the execution of a sequential softwares, with tools such as Valgrind and TAU. We can extract from these traces information to point out on which part of the source code we must focus in order to parallelize this software. We have also seen solutions to implement parallel algorithm.

A methodology must be defined to efficiently use these tools, and manage the process of parallelization. This methodology is inspired from software engineering discipline. This connection between software engineering model and parallel programing has been first discussed in a white paper from Intel[17]. We will first briefly present some of the development models defined by the software engineering. Then, we will transpose this method for our parallelization problem.

A. Software engineering models

There are lots of methodologies to manage a project. This subject is still an open software engineering topic. A methodology can be defined as a framework. This framework defines the stages involved in the development of the software: in which order should we consider the stages? when can we tell that a stage is completed? how to make the transition between two stages? what could force us to go back to the previous stage? The first related framework to manage software development has been proposed in the 70's by [18].

A standard framework for developing software consists on:

requirements capture: This task's goal is to determine the requirements for a new system or to modify an existing one. This step must take into account the possible conflicts which could exist between different stakeholders. Requirements capture is a critical step to the success of the project.

analysis: All the requirements are analyzed, classified. The road map of the project is defined, as well as the required work team. The platform and the programing language are chosen during this phase.

design: After software's purposes and specifications are determined, software developers will design a plan for a solution. It includes low-level component and algorithm implementation issues as well as the architectural view.

implementation: The purpose of programming is to create a program that exhibits a certain desired behavior (customization). The process of writing source code often requires expertise in many different subjects, including knowledge of the application domain, specialized algorithms and formal logic.

testing: Software Testing is an empirical investigation conducted to provide stakeholders with information about the quality of the product or service under test, respecting the context in which it is intended to operate. Software Testing also provides an objective, independent view of the software to allow the business to appreciate and understand the risks at implementation of the software.

deployment and maintenance: Deployment starts after the code is appropriately tested, approved for release and sold or otherwise distributed into a production environment. It may be necessary to add code that does not fit the original design to correct an unforeseen problem or fill a customer request.

There are several ways to order these steps. The basic one is the *Code-and-fix* model. Actually, it is more an absence of model. The developer just codes an initial version of the application, doing the analysis at the same time, and fixing the problems when he encounters one.

The first well designed model has been the *waterfall model*. To follow the waterfall model, one proceeds from one phase to the next in a purely sequential manner. For example, one first completes requirements specification, which are set in stone. When the requirements are fully completed, one proceeds to design. The waterfall model is argued by many to be a bad idea in practice, mainly because of their belief that it is impossible, for any non-trivial project, to get one phase of a software product's life cycle perfected before moving on to the next phases and learning from them.

Another one is *iterative development*: it slices the deliverable business value (system functionality) into iterations (also called increments) at the beginning of the project. In each iteration a slice of functionality is delivered through cross-discipline work, starting from the model/requirements through to the testing/deployment. This model is more flexible.

Our methodology has been inspired by this iterative model.

B. Parallelizing model

To parallelize a software is a new form of coding. A "classical" project often begins from scratch, or using libraries. In some cases it can work upon an existing software, in order to add some functionalities or correct some errors. In our case this is different. We do not want new functionalities, the program is supposed to fulfill the requirements.

As there are not already any efficient tool to detect and correct data dependencies, we must adopt a *code-and-test* approach. Using this approach without any road map improves the risks of failure. This is the main reason to define a software development model. Our model is inspired by the *iterative* one, with an implementation - testing - debugging phase flighty different.

1) Requirements capture: This step does not have the same importance as in a classical project. As the software already exists, we don't have to define its requirements. The main requirement remains the same, to improve as much as possible the resource utilization on a multi-core platform.

2) Analysis: The analysis is done on the sequential version of the code with the performance analysis tools presented in section II in order to discover hot spots. They will represent the different iterations of the development cycle. We must also try to define the maximum reachable speed up. We will use for that the Amdahl's law, that can give us the maximum speedup from the portion of sequential code. The goal here is not to reach the maximum theoretical speed up. It can help us to understand the parallelization. If the theoretical speed up is low, we can not pretend to obtain good performances.

3) Design: There are some typical patterns in the source code for the parallelization design. Some of these patterns have been presented in section III. The call graph produced during the analysis iteration can give us interesting leads. The higher in the tree we will try to parallelize, the better should be the performances. On one hand, overhead dues to thread creation and barrier are less important. On the other hand, the risks to have data dependencies are higher.

During this phase, we must evaluate if the thread management overhead is covered by the execution time spend into the parallel region.

4) Implementation: The implementation, as discussed in section III, can take several forms. In the first iteration of the development cycle, we will use OpenMP. OpenMP offers very simple ways to implement parallel code. As a downside, performances may be worst than with native thread implementation, as *pthread*. Our goal here is to test a design. Once a design has been approved, it can be implemented with low level techniques. This is discussed in the enhancement iteration.

If the parallelization seems too difficult, involving too much changes in the algorithm or if the debug iteration shows that the data dependences are too strong, it should be considered to go back to the design iteration, and rethink it, going upper in callgraph tree.

5) Testing and debugging: The parallel implementation is then executed. For this phase, test unit sets are really important, to ensure the produced data is correct.

The number of cores used for execution is not defined. The more threads are running, the better are the chances to reveals parallel problems. There can be grouped in two categories.

- data races: they arise in software when separate processes or threads of execution depend on some shared state. Operations upon shared states are critical sections that must be atomic to avoid harmful collision between processes or threads that share those states.
- deadlocks: it refers to a specific condition when two or more processes are each waiting for each other to release a resource, or more than two processes are waiting for resources in a circular chain.

Tools such as *Thread Checker* Intel tool improve greatly the research and the resolution of bugs such as data-races, deadlocks situation and so on.

6) Enhancement: This iteration has been added from the classic software engineering model. Once the testing and debugging iterations are successfully completed, we can try to optimize some parts of the parallelization code, using for example native thread mechanisms. Reproduce an OpenMP behavior should always be possible.

In the case of embedded system, a program can not be compiled with OpenMP. It makes this iteration obligatory. It seems better in this case to consider this iteration once all the increments have been successfully parallelized.

7) Discussion: The iterations are now defined. We present them in a more comprehensive way in figure 2. This diagram shows more explicitly the interactions between the iterations.

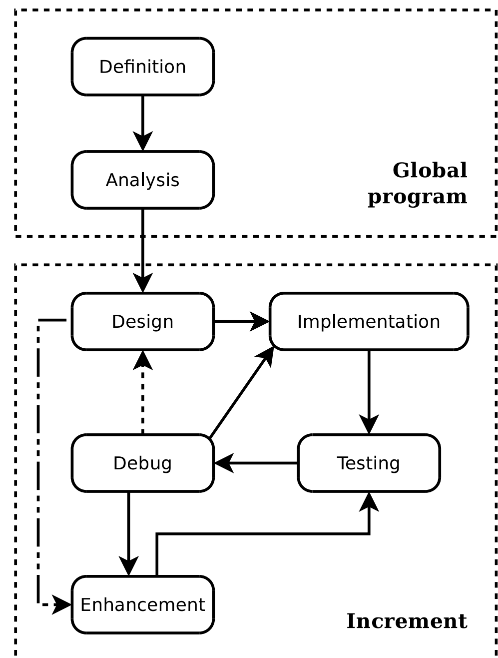

Fig. 2. Software development model for parallelizing project

As we can see, the definition and the analysis are made only once, at the beginning of the project. The analysis iteration defines the increments. These increments are managed one by one, supposing that they are independent. For each increment, we begin by trying naive approach, then we observe how the program behaves. In some cases, we should reconsider the design. Once the program is correct, we can go to the next increment, or try to enhance the current one.

IV. Application of the methodology on an embedded system software: TNR

TNR means Temporal Noise Reduction. The goal of this application is to reduce the noise on all the frames composing a video stream, taking into account some temporality.

A. Presentation

1) What is denoising: Noise is often present on video image. This noise can have several sources. It can come from the capture of the video like random brightness variation or color information produced by the sensor of a digital camera. It is also present on analog image recorded with a photographic film having a too high light-sensitivity. Noise can also appear during the compression of the video and the transmission of the data.

All this noise is unwanted as it deteriorates the image quality. But cleaning noise means removing information. Applying a too rough denoising method could alter the image, loosing some information, making it blur. It is also important to provide a certain consistency between the different frames, to prevent some video blinking effects.

2) Spatio-temporal noise reduction: Spatio-temporal denoising, as it is written, means denoising using two factors: space and time. The spatial denoising is the classic method. It consists in analyzing an image by looking for important differences between pixels and its neighbors. When it finds a pixel with an high contrast comparing to its neighbors, it applies a transformation to the color of this pixel, about the mean of the neighbors color. Spatial denoising is applied on each video frame as independent images.

The temporal denoising is the next step. We perform a motion detect filter and a temporal filter on a frame, taking into account the adjacent frames. To simplify, we get the previous denoised image, and apply it to the current image like a filter. On the actual image, we will perform the spatial denoising, taking into account the previous image filter. It allows to apply almost the same denoising method on this image, limiting a blinking effect.

3) STMicroelectronics implementation: The TNR application must be seen as a part of a larger program for video decoding. In embedded systems, it is between the video decoder and the output stream. The version used for the parallelization test is a standalone one. It means it can be compiled and executed on a computer. The input streaming (coming from the decoder) and the output streaming are simulated as reading and writing files on the hard disk.

The TNR application takes as input parameters the path where images to be denoised are stored, the path to store the denoised images, and the number of images. Each single image is stored in 3 different files: one file for each channel Y, U and V. This behavior reflects how the program works in the global decoding software. The images to be denoised have already been decoded (e.g. by an H264 algorithm). The image is not compressed at all, in broad outline $1pixel = 1octet$. For a video, each file will have the same resolution and the same size. This should produce a predictable execution.

For each image, the program will work on even then on odd lines, as two separated and independent images. As there is not complete documentation about the TNR application, we only can guess why the denoising operation is split among the lines. We can deduce that this denoising algorithm is done for an interlaced video (like common analog television signals). The result should be better for this kind of output.

B. Use of the model

The previous documentation work has been done to present the TNR algorithm in this paper. We must consider that the developer in charge of parallelizing such program has not this information.

1) Requirements capture: TNR is a well known STMicroelectronics algorithm. The goal here is not to reach extreme performance, but to show the validity of our parallelization model. The second objective is of course to obtain the same denoised image quality. As it is difficult to judge the image quality, we want the exact same images. We will test the resulted image set, comparing byte per byte with *diff* tool, these images with an image set produced by the sequential version.

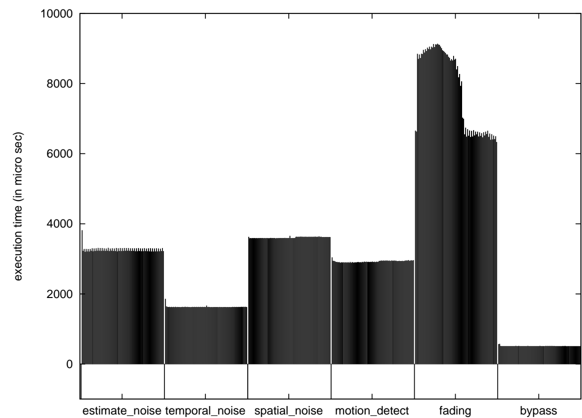

Fig. 4. The execution time among the iterations seems to be stable for each function.

2) Analysis: We begin by tracing the sequential version of the program. For that, we prefer to use Valgrind. As we have seen above, Valgrind is preferred because it is not architecture or compiler dependent, and measurements are still accurate.

We execute *TNR* with a set of 41 images. The result of this execution is presented in figure IV-B2. Nearly all the execution time is spend in 6 functions: `bypass`, `estimate_noise`, `temporal_noise_reduction`, `spatial_noise_reduction`, `motion_detect` and `fading`. This kind of callgraph form suggests us that these 6 functions are executed ones after the others in a loop. There are called from one function, `c9nr_top`. This should make the parallelization easier, as the 6 consuming functions are called in the same function.

We can notice that `c9nr_top` is called 82 times, whereas there are 41 images. A further look into the code shows us that the image file is split into two buffers: one for the even lines, and another one for the odd lines. It is difficult to guarantee, but it seems that the computation is independent for both buffers.

We can extract an interesting information from this callgraph. There is the execution time passed in each function (in percentage) according to the total execution time. We can deduce from it that a naive approach should have poor results, because the load balancing would be a disaster among the functions.

We tried to measure more accurately the execution time of the 6 functions among the iterations. The results of this trace are shown in figure 4. We can observe that among the 82 iterations, the execution time of each function is stable, except maybe for the `fading` function. However, this stability is a good sign for the parallelization, as the execution is predictable. We will see the consequences for the design in the next paragraph.

As we only focus on the `c9nr_top` function, we can ignore the time taken by processes like loading and storing the images. We can not parallelize these processes, as

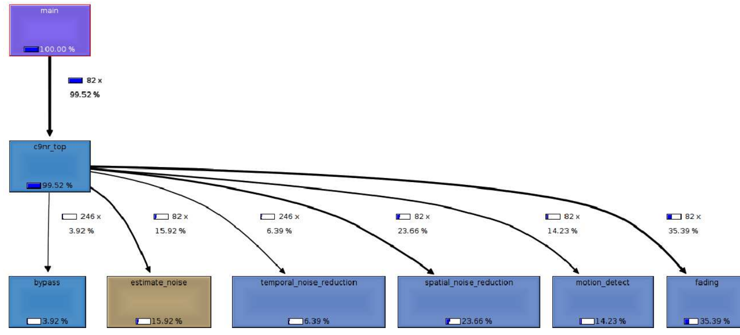

Fig. 3. The result of the TNR execution traced by valgrind. The 6 main functions appear clearly as leaves.

they represent the data transmission in the real situation (in the embedded system). We can now almost distinguish the parallelizable sections and the one they can not be: the parallelizable portion is almost 95%. Thanks to the Amdahl's law, we can predict the maximum reachable speed up for 8 processors:

$$Speedup(8) = (1 - 0.95) + 8 * 0.95 = 7.65$$

The goal here is not to reach at any cost this maximum speed up, but to give a better comprehension of the parallelization.

3) Design, implementation, test and debug: As it has been discussed previously, the higher in the callgraph we try to parallelize, better should be the performances. Indeed, we minimize the overhead effects of thread creations. In this paragraph, we will present the different approaches we tried, and try to explain why they failed.

a) Parallelizing among the images: The higher we can try to parallelize is at the level of `c9nr_top` call. This function is mainly composed of a for loop. Each iteration of this loop represents one frame of the video. The easiest way to parallelize it should be to denoise several images in the same time. Before to spend a long time reading the source code, we can easily try to parallelize. Trying to understand the data dependences here is really a hard work. There are about 20 pointers. It is not obvious at all how the pointers are allocated from one iteration to the other.

The implementation is easy. We add `#pragma omp parallel for` just above the main loop. Then, we move the declaration and the allocation of the pointers which will contain images data inside the for loop. It will produce an overhead due to the memory allocation for each iteration, but it is a simple way to firstly test our implementation. The resulted performance are satisfying, knowing the overheads due to the memory allocation.

However, the tests are negatives. There are binary differences between a denoised images produced by the sequential algorithm and one produced by the parallel algorithm. Looking further, we can notice that the first produced image does not have any differences. This points out one of the main problem for parallelize *TNR*. As it is a spatio-temporal denoiser, the algorithm takes into account the previous image, but the previous *denoised* image. This is explained in the figure 5. This data dependence is very strong, as we must wait for the previous image to be generated (i.e. the previous image must have been processed by the last of the 6 functions) before we can begin to compute the next image. Even a pipeline design (described in figure 5) is not possible.

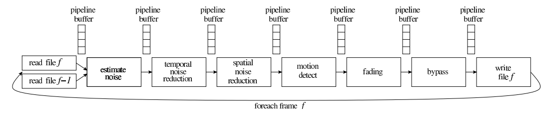

Fig. 5. A pipeline design. There exists a data dependence problem among the frames. To denoise a frame, we need the current frame and the previous denoised one.

b) Parallelizing among the denoising processes: Then we tried to naively parallelize the 6 time consuming functions. We call this *massive parallelization*. This design is presented in figure 6. Each function is seen as a task, and these task can run in parallel for each iteration.

```
#pragma omp parallel {
#pragma omp task
  fading(. . .);
#pragma omp task {
  *pucNLE = estimate_noise(. . .);
  motion_detect(. . .);
  }
#pragma omp task {
  temporal_noise_reduction(. . .);
  temporal_noise_reduction(. . .);
  temporal_noise_reduction(. . .);
  bypass(. . .);
```

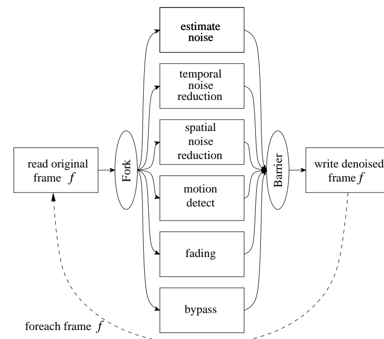

Fig. 6. A naive massive parallelization design. It must be noticed the strong existing barriers, from where load balancing problems could appear.

```
    bypass(. . .);
    spatial_noise_reduction(. . .);
  }
}
```

We can predict that the performance should be quite poor. Regarding to the sequential analysis (see figure IV-B2), the 6 functions does not have the same execution time average. As it has been discussed above, the load balancing should be bad: the threads should all wait for the thread executing `fading` for each iteration. We can try to distribute in a better way the load, aggregating the functions in order for each thread to take about one third of the execution for each execution (see pseudo code example in 6). However, this kind of parallelization is very architecture dependent. Here, it could be efficient with 3 computation units.

Trying to implement this design, we faced a second data dependence problem. There is a strong dependence between almost each functions. Each function take into parameter one or several results from previous functions. This involves that a given function must wait that the previous function finish its execution; moreover, they must be executed in the right order. There could be a solution, where the next function can work on already denoised areas in the image. But this solution involves a long expertise of the source code, and a very hard implementation.

This naive solution seems not feasible.

c) Fine grain: As the previous approaches did not work, we must work deeper in the callgraph. The next level in the callgraph is the last: we work now at the *leaves stage*. The parallelization will be designed inside the 6 denoising functions.

All the functions have the same pattern: a *nested for loop*. These nested loops work on the image matrix, applying primitive calculation on each cells. A first look into these loops let us think that each iteration should be independent. This assumption will be confirmed by the test phase.

We present here the parallelization for the simplest of the 6 functions, `temporal_noise_reduction`. The parallelization directive is applied to the outer loop. OpenMP is able to efficiently parallelize a nested loop like this. Parallelizing the inner loop would be a very bad design, generating a huge overhead for the multiple thread creations.

Listing 1. Example of a parallelized loop in TNR program

```
#pragma omp parallel for
for ( y = 0;y < ulYInSize;y++ ) {
  for ( x = 0;x < ulXInSize;x++ ) {
    P = * ( pucPPBuf + y * ulXInSize + x );
    C = * ( pucCBuf + y * ulXInSize + x );
    * ( pucOutBuf + y*ulXInSize + x ) = ( U8 ) ( ( C * (
        32 - ulTempFactor ) + P * ( ulTempFactor ) + 16 )
        >> 5 );
  }
}
```

d) : Tests shows that something is wrong. As described in paragraph IV-B3a, there are differences in the denoised images. TNR application is then executed with Thread Checker. A brief overview of the results are shown in table I.

`int tab[size_i][size_j]`
`for (i = 0; i < size_i; i++)`
`  for (j = 0; j < size_j; j++)`
`    tab[i][j] = ...`
	`int tab[size_i][size_j]`
	`for (j = 0; i < size_i; i++)`
	`  for (i = 0; j < size_j; j++)`
	`    tab[i][j] = ...`

TABLE II
COMPARISON BETWEEN GOOD AND BAD NESTED LOOP IMPLEMENTATION. MEMORY ACCESS TO THE MATRIX IS DONE BY `TAB[I][J]`. THIS IS EQUIVALENT TO `(*TAB + I * SIZE_J + J)`

The first error can be observed in code 1. There is data race on the pointer variable `P` (there is the same error about `C`). There are temporary variables representing the current working cells. A solution should be to declare these variables inside the loops. We prefer the OpenMP solution, declaring them as `private` for the parallel region. Instances of these variables will be unique for each threads.

The second error is a counter access which is not protected. We protect the access to this counter with `#pragma omp atomic`. In this case, we can not be sure that this counter must be shared among the threads. The next execution will inform us if we are right or wrong.

e) Enhancement: The tests finally pass. The design is then approved. The performance of our parallel implementation is now measured. The experiences are performed on *idkoiff*, a 8 x AMD Opteron 875, which gives a total of 16 cores. The program is compiled with gcc-4.3.3. OpenMP is integrated to gcc. We measure the speedup, executing the TNR application with 1, 2, 4, 8, and 16 threads.

We first executed a modified version of TNR. In this version, we voluntarily inverted the two nested loop for all the 6 parallelized functions. An example of this error is presented in table II. This gross error, which can cause bad performances in sequential execution, is catastrophic for parallel execution. Indeed the memory is not accessed linearly, which causes cache misses. This programing fault has been done in order to show the performance differences which can result. This difference is presented in figure 7.

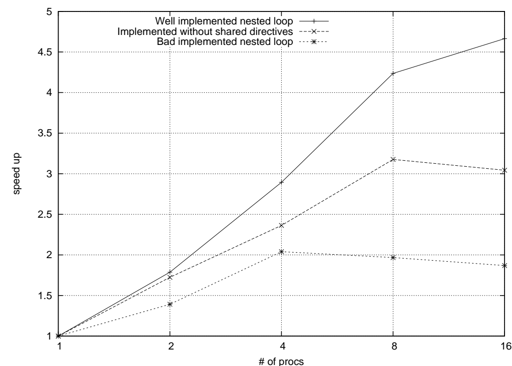

Fig. 7. Different implementation speed up on idkoiff with a 720 x 500 resolution video

Then we can tweak the OpenMP directives. We must look

ID	Short Desc	Description	1st access	2nd access
4	Write → Write data race	Memory write of P at ”c9nr_noise_reduction.c”:20 conflicts with a prior memory write of P at ”c9nr_noise_reduction.c”:20 (output dependence)	“c9nr_noise_ reduction.c” :20	“c9nr_noise_ reduction.c” :20
7	Read → Write data race	Memory write of count at ”c9nr_noise_estimation.c”:87 conflicts with a prior memory read of count at ”c9nr_noise_estimation.c”:87 (anti dependence)	“c9nr_noise_ estimation.c” :87	“c9nr_noise_ estimation.c” :87

TABLE I
THREAD CHECKER OUTPUT FOR TNR EXECUTION

closely to the `shared` directives. For instance, in the source code shown above (see 1), we must look closely to the variables `pucPBuf` and `pucCBuf`. There are the matrices pointers to the previous and current frames. We can notice that during this parallel region, these matrices are only read. If `pucPBuf` and `pucCBuf` are not declared as shared, their content is copied for each thread at before each execution of this parallel region. Applying good shared policy among the 6 parallel regions improves another time the performances (see 7).

The scheduling can also be defined. We choose a static scheduling. Indeed the primitive operations in the inner loop are very stable, the execution time of each iteration should be almost equal. There is not load balancing issue in this case. We also define to OpenMP how to split the iterations among the threads.

The last step of enhancement is to transform the OpenMP directives into native thread code. A good design would be to create the threads only once, instead of recreating them for each frame.

V. CONCLUSION

We have presented a parallelization method for embedded applications, inspired from the software engineering development models. It seems important that such methods exists; parallel development is still young, and in constant evolution. Even for small parallelization project, it is difficult to choose the good parallel API, to find how to start in the existing source code. Lots of mistakes have been done in software development before software engineering; the same thing must not append for parallel development.

This model still contains weaknesses, partly dues to the tools. The most critic one is the enhancement one. Rewrite the application with native threads, inspired by the OpenMP patterns is not a good solution. I see two perspectives: either modify the OpenMP tool, enabling the generation of native embedded thread program; or find a way to extract the intermediate code generated by OpenMP, making the parallel patterns more explicit.

Works such as the Automatic Profile-Driven Parallelization[12] are very promising. Indeed, if some automatic parallelization is one day efficient, a methodology like the one defined here would be obsolete, as it would be able to produce quite good parallelism with very low effort. parallelization methodology

REFERENCES

[1] S. Shende and A. Malony, “The TAU parallel performance system,” *International Journal of High Performance Computing Applications*, vol. 20, no. 2, p. 287, 2006.

[2] N. Bell Laboratories, Murray Hill, “Unix programmer’s manual, prof command.”

[3] S. Graham, P. Kessler, and M. Mckusick, “Gprof: A call graph execution profiler,” in *Proceedings of the 1982 SIGPLAN symposium on Compiler construction*. ACM New York, NY, USA, 1982, pp. 120–126.

[4] K. Lindlan, J. Cuny, A. Malony, S. Shende, B. Mohr, R. Rivenburgh, and C. Rasmussen, “A tool framework for static and dynamic analysis of object-oriented software with templates,” in *Supercomputing, ACM/IEEE 2000 Conference*, 2000, pp. 49–49.

[5] L. Djoudi, D. Barthou, P. Carribault, C. Lemuet, J. Acquaviva, and W. Jalby, “MAQAO: Modular Assembler Quality Analyzer and Optimizer for Itanium 2,” in *The 4th Workshop on EPIC architectures and compiler technology, San Jose*, 2005.

[6] N. Nethercote and J. Seward, “Valgrind: A framework for heavyweight dynamic binary instrumentation,” in *Proceedings of the 2007 PLDI conference*, vol. 42, no. 6. ACM New York, NY, USA, 2007, pp. 89–100.

[7] J. Weidendorfer, M. Kowarschik, and C. Trinitis, “A tool suite for simulation based analysis of memory access behavior,” *LECTURE NOTES IN COMPUTER SCIENCE*, pp. 440–447, 2004.

[8] W. Cohen, “Multiple architecture characterization of the linux build process with OProfile,” in *Workshop on Workload Characterization*, 2003.

[9] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou, “Cilk: An efficient multithreaded runtime system,” *ACM SigPlan Notices*, vol. 30, no. 8, pp. 207–216, 1995.

[10] http://gcc.gnu.org/wiki/Graphite.

[11] D. Maydan, J. Hennessy, and M. Lam, “Effectiveness of Data Dependence Analysis.”

[12] G. Tournavitis and B. Franke, “Towards Automatic Profile-Driven Parallelization of Embedded Multimedia Applications.”

[13] B. Nichols and D. Buttlar, *Pthreads programming*. O’Reilly Media, Inc., 1996.

[14] L. Dagum, R. Menon, and S. Inc, “OpenMP: an industry standard API for shared-memory programming,” *IEEE Computational Science & Engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[15] A. Jannesari, K. Bao, V. Pankratius, and W. Tichy, “Helgrind+: An efficient dynamic race detector,” in *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing-Volume 00*. IEEE Computer Society, 2009, pp. 1–13.

[16] U. Banerjee, B. Bliss, Z. Ma, and P. Petersen, “Unraveling Data Race Detection in the Intel® Thread Checker,” in *First Workshop on Software Tools for Multi-core Systems (STMCS), in conjunction with IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. Citeseer, 2006.

[17] http://software.intel.com/en-us/articles/a-methodology-for-threading-serial-applications/.

[18] W. Royce, “Managing the development of large software systems: concepts and techniques,” in *Proceedings of the 9th international conference on Software Engineering*. IEEE Computer Society Press Los Alamitos, CA, USA, 1970, pp. 328–338.